TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DIPLOMARBEIT

Reinforcement Learning in Agent based Modelling

ausgeführt am Institut für
Analysis & Scientific Computing
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof.i.R. Dipl.-Ing. Dr.techn.
Felix Breitenecker

und
Projektass. Dipl.-Ing. Dr.techn.
Nikolas Popper

durch
Sebastian von der Thannen
Matrikel-Nr. 01226852

Wien, 25.10.2018

(Unterschrift Verfasser) (Unterschrift Betreuer)

Danksagung

Als erstes möchte ich meinen herzlichsten Dank an meine beiden Betreuer, Prof. Breitenecker und Dr. Popper, ausrichten, die mir die Möglichkeit gegeben haben, in diesem spannenden Forschungsbereich eine Diplomarbeit zu schreiben. Zusätzlich bedanke ich mich bei DI Dominik Brunmeir, der mich ebenfalls während der gesamten Arbeit unterstützt und betreut hat.

Weiters bedanke ich mich bei all meinen Studienfreunden für die zahlreichen interessanten, manchmal auch sinnfreien, vor allem aber unterhaltsamen Gespräche. Ohne ihre Unterstützung wäre das Studium wohl deutlich steiniger und farbloser verlaufen.

Ebenso danke ich meinen Freunden abseits des Studiums für die etlichen späten gemeinsamen Stunden, welche mich immer auf andere Gedanken bringen konnten. Ihre Ohren standen immer offen für jedweden Kummer.

Zu guter Letzt gebührt mein Dank natürlich meiner Familie und speziell meinen Eltern, die mich vor allem finanziell und psychisch bedingungslos unterstützt haben. Ohne sie wäre das Studium nicht möglich gewesen.

Kurzfassung

In den letzten Jahren konnte die Forschung im Bereich des maschinellen Lernens in Verbindung mit künstlichen neuronalen Netzen enorme Fortschritte erzielen. Insbesondere im Bereich des bestärkenden Lernens wurden viele Durchbrüche erzielt (z. B. das Spielen von Atari-Spielen und AlphaGo von Google Deep Mind). Die meisten dieser behandelten Probleme umfassen einen einzigen Agenten, der sich in einer Umgebung befindet, mit welcher er interagieren kann. Ziel des Agenten ist es dabei, mit Hilfe einer Belohnungsfunktion herauszufinden, welche Aktionen ihn zur maximalen Belohnung führen.

Mit den selben Techniken versucht diese Arbeit einen generellen Rahmen zu schaffen, um bestärkendes Lernen in der agentenbasierten Modellierung einzusetzen. Anschließend wird dieses Konzept an einem agentenbasierten Räuber-Beute Modell angewendet und evaluiert.

Da einige Modelle es erlauben, die Agenten in Gruppen einzuteilen, wie es zum Beispiel für das Räuber-Beute Modell der Fall ist, muss für jede dieser Gruppen eine eigene Belohnungsfunktion definiert werden. Dadurch kann jede Gruppe ihr optimales Verhalten erlernen. Diese daraus resultierende Verhaltensfunktion, die durch ein neuronales Netz approximiert wird, führt den Agenten zu einer optimalen Verhaltensweise, um die erwartete zukünftige Gesamtbelohnung, basierend auf seinem aktuellen Zustand, zu maximieren. Im Vergleich zu herkömmlichen agentenbasierten Modellen kann dieser Ansatz den Modellierungsprozess vereinfachen und gleichzeitig die Verzerrung des Modells verringern, da die vom Modellierer festgelegten Verhaltensregeln durch eine Belohnungsfunktion ersetzt werden.

Diese Arbeit versucht verschiedene Ansätze aufzuzeigen, um sowohl eine sinnvolle Belohnungsfunktion, als auch gute Parameterwerte zu finden. Damit soll eine globale Konvergenz bei der Modellierung komplexer Interaktionen zwischen Agenten in einer Umgebung gewährleistet werden.

Abstract

In recent years, huge progress has been made in machine learning using neural networks as function approximators. Especially in reinforcement learning, extensive research is ongoing and a lot of breakthroughs were achieved (e.g. playing atari games and AlphaGo by Google Deep Mind). Most of these problems involve a single agent thrown into an environment where it has to figure out how to perform optimally based on given rewards for each action.

Using these techniques, the thesis aims to develop a general framework for agent based modelling using reinforcement learning and evaluate the results on a pred-ator-prey model using usual approaches such as the Lotka-Volterra equations or rule based models.

As some models require a classification of agents in groups, as it is for the predator-prey model, each group of agents demand their own reward function in order to find its optimal policy. This policy function, which will be approximated by a neural network, gives the agent advice for the best action to take with focus on maximising the agent's expected total future rewards based on their current state. Compared to usual agent based models, this approach can simplify the modelling process while decreasing the bias of the model since hard coded behavioural rules are replaced by a reward function.

This thesis tries to explore different approaches to find both, a meaningful reward function and good parameters to assure global convergence when modelling complex interactions between agents in an environment.

Contents

1. Introduction

Evolving intelligence occurs naturally in our world. From small cyto-creatures to animals including humans, all living beings have some kind of intelligence that evolves in lifetime through environmental changing interactions plus experiencing some kind of feedback. Therefore using reinforcement learning can be seen as a natural approach in biological or social agent based models. For instance, Wang et al. used reinforcement learning to characterise cell movement of C. elegans [27], Yang et al. studied population dynamics in an thousands of agents predator prey model [29] and Google DeepMind studied cooperation and defection in multi-agent social dilemmas [12]. In settings where agents suffer a lack of information such as a partially observable environment or unawareness of opponent's policies, game theory and policy optimisations intertwine [9].

In order to make things simpler, one can assume that a creature's life is nothing more than a time-discrete, stochastic control process, i.e. a Markov decision process (MDP). Everything that can be observed by the senses of the creature forms the state space. All possible actions that can be taken form the action space. These spaces are often discretised as well as the time. Now living in an environment the creature has a state, determined by it's senses. Taking an action may cause a change in the environment and therefore may trigger a change in it's state. This state transition can be a stochastic process, such that two identical state-action pairs don't have to lead to the same state transition. Now in order to measure the quality of the actions one assumes a third observable quantity called the reward. This is a real valued feedback from the environment to the creature after each action that rates the quality of the new state. In real world, this feedback corresponds to punishment or achieving a goal, e.g. pain, death or satisfying needs and positive emotions. Each individual seeks to learn a behaviour that maximises it's cumulative rewards. Such an optimal behaviour is the solution to this MDP.

The process of interacting with the environment leads to time series, called trajectories

$$\tau = (s_0, a_0, r_1, s_1, a_1, ..., r_T, s_T)\,,$$

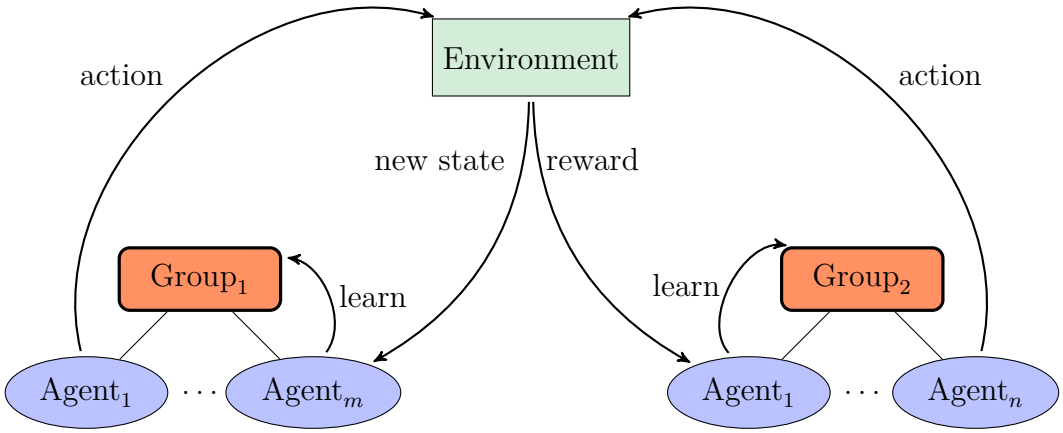

Figure 1.1: Environment with State-Action-Reward-State setting.

which may be finite in case of a goal achievement, or fail. The creature's policy π is often modelled by a probability distribution

$$\pi(s) = P(a \mid s),$$

but also may be deterministic.

In the following we classify different kind of creatures, i.e. creatures that show the same behaviour, in groups and call creatures of the same group agents. Figure 1.1 shows the simplified life process of groups of agents interacting with an environment, where each agent produces it's own trajectory.

Following these trajectories, the value of each state can be defined as

$$V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_{t+1} \mid s_0 = s\right].$$

Whereby the problem formulation gets precise by adjusting the policy in order to maximise the expected total future rewards $V_\pi(s)$ for each state. Furthermore we take another simplification. Since agents of the same group share the same reward function, we can solve the MDP only once for each group, instead of solving the MDP for each agent, and collect all trajectories of the agents as sampled trajectories for estimations.

Figure 1.2 shows the group control framework, where the group's policy and value function are approximated. We will use neural networks as function approximators since their flexibility makes them powerful. Also it can be seen as a

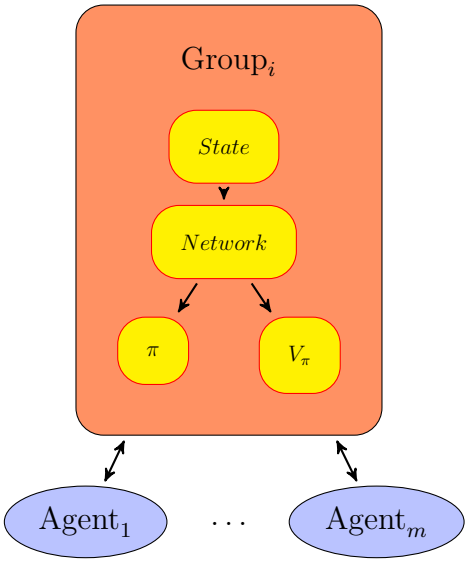

Figure 1.2: Unfold group control.

natural approach since neural networks are a model for a biological neural network such as the animal's brain [14]. Further details are given in section 5.

The optimisation targets L_i, that are to be minimised, are defined as

$$L_1(\theta) = \sum_t (R_t - V_\pi(s_t))^2$$
$$L_2(\theta) = -\mathbb{E}\left[V_\pi(s)\right],$$

where R_t is the sampled cumulative reward

$$R_t := r_t + \gamma R_{t+1},\ R_T = V_\pi(s_T).$$

This optimisation problem is highly nonlinear but can be solved using gradient descent methods, described in section 4 and 5.

2. Motivation

The combination of agent based modelling with reinforcement learning is the goal of this thesis. This means that the classical ABM is given a new perception, i.e. the definition of rules gets replaced by a reinforcement learning framework which is controlled by a reward function only. However the question remains, why and when it makes sense to apply reinforcement learning to agent based modelling and if this new approach gives any benefit over the classical one. This chapter tries to give a motivation and an overview, that explains the differences of these two approaches.

Therefore it is necessary to know how a classical agent based model is implemented and how it works. Chapter 3 tries to give a definition of classical ABMs and indicate their limits, advantages and disadvantages. Essentially, ABMs can be applied whenever multiple agents interact with each other and the environment, based on some predefined rules. Together they generate a collective complex dynamic system. Especially in areas like social science, economics and biology agent based modelling can be found.

In order to understand the advantages of reinforcement learning in agent based modelling one has to grasp the purpose that RL tries to pursue in general. Chapter 4 gives a theoretical insight of this special type of machine learning paradigm. Basically it is about optimising the behaviour of an agent, that is trying to reach a goal, while acting in an environment. The only information that he receives is the actual state, that he is in, and a valuation of this state, called the reward. Such settings often occur in game theory, economics and optimal control, thus reinforcement learning is mostly applied in these fields.

In addition, chapter 5 gives a theoretical background about neural networks as function approximators. They are used in this thesis in connection with reinforcement learning. One can get an idea of their structure, functionality and how they can be used as approximators.

As a roundup, chapter 6 combines these three concepts and presents a practical approach of the implementation of reinforcement learning in agent based modelling using neural networks as function approximator. Also this combination is applied to a predator prey model as a validation.

Beforehand, we try to motivate this combination.

The essential part in classical agent based modelling is the definition of behavioural rules. Especially it is necessary to define breakpoints, i.e. when agents should act in the defined manner. In most cases, these breakpoints and behavioural rules are extracted from real world data and observations, that one intends to reproduce. This strategy however, may hold the danger that wrong assumptions and implications are made by the modeller. Particularly this can happen more likely if the data, that is obtained from real world observations, doesn't reveal a clear pattern or structure, that would allow logical implications. In these situations, the bias of the modeller flows into the model and thus the simulation. Dependent on the application and the purpose of the model, this may have a negative effect. Usually there are two scenarios in which ABMs are used.

On the one hand, there is the classical problem where the interest lies in the dynamic of the system, that emerges during simulations. In this case it is clear which rules need to be defined. It is rather the emergent behaviour of the total system that is unknown previously. For this classical scenarios it doesn't make much sense to use reinforcement learning. The main problem is the definition of the reward function and just that would be unreasonable by the nature of this problem. How should it be feasible to determine a meaningful reward function on the outcome, if it is not known, while at the same time, the policies of the agents are already evident. Thus it will be the second scenario, where the implementation of reinforcement learning may have a beneficial effect.

This scenario can be seen as the inverse problem, where the dynamic of a system is observed, but it is unclear how the agents behave such that this emergence occurs. This makes it necessary to reverse engineer the system. Trying to find the breakpoints and behavioural rules for the agents, that lead to the desired system dynamics, can be a hard and expensive task, with lots of trail and errors. Thus, the use of reinforcement learning can help to solve this task. The definition of a reward function, that gives positive rewards on the desired outcome and negative else, can help finding behavioural rules and breakpoints automatically such that the desired system dynamic is generated.

Though, it is important that the reward function is independent of the agent's actions. It should only be defined on the states, which are also generated by the dynamic of the system. Otherwise it would mislead the purpose of using reinforcement learning in this context, because agents would start to draw rewarded actions. Then this behaviour could have been replaced by the definition of behavioural rules in the first place. This also shows that reinforcement learning in agent based modelling can accomplish the same as using the classical rule based approach, yet with a long way round. The same doesn't apply in reverse.

In some cases, the problem meets somewhere in between these two mentioned

scenarios. This holds, for example, for the predator prey model, which was used as a validation in chapter 6. On the one hand, the behaviour of the predator and the preys is well studied and can be defined on a common sense. Predators try to hunt down preys by pursuing them and preys try to escape the predators. On the other hand, the only thing we can say for sure is that the predators need to hunt, so as not to starve, and the preys need to avoid the contact with predators in order to survive. But we don't know exactly how they achieve this.

The possibility to approach this problem in these two ways, makes it reasonable as a validation. If the reinforcement learning implementation leads to the same results as in the classical scenario, based on real world observations, both approaches would be legitimate. Indeed, this is shown in chapter 6.

Now, we could simply implement the behavioural rules as proposed by the classical agent based models. Or we could use reinforcement learning by defining a reward function for these groups. The latter approach is now in our interest.

The reward function should represent the desired system dynamic, where preys get a negative reward if they got in contact with a predator and predators get a positive reward when they attack preys. This is exactly what is done in chapter 6. The result of this implementation indicates the exciting potential of reinforcement learning in agent based modelling. Not only that the agents learned the same behaviour as it is proposed by classical ABMs. Beyond that, the predators learned a cooperative behaviour without any communication. They found an optimal policy that made them more successful in hunting down preys.

In a classical basic predator prey ABM it would be hard to determine the optimal breakpoints on which a predator should stop pursuing a prey and join a group of other predators for cooperative hunting. In the reinforcement implementation these breakpoints and behavioural rules where found automatically by the algorithm. However things look more promising as they actually are. It can be hard to find optimal hyperparameters for the RL framework that lead to a successful learning process. This always depends on the complexity of the model and system. Nevertheless, if these parameters are found, the ABM can become more realistic in a natural way.

This should give an idea about the possibilities and limits that arise when using reinforcement learning combined with agent based modelling in general.

3. Agent Based Modelling

Agent based modelling is an individual based modelling technique that can be applied to analyse complex dynamic systems. The dynamic of this system is generated by interactions of several individuals, the so called agents, in an environment. Such systems often occur in social science, biology, economics, which makes agent based modelling popular in these fields. One advantage of this modelling technique is it's simplicity of defining intelligible rules on a microscopic level that lead to a complex dynamic on a macroscopic view. This concept is a typical bottom-up modelling approach. The base elements are described in great detail first, followed by linking together available information to form a bigger top-level system. Where, on the other hand, a top-down approach would start on an abstract macroscopic level and subsequently breaking down the problem into smaller sub-systems.

Different from equation based modelling techniques, this rules are insightful and easy to define, especially for researches that don't have a technical background. Also it is easy to involve targeted stochastic parameters. Thereby the model gets an additional dynamic and may seems more realistic. Though, finding stationary solutions or analyse the stability and convergence of ABMs is hard.

Since ABM was developed by mathematicians, computer scientists, economists and even political scientists (Conway, Schelling, Axelrod et al.), various types and definitions of ABMs exist, reaching from cellular automata to complex adaptive systems. However the following section will attempt an own approach of a basic formulation of ABM.

3.1 Principle Components

In general, a scientific model always aims to represent features of the real world by simplifying its complexity and reducing the problem to its main components but at the same time reproduce the real world observations in a realistic, if not exact, manner. Thus, disassembling a complex system, by focusing on some carefully selected aspects, into an agent based model is in our interest. Basically ABMs have two main components. The agents and the environment.

Environment

The environment can be seen as a pot, metaphorically speaking. It forms a necessary limit for the agents in which they are able to experience, move and interact with each other. Still it can be as manifold as a pot divided into base and side face as space and time.

The most common and straight forward choice for the space is a bounded d-dimensional discrete or continuous subset of $\mathbb{R}^n$. This includes manifolds like the torus, that makes it possible to simulate an unbounded domain.

For time, less choices are available, since it has only one dimension. It can be modelled continuously but, typically one chooses a discrete axis with some step size.

Supplementary, an environment holds some other variable and constant physical properties that can or cannot be observed by the agents. A variable physical property could be a morphing space like flora and fauna or available food spread over the domain, but also climate or catastrophes. Constants could be for example the space capacity or forces, e.g. gravity. The design of those properties can have a significant impact on the stability of the whole model. Preserving stability may be an essential task for the environment.

In addition, the environment controls the agent's access to information. Spatial such as nearest neighbour, distance, collision but also time dependent information like velocity or maybe even forecast or review. Also the environment could provide and manage communication among agents. At some point this certainly interferes with properties of individual agents.

Furthermore, in a different perception, it is also possible to understand the environment as a special type of agent, with whom other agents can interact.

Agents

The most substantial part in ABMs are the agents. They are responsible for the system's dynamic and animate the whole model. Primarily agents are divided into different groups. These groups represent different behaviour in the first place. Secondly they can obtain different kinds of physical properties such as life span, birth rate or robustness but also maximum speed or agility.

In order to obtain an individual behaviour among group members, agents usually have individual physical properties like age, health and of course environmental based properties like position and velocity.

Based on all these properties and cognition, obtained from the environment, group and individual, one can now specify behavioural rules. These rules should be chosen carefully to keep them irreducible and as simple as possible for a smooth evaluation - calibration loop (see figure 3.1) of the model. As for properties, agent's

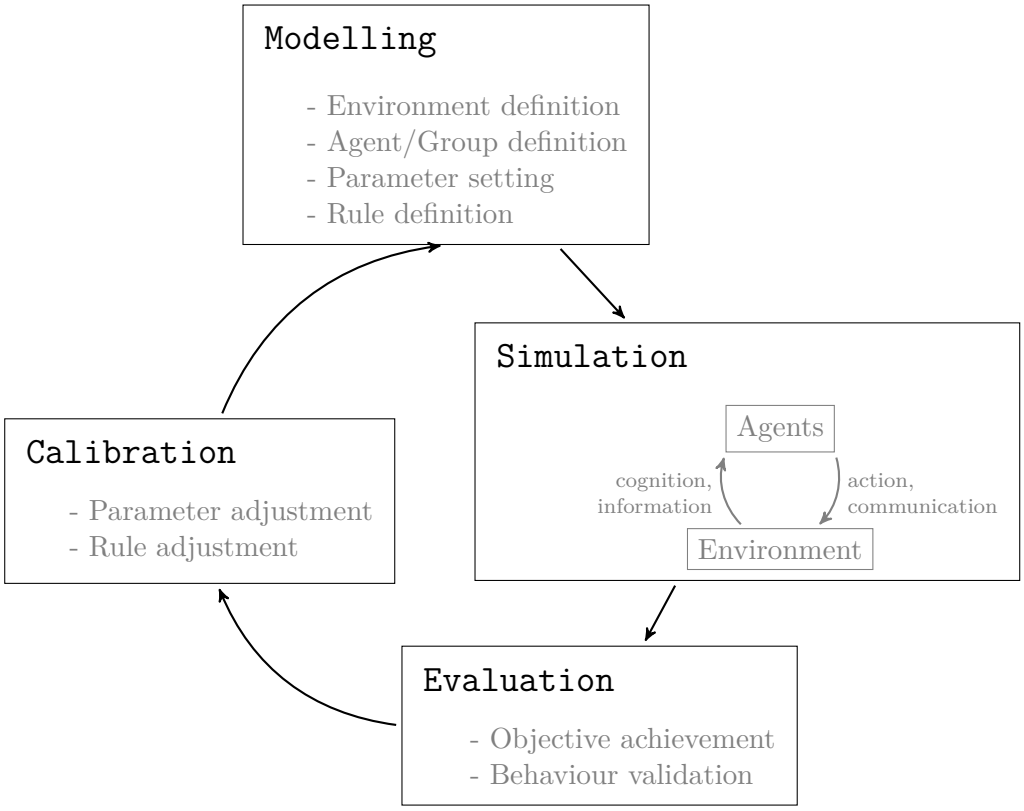

Figure 3.1: MSEC - loop

behavioural rules don't have to be static. Agents could change their behaviour over time (e.g. learn, adapt, defect to other groups, etc.).

In general there there are no restrictions to the level of detail of the model. This suggests that ABMs are rather a general concept than a concrete modelling instruction.

3.2 Characterisation

As Bonabeau [2] proposed, ABM has three main characteristics.

Capturing emergent behaviour

This feature becomes clear if one thinks about a flock of birds. That is a typical system that can be modelled by agents. While their group behaviour as a whole is captivating, the behaviour of a single bird is quite unexciting: Simply avoid collisions and track neighbouring birds. Though, if one of these rules gets omitted,

the flock either implodes or drifts apart. So even few and simple rules can result in an impressive and even counterintuitive emergent behaviour.

Natural description of a systems

There are a lot of problems, like population dynamic in a predator prey scenario or flocking behaviour, that can be modelled in various ways. Though, if we compare the Lotka-Volterra equations to a basic predator prey ABM, where the population size gets extracted, it is clear that the ABM gives a more natural description.

First of all, differential equations tend to smooth out solutions. In real world dynamic systems however, often discontinuous fluctuations or perturbations occur, that can't be reproduced with DEs. In fact these fluctuations come naturally with ABMs. This not only makes them a natural description but also makes them seem more realistic.

Flexibility

As indicated in section 3.1, this third characteristic should not surprise. Not only is it possible to scale the model in detail and extent during the modelling process, but also adding features or agents even during the simulation is feasible. Anyway it is often better to start with a simple and more transparent model, and increase complexity with the validation - calibration loop later.

Advantages

One big advantage of ABMs compared to other modelling techniques is for sure, as already mentioned, its transparent and simple design, yet the ability to model complex systems. In addition, the flexibility and scalability in space and interaction rules of ABMs makes them unrivalled. As it was suggested in section 3.1, there are virtually no limits. Compared to other modelling techniques, it doesn't take much of an effort to extend the model.

Along with that comes the possibility to naturally describe a system with rules that don't need a mathematical or physical formulation or structure. This makes it an attractive modelling technique to people that don't have a technical background and makes it easy for them to adapt parameters and rules according to their experience and knowledge of real world data without the need of a modelling expert.

Disadvantages

If we recall the flock example of emergent behaviour, we saw that small changes in behavioural rules can result in a totally different outcome. This means that these

systems are quite unstable and rules, that fundamentally determine the wanted emergence, sometimes have to be identified first. Missing data or knowledge of agent behaviour can lead to a time consuming and exhausting calibration process. Due to the flexibility of ABMs, the implementation is more work than other modelling techniques. A general framework can only be built for some basic problems. With an increasing complexity and size of a model, the simulation process requires powerful computational resources and runtime. A mathematical analysis of convergence, stability and correctness is nearly impossible.

4. Reinforcement Learning

Reinforcement learning is part of machine learning, that mainly differs from other machine learning methods by the lack of a priori known data in terms of experience or samples. In supervised learning, a major counterpart of reinforcement learning, labeled samples are used, on which the machine is trained. For instance take a set of pictures of handwritten digits that are labeled with the corresponding number that is shown in the picture. The machine takes these labeled pictures and tries to find underlying patterns in order to predict the correct digits on new, previously unknown pictures independent of the training set. Another paradigm of machine learning is unsupervised learning where it suffices to have unlabelled raw samples. While feeding the machine with samples it is generating labels on the fly and trying to find an underlying structure. In reinforcement learning one does not assume any a priori data samples of the system, but rather a given environment where the machine explores the space of data by its own and labels the data with the help of the corresponding received rewards (e.g. hot or cold) following his actions. The goal of the exploration is to maximise the rewards received. The idea of reinforcement learning arose from psychology and tries to model the process how creatures, such as humans, learn. Thus it plays a major role in research of artificial intelligence.

> 'When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.' [24, p. 1]

This environment can be formulated as a Markov decision process (MDP).

4.1 Formulation of a Markov Decision Process

Most of the following definitions, proofs and concepts can be found in [24]. This section should summarise the most important insights and give a general overview of the setting. First it is necessary to formalise the framework of reinforcement learning in a mathematical context.

Definition 4.1.1 (Markov decision process)**.** *A Markov decision process (MDP) is a tuple (S, A, R, T, γ), consisting of the space of states S, the space of actions A and a reward function $R : S \times A \times S \to \mathbb{R}$. T gives a state transition probability distribution $T : S \times A \times S \to [0, 1] : T(s, a, s') \sim P(s' \mid s, a)$, the probability to end in state s' when taking action a while beeing in state s. And γ a decay parameter.*

A Markov decision process is thus a time discrete stochastic process that demands a decision maker who takes the actions. This decision maker is called agent. The strategy of the decision maker is called policy.

Definition 4.1.2 (Policy)**.** *A policy π in general is a conditional probability distribution $\pi(a \mid s)$ on the action space A, dependent on state s. A policy is called soft, if $\exists\, \epsilon > 0 : \pi(a \mid s) \geq \epsilon > 0, \quad \forall a \in A, s \in S$. Let Π denote the set of all policies. In case of a deterministic policy, π often gets replaced by a mapping $\hat{\pi} : S \to A$ instead of writing $\pi(a \mid s) = \mathbb{1}_{\{\hat{\pi}(s)=a\}}$.*

While acting in this environment, the agent, equipped with a policy π, now produces stochastic trajectories

$$\tau = (s_0, a_0, r_1, s_1, a_1, ..., r_T, s_T)\,,$$

that may be infinite or eventually end after T steps in a final state s_T. For example when a goal is achieved or failed.

Definition 4.1.3. *The value function under a fixed policy π is defined as*

$$V^\pi(s) \coloneqq \mathbb{E}\left[\sum_{k=0}^{T-1-t} \gamma^k r_{t+k+1} \mid s_t = s\right],\ s \in S, \tag{4.1}$$

i.e. the expected total accumulated future reward under the policy π. Furthermore one can define the Q-function, also known as action-value function.

Definition 4.1.4. *The action-value function under a fixed policy π is defined as*

$$Q^\pi(s, a) \coloneqq \mathbb{E}\left[\sum_{k=0}^{T-1-t} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right],\ s \in S, a \in A \tag{4.2}$$

that, in addition to V, takes the first action into account. The connection between the value and the Q-function is evident

$$V^\pi(s) = \mathbb{E}\left[Q^\pi(s_t, a_t) \mid s_t = s\right].$$

Often, in addition, the advantage function is considered.

Definition 4.1.5. *The advantage function under a fixed policy π is defined as*

$$A^\pi(s,a) \coloneqq Q^\pi(s,a) - V^\pi(s). \tag{4.3}$$

It gives the advantage of taking action a in state s over taking an action according to the policy π.

The value function V^π and the Q-function Q^π fulfil a recursion, called the Bellman equation. For the value function the Bellman equation reads

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[\sum_{k=0}^{T-1-t} \gamma^k r_{t+k+1} \mid s_t = s\right] \\ &= \mathbb{E}\left[r_{t+1} + \gamma \sum_{k=0}^{T-1-t} \gamma^t r_{t+k+2} \mid s_t = s\right] \\ &= \mathbb{E}\left[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\right] \end{aligned} \tag{4.4}$$

and similar for Q^π.

Definition 4.1.6. *A policy π' is said to improve π, i.e.*

$$\pi' \geq \pi \quad \Leftrightarrow \quad V^{\pi'} \geq V^\pi \quad \Leftrightarrow \quad V^{\pi'}(s) \geq V^\pi(s) \quad \forall s \in S.$$

Lemma 4.1.1. *Let π', π be two policies, then*

$$\forall s \in S: \quad \mathbb{E}\left[Q^\pi(s,a) \mid a \sim \pi'(s)\right] \geq V^\pi(s) \quad \Rightarrow \quad V^{\pi'} \geq V^\pi.$$

Proof.

$$\begin{aligned} V^\pi(s) &\leq \mathbb{E}\left[Q^\pi(s,a) \mid a \sim \pi'(s)\right] = \sum_{a \in A} \pi'(a \mid s) Q^\pi(s,a) \\ &= \mathbb{E}\left[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t \sim \pi'(s)\right] \\ &\leq \mathbb{E}\left[r_{t+1} + \gamma\, \mathbb{E}\left[Q^\pi(s_{t+1}, a_{t+1}) \mid a_{t+1} \sim \pi'(s_{t+1})\right] \mid s_t = s, a_t \sim \pi'(s)\right] \\ &= \mathbb{E}\left[r_{t+1} + \gamma\, \mathbb{E}\left[r_{t+2} + \gamma V^\pi(s_{t+2}) \mid s_{t+1}, a_{t+1} \sim \pi'(s_{t+1})\right] \mid s_t = s, a_t \sim \pi'(s)\right] \\ &= \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^\pi(s_{t+2}) \mid s_t = s, a_t \sim \pi'(s), a_{t+1} \sim \pi'(s_{t+1})\right] \\ &\quad \vdots \\ &\leq \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid s_t = s, a_t \sim \pi'(s), a_{t+1} \sim \pi'(s_{t+1}), \dots\right] \\ &= V^{\pi'}(s) \end{aligned}$$

□

The same holds for the strict inequalities. That is useful for convergence analysis of policy iteration. For that purpose we introduce a specific type of policy.

Definition 4.1.7 (ϵ-greedy policy)**.** *A policy π_ϵ is called an ϵ-greedy policy if*

$$\pi_\epsilon \in \Pi, \quad \exists\, \hat{\pi} : S \to A : \quad \pi_\epsilon(a \mid s) = \mathbb{1}_{\{\hat{\pi}(s)=a\}}(1-\epsilon) + \mathbb{1}_{\{\hat{\pi}(s)\neq a\}}\frac{\epsilon}{l}, \quad \epsilon > 0,$$

with $l \coloneqq |A| - 1$.

Every ϵ-greedy policy is soft.

Theorem 4.1.1 (Policy improvement theorem)**.** *Let π be an arbitrary policy but soft, Q^π the corresponding action value function. Now the ϵ-greedy policy improvement π_ϵ is defined by choosing ϵ such that $\min_{a,s} \pi(a \mid s) \geq \frac{\epsilon}{l} > 0$, and $\hat{\pi}(s) \coloneqq \arg\max_{a\in A} Q^\pi(s,a)$ then π_ϵ improves π:*

$$\pi_\epsilon \geq \pi.$$

Proof. Let s be arbitrary and $a^* = \arg\max_{a\in A} Q^\pi(s,a)$, we can rewrite V^π

$$\begin{aligned} V^\pi(s) &= \sum_{a\in A} \pi(a \mid s) Q^\pi(s,a) \\ &= \pi(a^* \mid s) Q^\pi(s,a^*) + \sum_{a\in A\setminus\{a^*\}} \pi(a \mid s) Q^\pi(s,a). \end{aligned}$$

Expanding the sum and using $Q^\pi(s,a^*) \geq Q^\pi(s,a)$ and that $\pi(a \mid s) - \frac{\epsilon}{l} \geq 0$ gives the estimate

$$\begin{aligned} \sum_{a\in A\setminus\{a^*\}} \pi(a \mid s) Q^\pi(s,a) &= \sum_{a\in A\setminus\{a^*\}} \left(\pi(a \mid s) - \frac{\epsilon}{l}\right) Q^\pi(s,a) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon}{l} Q^\pi(s,a) \\ &\leq Q^\pi(s,a^*) \sum_{a\in A\setminus\{a^*\}} \left(\pi(a \mid s) - \frac{\epsilon}{l}\right) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon}{l} Q^\pi(s,a) \\ &= Q^\pi(s,a^*)(1 - \pi\left(a^* \mid s\right) - \epsilon) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon}{l} Q^\pi(s,a). \end{aligned}$$

Now by definition of π_ϵ

$$\begin{aligned} V^\pi(s) &= \pi(a^* \mid s)Q^\pi(s,a^*) + \sum_{a\in A\setminus\{a^*\}} \pi(a \mid s)Q^\pi(s,a) \\ &\leq \pi(a^* \mid s)Q^\pi(s,a^*) + Q^\pi(s,a^*)(1-\pi\left(a^* \mid s\right)-\epsilon) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon}{l} Q^\pi(s,a) \\ &= (1-\epsilon)Q^\pi(s,a^*) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon}{l} Q^\pi(s,a) = \sum_{a\in A} \pi_\epsilon(a \mid s)Q^\pi(s,a) \\ &= \mathbb{E}\left[Q^\pi(s,a) \mid a \sim \pi_\epsilon(s)\right] \end{aligned}$$

and finally with lemma 4.1.1

$$V^{\pi_\epsilon}(s) \geq V^\pi(s) \quad \forall s \in S.$$

□

Assume we iteratively do the policy improvement and set $\epsilon_{k+1} < \epsilon_k$ then, in particular there holds

$$\begin{aligned} V^{\pi_{\epsilon_k}}(s) &= (1-\epsilon_k)Q^{\pi_{\epsilon_k}}(s,a^*) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon_k}{l} Q^{\pi_{\epsilon_k}}(s,a) \\ &< (1-\epsilon_{k+1})Q^{\pi_{\epsilon_k}}(s,a^*) + \sum_{a\in A\setminus\{a^*\}} \frac{\epsilon_{k+1}}{l} Q^{\pi_{\epsilon_k}}(s,a) \\ &= \mathbb{E}\left[Q^{\pi_{\epsilon_k}}(s,a) \mid a \sim \pi_{\epsilon_{k+1}}(s)\right] \end{aligned}$$

for all $s \in S$ where $Q^{\pi_{\epsilon_{k+1}}}(s,\cdot)$ is not constant in a. Thus with lemma 4.1.1

$$V^{\pi_{\epsilon_k}}(s) < V^{\pi_{\epsilon_{k+1}}}(s)$$

Definition 4.1.8. *A solution for an MDP is called an optimal policy $\pi^* \in \Pi$ that maximises V^π.*

$$\pi^* \coloneqq \arg\max_{\pi\in\Pi} V^\pi$$

Maximising the value function V^π with respect to the policy π gives the optimal value function V^*

$$V^* \coloneqq \sup_{\pi\in\Pi} V^\pi.$$

The same holds for the optimal action-value function Q^* with the correspondences

$$\begin{aligned} Q^* &= \sup_{\pi\in\Pi} Q^\pi \\ V^*(s) &= \sup_{a\in A} Q^*(s,a). \end{aligned}$$

The optimal value function V^* fulfils the Bellman optimality equation

$$V^*(s) = \sup_{a \in A} \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right]. \tag{4.5}$$

To show that V^* exists, we introduce the Bellman optimality operator.

Definition 4.1.9. *The Bellman optimality operator $B : \mathcal{V} \to \mathcal{V}$, with $\mathcal{V} = B(S, \mathbb{R})$ the space of all bounded V^π for all policies π, is defined as*

$$(BV^\pi)(s) \coloneqq \sup_{a \in A} \mathbb{E}\left[r_{t+1} + \gamma\, V^\pi(s_{t+1}) \mid s_t = s, a_t = a\right]. \tag{4.6}$$

Theorem 4.1.2. *The Bellman optimality operator B is a contraction with the fixed point V^**

$$BV^* = V^*.$$

Proof. By definition 4.1.3 and 4.1.9 and equation 4.5 the optimal value function V^* is a fixed point of B, if it exists. Defining the metric $d(V^{\pi_1}, V^{\pi_2}) \coloneqq \sup_{s \in S} |V^{\pi_1}(s) - V^{\pi_2}(s)|$ makes $(\mathcal{V}, d)$, complete. It follows that

$$\begin{aligned}
d(BV^{\pi_1}, BV^{\pi_2}) =& \sup_{s \in S} |BV^{\pi_1}(s) - BV^{\pi_2}(s)| \\
=& \sup_{s \in S} \Bigg| \sup_{a \in A} \mathbb{E}\left[r_{t+1} + \gamma\, V^{\pi_1}(s_{t+1}) \mid s_t = s, a_t = a\right] \\
& - \sup_{a \in A} \mathbb{E}\left[r_{t+1} + \gamma\, V^{\pi_2}(s_{t+1}) \mid s_t = s, a_t = a\right] \Bigg| \\
\leq& \sup_{s,a} \left|\mathbb{E}\left[\gamma\left(V^{\pi_1}(s_{t+1}) - V^{\pi_2}(s_{t+1})\right) \mid s_t = s, a_t = a\right]\right| \\
=& \sup_{s,a} \left| \sum_{s'} P(s' \mid s, a)\, \gamma\left(V^{\pi_1}(s') - V^{\pi_2}(s')\right) \right| \\
\leq& \gamma \sup_{s'} |(V^{\pi_1}(s') - V^{\pi_2}(s'))| \\
=& \gamma\, d(V^{\pi_1}, V^{\pi_2}),
\end{aligned}$$

since

$$|\sup_x f(x) - \sup_x g(x)| \leq \sup_x |f(x) - g(x)|,$$

the linearity of $\mathbb{E}$ and

$$\forall s, a : \sum_{s'} P(s' \mid s, a) = 1.$$

Therefore the Bellman operator B is a contraction as $\gamma < 1$. By the contraction mapping principle there exists a unique fixed point V'. Thus $V^* = V'$ and an optimal π^* exists. □

In theory, this gives also a constructive approach how V^* can be found (called value iteration). Starting with an arbitrary V, $B^n V \to V^*$.

$$d(B^n V^{\pi_1}, B^n V^{\pi_2}) \leq \gamma\, d(B^{n-1} V^{\pi_1}, B^{n-1} V^{\pi_2}) \leq \gamma^n\, d(V^{\pi_1}, V^{\pi_2}) \to 0$$

and in particular

$$d(B^n V, V^*) = d(B^n V, B^n V^*) \leq \gamma^n\, d(V, V^*) \to 0.$$

The convergence towards V^* is independent of a policy π. In contrast to the ordinary Bellman operator

$$(BV^\pi)(s) \coloneqq \mathbb{E}\left[r_{t+1} + \gamma\, V^\pi(s_{t+1}) \mid s_t = s\right], \tag{4.7}$$

which has fixed point V^π, dependent on the policy π used in the expectation value. The proof follows the same lines as above.

If we get back to ϵ-greedy policies, it is clear that the optimal policy π^* will barely lie in Π_ϵ and not even be soft. Still we can approximate π^* by policies in Π_ϵ arbitrarily exact. To see this we show the following theorem.

Theorem 4.1.3. *If the optimal policy is non deterministic, then there always exists an optimal deterministic policy.*

Proof. Let π^* be the non deterministic optimal policy and V^*, Q^* the corresponding value, action value function. By the Bellman equation, V^* reads

$$V^*(s) = \sum_{s' \in S, a \in A} \pi^*(a \mid s) P(s' \mid s, a)(R(s, a, s') + \gamma V^*(s')).$$

Using the Bellman optimality equation, we can also write V^* as

$$V^*(s) = \sup_{a \in A} \sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V^*(s')) = \sup_{a \in A} Q^*(s, a)$$

If A is compact we can define a deterministic policy $\hat{\pi}^*(s) = \arg\max_{a \in A} Q^*(s, a)$. Now

$$\begin{aligned} V^*(s) &= \sum_{s' \in S, a \in A} \pi^*(a \mid s) P(s' \mid s, a)(R(s, a, s') + \gamma V^*(s')) \\ &= \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V^*(s')) \\ &= \sum_{s' \in S} P(s' \mid s, \hat{\pi}^*(s))(R(s, \hat{\pi}^*(s), s') + \gamma V^*(s')) \end{aligned}$$

Doing the same by iteratively inserting the recursion for $V^*(s')$ in every step gives us

$$V^*(s) = \sum_{s' \in S} P(s' \mid s, \hat{\pi}^*(s))(R(s, \hat{\pi}^*(s), s') + \gamma V^{\hat{\pi}^*}(s')) = V^{\hat{\pi}^*(s)}$$

Thus $\hat{\pi}^*$ is optimal as well. □

So let $\hat{\pi}^*$ be the deterministic optimal policy, then if we define π_ϵ as the ϵ-greedy policy on the actions of $\hat{\pi}^*$, obviously $\lim_{\epsilon \to 0} \pi_\epsilon = \hat{\pi}^*$.

In the following sections we are going to discuss practical approaches for finding an optimal value/Q-function. Tabular methods such as dynamic programming, Monte Carlo methods and temporal differences (TD) can be efficient for problems where the state and action spaces are discrete and small enough. Then the value function V and action-value function Q can be represented by an array of size $|S|$ and $|S| \times |A|$ respectively.

For problems with large, possibly infinite, state/action spaces finding an accurate solution may be impossible. Thus function approximators like neural networks will be our approach.

It is to say that there are other methods such as genetic algorithms [23], genetic programming, simulated annealing that have been used to solve RL problems. They basically observe non learning agents with different policies and then choose the one that obtained the most reward. It can be seen as the evolutionary approach [24, p. 9]. However in this thesis, we will stick to methods that rely on estimating the value function.

4.2 Solving a Markov Decision Process

4.2.1 Dynamic Programming

Since the value function V^π and the policy π are in strong correspondence, naturally two different approaches arise: value and policy iteration. These two methods come from dynamic programming. They can be very efficient if the model of the environment is known and the problem is not too large. Also they are guaranteed to find the optimal policy in polynomial time [13].

Value iteration

Value iteration is based on (4.5) using that the Bellman optimality operator (see definition 4.1.9) is a contraction with fixed point V^*. Thus applying B iteratively

on a randomly initialised V_0^*

$$V_{n+1}^*(s) = (BV_n^*)(s) = \max_{a \in A} \left(\sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V_n^*(s')) \right),$$

converges to V^*. And in matrix notation

$$V_{n+1}^* = \max_j \left[(P(R + \gamma V_n^*))_{\cdot, j} \right].$$

An error estimate and terminating condition is given by the following theorem.

Theorem 4.2.1 (Value iteration error estimate)**.** *Using the metric from $(\mathcal{V}, d)$, e.g. $\|V_0 - V_1\| \coloneqq d(V_0, V_1)$ from theorem 4.1.2 or the euclidean metric ($\mathcal{V}$ is finite-dimensional) the following estimate hold.*

$$\forall \epsilon > 0 : \qquad \|V_{n+1}^* - V_n^*\| \leq \epsilon \frac{1 - \gamma}{\gamma} \qquad \Rightarrow \qquad \|V^* - V_{n+1}^*\| \leq \epsilon. \tag{4.8}$$

Proof. Straight forward using the properties of the contraction of the Bellman operator. Let $\epsilon > 0$ arbitrarily chosen and n large enough that $\|V_{n+1}^* - V_n^*\| \leq \epsilon \frac{1-\gamma}{\gamma}$ then

$$\begin{aligned} \|V^* - V_{n+1}^*\| = \|BV^* - BV_n^*\| &\leq \|BV^* - BV_{n+1}^*\| + \|BV_{n+1}^* - BV_n^*\| \\ &\leq \gamma \|V^* - V_{n+1}^*\| + \gamma \|V_{n+1}^* - V_n^*\| \\ \Leftrightarrow \qquad \|V^* - V_{n+1}^*\| &\leq \frac{\gamma}{1 - \gamma} \|V_{n+1}^* - V_n^*\| \leq \epsilon. \end{aligned}$$

□

Once the iteration terminates, i.e. $\|V_{n+1}^* - V_n^*\| \leq \epsilon \frac{1-\gamma}{\gamma}$, we can get our policy π^ϵ from V_{n+1}^* by doing one step of policy iteration. The policy π^ϵ is ϵ-optimal, which means the corresponding value function V^ϵ is ϵ-close to V^*: $\|V^* - V^\epsilon\| \leq \epsilon$.

Policy iteration

Till now we only talked about finding V^* and Q^* without practically talking about π^*, although the policy is essential for the agent. Policy iteration provides a way to compute π_{k+1} as an improvement of π_k iteratively from V^{π_k} starting with a policy π_0. The value function V^{π_k} again can be computed from π_k by solving a

linear system given by the Bellman equation 4.4.

$$\begin{aligned} V^{\pi_k}(s) &= \mathbb{E}\left[r_{t+1} + \gamma V^{\pi_k}(s_{t+1}) \mid s_t = s\right] \\ &= \sum_{s' \in S} P(s' \mid s)(r_{t+1} + \gamma V^{\pi_k}(s')) \\ &= \sum_{s' \in S, a \in A} P(s' \mid s, a)\pi_k(a \mid s)(R(s, a, s') + \gamma V^{\pi_k}(s')), \end{aligned}$$

since

$$P(s' \mid s) = \sum_{a \in A} P(s', a \mid s) = \sum_{a \in A} P(s' \mid a, s)P(a \mid s),$$

where $P(a \mid s)$ is just our policy $\pi_k(a \mid s)$ and $P(s' \mid a, s)$ the known transition probability of the MDP. Written in matrix notation with the according probability matrix Π_k and reward matrix R gives

$$V^{\pi_k} = \Pi_k R + \gamma \Pi_k V^{\pi_k} \tag{4.9}$$

$$\Leftrightarrow (I - \gamma \Pi_k) V^{\pi_k} = \Pi_k R. \tag{4.10}$$

This is solvable as the eigenvalues of $\gamma\Pi_k$ are in the interior of the unit disc (see Perron-Frobenius theorem).

Knowing V^{π_k}, one can derive a policy π_{k+1} that improves V^{π_k}, using the action-value function Q^{π_k} that is obtained from V^{π_k}

$$\begin{aligned} Q^{\pi_k}(s, a) &= \mathbb{E}\left[r_{t+1} + \gamma V^{\pi_k}(s_{t+1}) \mid s_t = s, a_t = a\right] \\ &= \sum_{s' \in S} P(s' \mid s, a)(R(s, a, s') + \gamma V^{\pi_k}(s')). \end{aligned}$$

Then we use the greedy policy update (theorem 4.1.1 with $\epsilon = 0$)

$$\pi_{k+1}(a \mid s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in A} Q^{\pi_k}(s, a) \\ 0 & \text{else} \end{cases}.$$

Theorem 4.2.2 (Convergence of policy iteration)**.** *The policy upgrade π_{k+1} improves V^{π_k}*

$$\pi_{k+1} \Rightarrow V^{\pi_k} \leq V^{\pi_{k+1}}$$

Proof. The policy update leads to

$$\begin{aligned} & V^{\pi_k}(s) = \sum_{a \in A} \pi_k(a \mid s) Q^{\pi_k}(s,a) \leq \sum_{a \in A} \pi_{k+1}(a \mid s) Q^{\pi_k}(s,a) \\ \Leftrightarrow \quad & \sum_{a \in A, s' \in S} \pi_k(a \mid s) P(s' \mid s,a)(R(s,a,s') + \gamma V^{\pi_k}(s')) \\ & \leq \sum_{a \in A, s' \in S} \pi_{k+1}(a \mid s) P(s' \mid s,a)(R(s,a,s') + \gamma V^{\pi_k}(s')) \end{aligned}$$

by construction of π_{k+1}, which reads in matrix notation

$$\begin{aligned} & V^{\pi_k} = \Pi_k R + \gamma \Pi_k V^{\pi_k} \leq \Pi_{k+1} R + \gamma \Pi_{k+1} V^{\pi_k} \\ \Leftrightarrow \quad & (I - \gamma \Pi_{k+1}) V^{\pi_k} \leq \Pi_{k+1} R \\ \Leftrightarrow \quad & V^{\pi_k} \leq (I - \gamma \Pi_{k+1})^{-1} \Pi_{k+1} R = V^{\pi_{k+1}}. \end{aligned}$$

□

4.2.2 Monte Carlo methods

In the previous chapter we required full knowledge of the environment's dynamic $P(s_{t+1} \mid s_t, a_t)$. This is a strong assumption that made value and policy iteration an efficient tool to find an optimal policy, but also less practicable in most cases.

From now on we resign the knowledge of $P(s_{t+1} \mid s_t, a_t)$ and start sampling trajectories τ. In addition we assume that these trajectories are finite, and call them episodes. In these sampling based methods one distinguishes between on-policy and off-policy. On-policy means that the episodes are sampled based on a policy π and the corresponding value function V^π is approximated. The new thing comes with off-policy (see section 4.2.2), where the approximated value function V^{π_ϵ} corresponds to a different policy π_ϵ, than the one π used for sampling. This gives the possibility to continue exploring the environment, while finding the optimal policy in parallel. It is trivial that off-policy is less prone to get stuck in local minima.

Another slight distinction that is made in the following prediction methods should be mentioned here beforehand. Namely first visit and every visit. Since it is possible to hit the same state s_t multiple times in the same episode τ it is to decide whether to use the cumulative rewards only on the first visit or on every visit of s_t for the estimation of $V^\pi(s_t)$. Although they are very similar, different theoretical properties arise.

The general idea of Monte Carlo methods appears in this context by generating independent samples from an unknown probability distribution, i.e. the environment's dynamic. By the law of large numbers, the estimated means, that are based on the samples, converge towards the real mean, that are determined by the unknown distribution.

Value prediction

Again we iteratively calculate our V^π for a policy π. After sampling a few episodes τ_i, $1 \leq i \leq n$ with π, one calculates the actual cumulative return for every state that occurs in any of these episodes. We assume that every episode either starts in an initial state s^*, or in a randomly chosen state s. Anyway, every τ_i should be independent.

$$\begin{aligned} \tau_i &= (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T) \\ G^{\tau_i}(s_t) &= \sum_{k=0}^{T-1-t} \gamma^k r_{t+k+1} \\ V_n^\pi(s_t) &= \frac{1}{n} \sum_{i=1}^{n} G^{\tau_i}(s_t) \end{aligned}$$

By the law of large numbers $V_n^\pi(s) \to V^\pi(s)$ converges. It is worth mentioning that the value estimates for each state s are independent. They don't rely on other state values as it is in dynamic programming, where bootstrapping is used. By the assumption that the episodes τ_i are independent, the error falls pointwise as

$$\mathrm{Var}\left[V_n^\pi(s)\right] = \frac{\mathrm{Var}\left[G^{\tau_i}(s)\right]}{n} \quad \forall s \in S.$$

The pointwise convergence of V_n^π doesn't give us any benefit, if the policy π isn't able to generate samples for every s. To assure global convergence, one has to stick to policies π with $\pi(a \mid s) > 0 \quad \forall a \in A, s \in S$. This also makes sense for the ϵ-greedy policy improvement theorem 4.1.1.

Value prediction gives a very basic algorithm to find the value function for a given policy π. In practice, one still has to do a policy improvement and that gets expensive if the model is not known, because one has to look one step ahead to choose the action with the best next state value. Then it's much more efficient to predict the action value itself.

Action value prediction

Of course the same can be done for the Q-function.

$$\begin{aligned} \tau_i &= (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T) \\ H^{\tau_i}(s_t, a_t) &= \sum_{k=0}^{T-1-t} \gamma^k r_{t+k+1} \\ Q_n^{\pi_k}(s_t, a_t) &= \frac{1}{n} \sum_{i=1}^{n} H^{\tau_i}(s_t, a_t) \end{aligned}$$

The convergence $Q_n^{\pi_k}(s_t, a_t) \to Q^{\pi_k}(s_t, a_t)$ holds as for the value prediction. Again, if $Q_n^{\pi_k}$ is known for n large enough, we can improve the policy by the ϵ-greedy policy π_ϵ from theorem 4.1.1.

Starting over again to find $Q^{\pi_{k+1}}$. The policy improvement is converging $\pi_{k+1} \geq \pi_k$ by theorem 4.1.1.

Off-Policy MC

Off-Policy Monte Carlo provides a method to obtain the value function for a policy π from episodes τ_i, that were generated by a different policy π'. It is only required that $\pi'(a \mid s) > 0$ holds, for every $s \in S$ and $a \in A$ where $\pi(a \mid s) > 0$. Let s be a state that is visited in the episodes τ_i, $G^{\tau_i}(s)$ the accumulated return in the episode τ_i following the policy π' from state s, $P_i(s)$ and $P_i'(s)$ the probability of that sequence τ_i to happen after s was visited by following the policy π and π' respectively, i.e.

$$P_i(s) = \prod_{\tau_i} \pi(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t).$$

Then, an estimator for V^π can be obtained by importance sampling

$$V_n^\pi(s) = \frac{\sum_{i=1}^n \frac{P_i(s)}{P_i'(s)} G^{\tau_i}(s)}{\sum_{i=1}^n \frac{P_i(s)}{P_i'(s)}}.$$

Again the benefit of Monte Carlo methods, that the environment's dynamics doesn't have to be known, arises from

$$\frac{P_i(s)}{P_i'(s)} = \frac{\prod_{\tau_i} \pi(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t)}{\prod_{\tau_i} \pi'(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t)} = \frac{\prod_{\tau_i} \pi(a_t \mid s_t)}{\prod_{\tau_i} \pi'(a_t \mid s_t)}.$$

4.2.3 Temporal Difference Learning

Short, TD methods combine DP and MC. Like MC methods they update the value function from sampled episodes without knowing the environment's dynamics and at the same time, they don't have to wait for the final outcome of the episode, they bootstrap like DP methods. This means that temporal difference method estimates the value of a state or action by using the values of other states or actions.

Assume we have a policy π and we want to find V^π. Again we start with a V_0^π and iteratively calculate V^π. This time, the update of V_k^π is based on the sampled error of the current estimator V_k^π with a step size α. But instead of updating the estimate of the value function after the end of each episode τ_i by

$$V_{n+1}^\pi(s_t) = V_n^\pi(s_t) + \alpha(G^{\tau_i}(s_t) - V_n^\pi(s_t)),$$

the simplest TD method updates the value function after each return r_t, based on the temporal tuple

$$\begin{aligned} \tau_i^t &= (s_t, a_t, r_{t+1}, s_{t+1}) \\ V_{n+1}^\pi(s_t) &= V_n^\pi(s_t) + \alpha(r_{t+1} + \gamma V_n^\pi(s_{t+1}) - V_n^\pi(s_t)). \end{aligned}$$

Here, the state values (also state-action values) of terminal states $V^\pi(s_T)$ are defined as zero for the update rule to make sense.

This method is called TD(0). The zero comes from the extension with eligibility traces, that will be described later, which makes this method a special case.

As already mentioned, TD methods combine the three main advantages from DP and MC: Bootstrapping, that the knowledge of the transition probabilities is not required and the usage of immediate updates after each return. Especially problems with very long or even infinite episodes make TD methods more applicable than MC. In addition, TD methods fortunately converge almost surely, as they can be seen as a linear stochastic approximation algorithm [26].

SARSA

The TD method can also be formulated for the action-value function. This method is called SARSA. The name SARSA comes from the information that is used in order to update the action-value function

$$\begin{aligned} \tau_i^t &= (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}) \\ Q_{n+1}^\pi(s_t, a_t) &= Q_n^\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_n^\pi(s_{t+1}, a_{t+1}) - Q_n^\pi(s_t, a_t)). \end{aligned}$$

Q-learning

It is important to mention that until now, the methods presented were all on-policy. Q-learning gives an off-policy method to estimate the value function of the greedy policy, while sampling with a corresponding soft policy (e.g. the ϵ-greedy policy).

$$\tau_i^t = (s_t, a_t, r_{t+1}, s_{t+1}) \tag{4.11}$$

$$Q_{n+1}^\pi(s_t, a_t) = Q_n^\pi(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a \in A} Q_n^\pi(s_{t+1}, a) - Q_n^\pi(s_t, a_t)). \tag{4.12}$$

The policy improvement is implicitly done, since $\pi = \arg\max_{a \in A} Q^\pi(s, a)$ already gives the new policy after every update. The off-policy property gives the advantage that all states and values can be explored by a soft policy, thus approximating the global optimum, while computing the optimal policy at the same time.

Actor-Critic

In the actor-critic approach, the process of finding the value function and the policy are separated. The value function acts as a critic that guides the actor (i.e. the policy). First a normal TD(0) step is done

$$\begin{aligned} \tau_i^t &= (s_t, a_t, r_{t+1}, s_{t+1}) \\ \delta_t &= r_{t+1} + \gamma V_n^\pi(s_{t+1}) - V_n^\pi(s_t) \\ V_{n+1}^\pi(s_t) &= V_n^\pi(s_t) + \alpha\delta_t, \end{aligned}$$

followed by the policy update

$$\begin{aligned} p_{n+1}(a, s) &= p_n(a, s) + \beta\delta_t \\ \pi_{n+1}(a \mid s) &= \frac{e^{p_{n+1}(a,s)}}{\sum_{a' \in A} e^{p_{n+1}(a',s)}}, \end{aligned}$$

with another step size parameter β. There are many other versions of actor-critic algorithms but the main idea stays the same. Actor-critic methods are able to find explicit stochastic policies that are useful in competitive and non-Markovian environments [22]. Another benefit is that they can be formulated to find policies in continuous action spaces.

Eligibility traces - λ-extensions

Eligibility traces are an extension for TD methods that aims for better convergence. It can be seen as an interpolation between the TD methods presented earlier and the Monte Carlo methods. Thus we shortly describe eligibility traces. Let's define the accumulated and bootstrapped n-step return $G_n^{\tau_i}(s_t)$ from state s_t in an episode τ_i

$$G_n^{\tau_i}(s_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V^\pi(s_{t+n}).$$

Then, the main idea is to use a weighted average

$$\hat{G}_\lambda^{\tau_i}(s_t) = (1 - \lambda) \sum_{n=1}^{T-t+1} \lambda^{n-1} G_n^{\tau_i}(s_t) + \lambda^{T-t+1} G^{\tau_i}(s_t)$$

of the accumulated returns $G_n^{\tau_i}(s_t)$ for the TD update. The TD(λ) update is defined as

$$\begin{aligned} \tau_i^t &= (s_t, a_t, r_{t+1}, s_{t+1}) \\ V_{n+1}^\pi(s_t) &= V_n^\pi(s_t) + \alpha(\hat{G}_\lambda^{\tau_i}(s_t) - V_n^\pi(s_t)). \end{aligned}$$

Now it is clear that for $\lambda = 0$ one obtains TD(0)

$$\hat{G}_0^{\tau_i}(s_t) = G_1^{\tau_i}(s_t) = r_{t+1} + \gamma V^\pi(s_{t+1})$$

and for $\lambda = 1$ the Monte Carlo update

$$\hat{G}_1^{\tau_i}(s_t) = G^{\tau_i}(s_t).$$

The problem that arises with this algorithm is that we lose the ability to make immediate updates. We have to wait until the end of an episode for the final return. So we lost the advantage of TD over Monte Carlo methods wich is discouraging at first glance. However this can be fixed by changing the view from forward to backwards by simply keeping track of the visited states. Before an update is performed, initialise a vector $e_0(s) = 0,\ \forall s \in S$ called the eligibility trace. Then, in each step reaching s_t, the error

$$\delta_t = r_{t+1} + \gamma V_n^\pi(s_{t+1}) - V_n^\pi(s_t)$$

is computed and e_t and V^π are updated for every state.

$$\begin{aligned} e_t(s) &= \begin{cases} \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \\ \gamma\lambda e_{t-1}(s) & \text{else} \end{cases} && \forall s \in S \\ V_{n+1}^\pi(s) &= V_n^\pi(s) + \alpha\delta_t e_t(s) && \forall s \in S \end{aligned}$$

Of course this extension can be done analogously for other TD methods such as SARSA and Q-learning.

4.2.4 Approximate Solution

Until now we always assumed that the state space S and the action space A are finit and small enough such that we can store the values of V^π and Q^π in a table where we could simply overwrite the values. Unfortunately this extensively limits the number of problems that are solvable with this methods. Let's say we have board of the size of a chessboard 8×8, where each cell can either hold a prey or be empty. If there is only one predator, the size of possible states is $8 \cdot 8 \cdot 2^{64} = 2^{70} \approx 1.18 \cdot 10^{21}$. Even if a state value would only take 1 byte to store, a total memory of one billion terabytes is necessary to store V^π. Another simple example, where tabular methods fail would be a continuous state space that consists of position and velocity. Therefore other methods are required that not only take much less memory, but also update the values in a local sense. By the huge size of states, it is rather unlikely that every state gets visited multiple times such that the Monte Carlo or TD methods apply. States that only differ slightly

won't be completely different in it's values. Therefore the value function can be modelled as a continuous function. This immediately reminds of methods that are used in supervised learning where a generalisation of the problem's solution is sought by using the information of few samples; function approximators.

Instead of updating the value function V_k^π, a parametrised function V_θ^π with parameter vector θ is used where θ gets changed such that V_θ^π approaches V^π. The number of parameters $\theta^{(i)}$ is usually far less than the number of possible states and a parameter update adapts multiple state value estimations simultaneously.

Value prediction

In general, in order to qualify parameter settings, it is necessary to define a measure, called cost or loss function J, that gives the distance from the approximate V_θ^π to the exact V^π. One example would be the mean squared error

$$MSE(\theta_k) = J(\theta_k) = \mathbb{E}\left[(V_{\theta_k}^\pi(s) - V^\pi(s))^2\right].$$

Then, by the gradient descent algorithm, the parameter update can be defined as

$$\theta_{k+1} = \theta_k - \alpha \nabla_{\theta_k} J(\theta_k),$$

with α such that

$$MSE(\theta_{k+1}) < MSE(\theta_k).$$

The mean squared error is motivated by the environment's dynamic. The parameter update focuses more on value errors of states, that are more likely to occur while sampling. In practice the real value of V^π is not known, though it can be replaced by any similar target. Most of them were already introduced in the previous section. Basically ever approximate target used for TD methods is appropriate.

The simplest and most common function approximators, besides neural networks (see chapter 5), are linear ones

$$V_\theta^\pi(s) = \sum_i \theta^{(i)} \phi_s^{(i)}.$$

Here, ϕ_s is the vector of state features, that contains feature information of the states. The choice and number of the state features is dependent on the problem and affects the performance of linear approximation significantly. Sometimes also feature combinations are necessary. For instance, let's have velocity and position as features. If a high velocity can either indicate a good state value but also a bad one, dependent on the position, an additional feature that represents their combination is necessary.

Finding an optimal parameter set $\theta^* = \arg\min_\theta MSE(\theta)$ is dependent on the complexity of the function approximator. Using linear function approximators

may make it easier to find θ^*. Still $V^\pi_{\theta^*}$ could be a bad approximation for V^π by the lack of complexity. Nonlinear function approximators like neural networks in contrast, are more flexible, but finding θ^* may be impossible. They rather tend to get stuck in local optima [24].

In the following we will take a look at some methods and how they are applied in combination with function approximators. They are mainly characterised by different loss functions on which the gradients are computed. For simplicity we stick to basic algorithms and omit the use of eligibility traces.

Q-learning

We already introduced Q-learning in the TD section and the main idea doesn't change. Rather than looking for V^π, the off-policy action-value function $Q^{\hat{\pi}}$ is computed. Starting with a randomly initialised parameter setting θ_0 and using

$$Q^{\hat{\pi}} \approx r_t + \gamma \max_a Q^{\hat{\pi}}_{\theta_k}(s_{t+1}, a)$$

as the approximation target. The cost function J, we want to minimise is then defined as

$$J(\theta_k) = \mathbb{E}\left[(r_t + \gamma \max_a Q^{\hat{\pi}}_{\theta_k}(s_{t+1}, a) - Q^{\hat{\pi}}_{\theta_k}(s_t, a_t))^2\right].$$

The expectation value is computed over a batch of uniformly chosen samples $\tau^t_i = (s_t, a_t, r_t, s_{t+1})$ that were once generated by a soft policy π_ϵ and kept in the so called experience replay memory. At the same time π_ϵ is taken from $\hat{\pi}$ which is

$$\hat{\pi}(s) = \arg\max_{a \in A} Q^{\hat{\pi}}_{\theta_k}(s, a),$$

to ensure exploration.

Q-learning is mostly applied when the action space is discrete and small enough. Then, it is useful to model $Q^{\hat{\pi}}$ as $S \to \mathbb{R}^{|A|}$, instead of $S \times A \to \mathbb{R}$. That means that a single forward pass of a state gives the Q values for every action at once. That makes it cheaper to get the targets and effective to get $\hat{\pi}(s)$.

Actor-Critic and the policy gradient theorem

This method in contrast to Q-learning can be applied to problems with huge action spaces or even a multidimensional continuous action space. As we've seen before, the actor-critic method demands two function approximators; one for the critic V^π and one for the actor π. This in turn demands two cost functions, one for each approximation. Let's recall the n-step discounted return $G^{\tau_i}_n$ that we used for eligibility traces

$$\begin{aligned} \tau^t_i &= (s_t, a_t, r_{t+1}, s_{t+1}, \ldots, r_{t+n}, s_{t+n}) \\ G^{\tau_i}_n(s_t) &= r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V^\pi(s_{t+n}). \end{aligned}$$

Then let J_V be defined as before

$$J^V(\theta^V) = \mathbb{E}\left[(G_n^{\tau_i}(s_t) - V_{\theta^V}^{\pi}(s_t))^2\right].$$

In addition we need a cost function for the update of θ^π. The goal of the policy improvement is to increase the estimated future rewards. Therefore let

$$J^\pi(\theta^\pi) = -\mathbb{E}\left[V^{\pi_\theta}(s_t)\right] = -\sum_{s\in S} P(s) \sum_{a\in A} \pi_{\theta^\pi}(a \mid s) Q^{\pi_\theta}(s,a)$$

be the cost function, that is to be minimised, i.e. maximise $\mathbb{E}\left[V^{\pi_\theta}(s_t)\right]$. Both are estimated on samples τ_i^t generated by π. We assume that $P(s)$ doesn't depend on π_θ like an initial start distribution (e.g. uniformly). Note that $V^{\pi_\theta} \neq V_{\theta^V}^{\pi}$, but by definition of $J^V(\theta^V)$ and $V^{\pi_\theta}(s_t) = \mathbb{E}\left[G_n^{\tau_i}(s_t)\right]$, we seek $V_{\theta^V}^{\pi} \to V^{\pi_\theta}$. Different from Q-learning, this method is on-policy. V^π corresponds to the behaviour policy π. At first glance it is not clear, how to compute the gradient $\nabla_{\theta^\pi} J^\pi(\theta^\pi)$ since V^{π_θ} explicitly (in π_{θ^π}) and implicitly (in Q^{π_θ}) depends on π_θ. Fortunately there exists an theorem that the derivative of Q^{π_θ} can be omitted.

Theorem 4.2.3 (Policy gradient theorem)**.** *Let π_θ be a parametrised policy with parameter θ and V^{π_θ} its value function. Then it holds*

$$\nabla_\theta V^{\pi_\theta}(s) \propto \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} Q^{\pi_\theta}(s',a) \nabla_\theta \pi_\theta(a \mid s'),$$

where $d^\pi(s,s')$ is a probability distribution for a fixed s.

Proof. First we expand V^{π_θ}.

$$V^{\pi_\theta}(s) = \sum_{a\in A} \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a)$$

Then the gradient gives

$$\nabla_\theta V^{\pi_\theta}(s) = \sum_{a\in A} \nabla_\theta \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a) + \pi_\theta(a \mid s) \nabla_\theta Q^{\pi_\theta}(s,a).$$

For simplicity we write $H(s) = \sum_{a \in A} Q^{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a \mid s)$. Now

$$\begin{aligned}
\nabla_\theta V^{\pi_\theta}(s) &= H(s) + \sum_{a \in A} \pi_\theta(a \mid s) \nabla_\theta Q^{\pi_\theta}(s,a) \\
&= H(s) + \sum_{a \in A} \pi_\theta(a \mid s) \nabla_\theta \sum_{s' \in S} P(s' \mid s,a) \left(R(s,a,s') + \gamma V^{\pi_\theta}(s') \right) \\
&= H(s) + \sum_{a \in A} \pi_\theta(a \mid s) \sum_{s' \in S} P(s' \mid s,a) \gamma \nabla_\theta V^{\pi_\theta}(s') \\
&= H(s) + \sum_{s' \in S} \left(\sum_{a \in A} \pi_\theta(a \mid s) P(s' \mid s,a) \right) \gamma \nabla_\theta V^{\pi_\theta}(s') \\
&= H(s) + \sum_{s' \in S} P_\pi(s,s',1) \gamma \nabla_\theta V^{\pi_\theta}(s'),
\end{aligned}$$

where $P_\pi(s,s',k)$ gives the probability to reach state s' from state s in k steps with $P_\pi(s,s',0) = \delta_{s,s'}$ We got an expression for $\nabla_\theta V^{\pi_\theta}$ in a recursive form. Iteratively expanding the recursion leads to

$$\begin{aligned}
\nabla_\theta V^{\pi_\theta}(s) &= H(s) + \sum_{s' \in S} P_\pi(s,s',1) \gamma \nabla_\theta V^{\pi_\theta}(s') \\
&= H(s) + \sum_{s' \in S} P_\pi(s,s',1) \gamma \left(H(s') + \sum_{s'' \in S} P_\pi(s',s'',1) \gamma \nabla_\theta V^{\pi_\theta}(s'') \right) \\
&= H(s) + \sum_{s' \in S} P_\pi(s,s',1) \gamma H(s') \\
&\quad + \gamma \sum_{s'' \in S} \left(\sum_{s' \in S} P_\pi(s,s',1) P_\pi(s',s'',1) \right) \gamma \nabla_\theta V^{\pi_\theta}(s'') \\
&= H(s) + \sum_{s' \in S} P_\pi(s,s',1) \gamma H(s') + \sum_{s'' \in S} P_\pi(s,s'',2) \gamma^2 \nabla_\theta V^{\pi_\theta}(s'') \\
&\vdots \\
&= \sum_{s' \in S} \sum_{k \geq 0} P_\pi(s,s',k) \gamma^k H(s')
\end{aligned}$$

If we write $\eta_\pi(s,s') = \sum_{k\geq 0} P_\pi(s,s',k)\gamma^k$ then

$$\begin{aligned}
\nabla_\theta V^{\pi_\theta}(s) &= \sum_{s'\in S} \eta_\pi(s,s') H(s') = \left(\sum_{s''\in S} \eta_\pi(s,s'')\right) \sum_{s'\in S} \frac{\eta_\pi(s,s')}{\sum_{s''\in S}\eta_\pi(s,s'')} H(s') \\
&= \left(\sum_{s''\in S} \eta_\pi(s,s'')\right) \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} Q^{\pi_\theta}(s',a) \nabla_\theta \pi_\theta(a \mid s') \\
&\propto \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} Q^{\pi_\theta}(s',a) \nabla_\theta \pi_\theta(a \mid s')
\end{aligned}$$

The proportion factor $\sum_{s''\in S}\eta_\pi(s,s'')$ gives the mean remaining episode length when state s is reached in a episodic MDP and is 1 for an infinite MDP [24, 25]. □

Most of the gradient based methods build on this theorem. For the actor-critic method we can use this theorem to get a practicable estimate for the gradient.

$$\begin{aligned}
J^\pi(\theta^\pi) &= -\mathbb{E}\left[V^\pi_{\theta^V}(s)\right] = -\sum_{s\in S} P(s) V^\pi_{\theta^V}(s) \\
\nabla_{\theta^\pi} J^\pi(\theta^\pi) &= -\sum_{s\in S} P(s) \nabla_\theta V^{\pi_\theta}(s) \\
&\propto -\sum_{s\in S} P(s) \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} Q^{\pi_\theta}(s',a) \nabla_\theta \pi_\theta(a \mid s') \\
&= -\sum_{s\in S} P(s) \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} Q^{\pi_\theta}(s',a) \pi_\theta(a \mid s') \nabla_\theta \log\left(\pi_\theta(a \mid s')\right) \\
&= -\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a \mid s)\right) Q^{\pi_\theta}(s,a)\right]
\end{aligned}$$

In addition, the policy gradient theorem allows adding a so called baseline. Let $b(s)$ be any function or random variable that is independent of actions a, then

$$\begin{aligned}
-\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a \mid s)\right) b(s)\right] &= -\sum_{s\in S} P(s) \sum_{s'\in S} d^\pi(s,s') \sum_{a\in A} b(s) \nabla_\theta \pi_\theta(a \mid s') \\
&= -\sum_{s\in S} P(s) \sum_{s'\in S} d^\pi(s,s') b(s) \nabla_\theta \left(\sum_{a\in A} \pi_\theta(a \mid s')\right) \\
&= -\sum_{s\in S} P(s) \sum_{s'\in S} d^\pi(s,s') b(s) \nabla_\theta (1) \\
&= 0
\end{aligned}$$

This gives the possibility to choose a baseline $b(s)$ that leaves the expected value unchanged

$$\begin{aligned}\nabla_{\theta^\pi} J^\pi(\theta^\pi) &= -\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a \mid s)\right) Q^{\pi_\theta}(s,a)\right] \\ &= -\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a \mid s)\right) \left(Q^{\pi_\theta}(s,a) - b(s)\right)\right],\end{aligned}$$

but reduces its variance. A popular choice for $b(s)$ is the value function $V^{\pi_\theta}(s)$ such that the advantage function $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ replaces Q^{π_θ}

$$\nabla_{\theta^\pi} J^\pi(\theta^\pi) = -\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a \mid s)\right) A^{\pi_\theta}(s,a)\right].$$

To solve this optimisation problem we start with some random initialised parameters θ^V and θ^π. Sample trajectories τ_i of given length n and estimate the gradients

$$\begin{aligned}\nabla_{\theta^V} J^V(\theta^V) &= \nabla_{\theta^V} \mathbb{E}\left[(G_n^{\tau_i}(s_t) - V_{\theta^V}^\pi(s_t))^2\right] \\ &\approx -\frac{1}{n} \sum_{s_t \in \tau_i} (G_n^{\tau_i}(s_t) - V_{\theta^V}^\pi(s_t)) \nabla_{\theta^V} V_{\theta^V}^\pi(s_t) \\ \nabla_{\theta^\pi} J^\pi(\theta^\pi) &= -\mathbb{E}\left[\nabla_\theta \log\left(\pi_\theta(a_t \mid s_t)\right) A^{\pi_\theta}(s_t,a_t)\right] \\ &\approx -\frac{1}{n} \sum_{s_t, a_t \in \tau_i} \nabla_{\theta^\pi} \log\left(\pi_\theta(a_t \mid s_t)\right) (G_n^{\tau_i}(s_t) - V_{\theta^V}^\pi(s_t)).\end{aligned}$$

Now, the parameters θ^V and θ^π can be updated by these estimated gradients, using gradient decent methods that were presented in section 5.2.1.

5. Artificial Neural Networks[1]

An artificial neural network is a branch of machine learning, that is deployed in research of artificial intelligence. Whereas other AI paradigms apply to problems which are hard for humans to solve, but sill have a mathematical model that describes the problem, neural networks usually apply to problems where it is hard to find a generalising model, such as speech and handwriting recognition, language translation or even car driving.

The first concepts about neurons and neural networks came up in the 1940s. Inspired by the human brain, researchers created hypothesis of learning and tried to build algorithms that where able to compute (e.g. Turing's B-type machines, Perceptrons). In the beginning things looked promising until it was shown that single layer perceptrons are unable to reproduce the XOR function which demonstrated the limits in computation. In addition, the second big issue was the lack of computational power to process larger networks. Though, in 1975 the back-propagation algorithm solved the problem regarding fast training of multi-layer networks. Especially since processing power (using GPUs and distributed computing) and digitalisation are growing exponentially (see Moore's Law), the use of neural networks have recently increased tremendously.

Figure 5.1 shows a typical topology of a neural network. It basically consists of an input layer, an arbitrary but fixed number of hidden layers and an output layer. Each layer has some neurons which have connections to every neuron in the subsequent layer.

As shown in figure 5.2 every connection has a weight and every neuron has an activation function and a bias such that the output of a neuron can be computed by applying the activation function on the sum over the weighted inputs of the neurons of the previous layer plus the bias.

$$\text{output} = \sigma(w \cdot x + b)$$

[1]This chapter was written earlier in 2017 in the course of a seminar in scientific computing, supervised by Prof. Dr. Clemens Heitzinger. For the sake of completeness of the thesis and profound understanding of neural networks, as far as possible, I decided to include this chapter.

A biological Inspiration for Computation

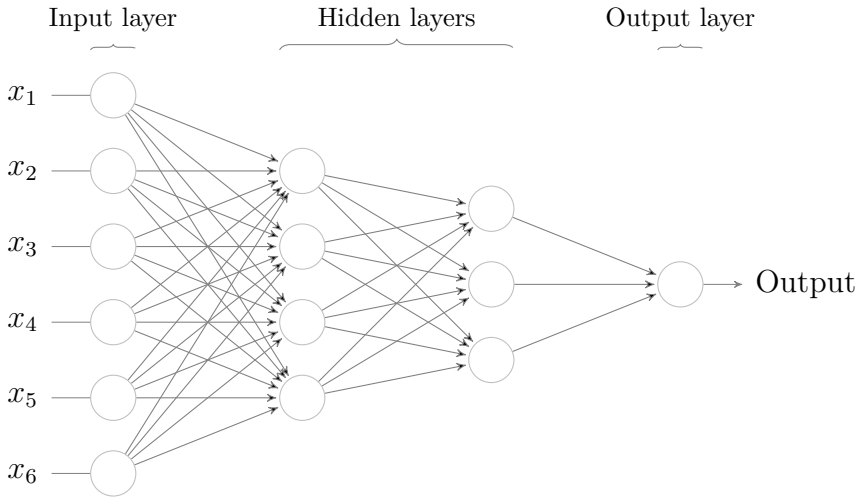

Figure 5.1: Topology of a neural network

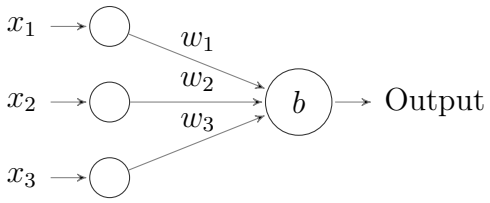

Figure 5.2: Single neuron

5.1 Perceptrons and Universality

5.1.1 Perceptron

The activation function of the first neurons like perceptrons had the Heaviside step function as activation function

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0. \end{cases}$$

Perceptrons are based on binary input and produce binary output as well. Furthermore they can be used to compute logical operations especially the `NAND`-Gate (Figure 5.3). These `NAND`-Gates are universal for computation which means that they can compute any logical function, if properly arranged.

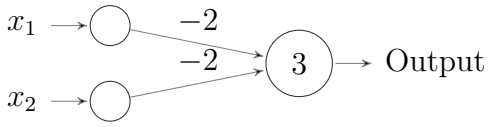

Figure 5.3: NAND - Gate

The property of neural networks being a universal function approximator is also called Turing complete. This is a great property, since it ensures that theoretically any function can be approximated by a sufficiently large network, that only consists of perceptrons. For example we can now use perceptron networks to classify data, but there is still a big problem that remains. Let's say we have some data we want to classify. Our network already performs well on some data A but still we want to do better. So we try to change weights and biases in order to increase the number of correctly classified data while keeping A correctly classified. This won't be possible without losing control over A because of the discontinuous activation function of a perceptron. Correctly calculated output from before could now flip in uncontrollable manner. The solution for this would be the use of continuous activation functions. However, this may cause the loss of universality, since these continuous perceptrons don't necessarily have to build `NAND`-Gates anymore. Fortunately we don't lose Turing completeness if our smooth activation functions are chosen accordingly, as shown in section 5.1.3.

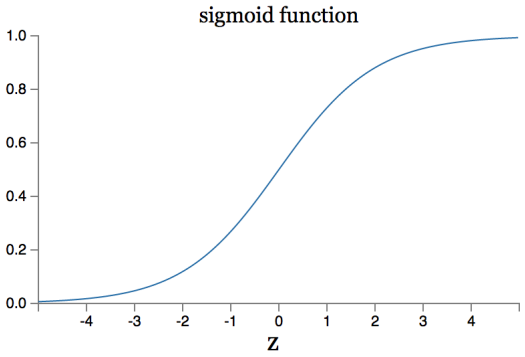

Figure 5.5: Sigmoid activation function $\sigma(Z) = \frac{1}{1+e^{-Z}}$

5.1.2 Smooth activation functions

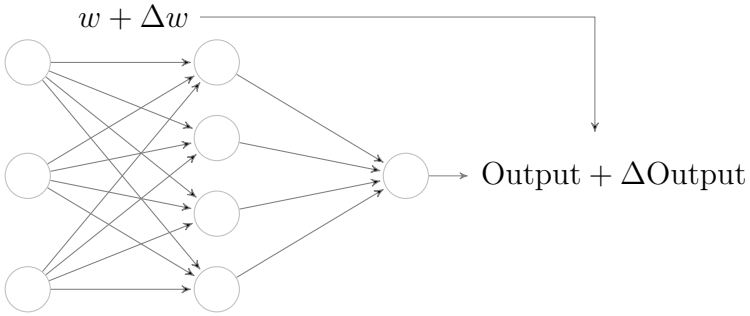

Figure 5.4: Continuous feed forward

In order to control the behaviour of the networks better, we want our network to result in small changes in the output when only small changes are made on some parameters (figure 5.4). This can be achieved by using smooth activation functions such as the sigmoid function (figure 5.5):

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Of course there are many others such as tanh, softmax, each one having its own advantages. Sigmoid function for example has a very simple form in its derivative.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

However how can we know, that we are still able to approximate any function when using this activation function? In 1989 Cybenko showed that single hidden

layer networks with sigmoidal (smoothed Heaviside) activation functions can approximate any continuous mapping on a compact subset of $\mathbb{R}^n$ [4]. In the same year Hornik et al. showed that multilayer feedforward networks are universal approximators [8]. These proofs are quite technical so I will just sketch the proof why single layer networks are universal.

5.1.3 Universality

Assume we have a continuous mapping $f : I_n \coloneqq [0,1]^n \mapsto [0,1]$ and a network with n inputs, one hidden layer with N neurons and one output. We want our network to approximate f:

$$\sigma\left(\sum_{i=1}^{N} w_{i,out}\ \sigma\left(w_{in,i}\cdot x + b_i\right) + b_{out}\right) \approx f(x), \tag{5.1}$$

with $w_{in,i}\cdot x = \sum_{j=1}^{n} w_{j,i}x_j$. Since $\sigma : \mathbb{R} \mapsto [0,1]$ is a homoeomorphism we can rewrite (5.1):

$$G(x) \coloneqq \sum_{i=1}^{N} w_{i,out}\ \sigma(w_{in,i}\cdot x + b_i) \approx \sigma^{-1}(f(x)) - b_{out}, \tag{5.2}$$

with $G(x) \in C(I_n)$. We need to show that finite sums of the form like $G(x)$ are dense in $C(I_n)$.

Proof. Let $S \subseteq C(I_n)$ be the set of functions of the form $G(x)$. Clearly S is a linear subspace of $C(I_n)$. Assume that the closure of S is not all of $C(I_n)$, then $R \coloneqq \overline{S}$ is a closed proper subspace of $C(I_n)$. By Hahn-Banach Theorem there is a bounded linear functional $F \neq 0$ on $C(I_n)$ with $F(R) = F(S) = 0$. By Riesz Representation Theorem, F is of the form

$$F(G) = \int_{I_n} \sum_{i=1}^{N} w_{i,out}\ \sigma(w_{in,i}\cdot x + b_i)\ d\mu(x) = 0$$
$$\Rightarrow F(G) = \int_{I_n} \sigma(w\cdot x + b)\ d\mu(x) = 0\ \forall w, b$$

Now one can show that $\mu \equiv 0$ which leads to a contradiction to $F \neq 0$. This follows by extending F on L^∞, defining a sequence of sigmoid functions that converges to translated Heaviside functions H_θ where still $F(H_\theta) = 0$, furthermore by linearity for all simple functions which are dense in L^∞. □

Similar $G(x)$ could be rewritten as a simple function by sharpening σ to the Heaviside function with steps s_i at $-\frac{b_i}{w_{in,i}}$ with element wise division.

$$G(x) = \sum_{i=1}^{N} w_{i,out}\, \sigma(w_{in,i} \cdot x + b_i) \approx \sum_{i=1}^{\frac{N}{2}} h_i \cdot \chi_{[s_i^1, s_i^2]} \approx \sigma^{-1}(f(x)) - b_{out},$$

$$\begin{aligned} \text{when} \quad h_i &= w_{2i-1,out} = -w_{2i,out} \\ s_i^1 &= -\frac{b_{2i-1}}{w_{in,2i-1}} \\ s_i^2 &= -\frac{b_{2i}}{w_{in,2i}}. \end{aligned}$$

Thus $G(x)$ can approximate simple functions and they can approximate any continuous mapping on a compact subset. So we obtain that our network with only one hidden layer can approximate any continuous mapping on a compact subset, when activation functions are used which can approximate the Heaviside function. Of course this class of smooth activation functions is not the only one to make neural networks universal. This property was also shown with other activation functions such as the rectified linear unit ($\mathrm{ReLU}(x) = \max(0, x)$), however the proof is different [17].

Using only smooth activation functions leads to a smooth neural network as a whole, in the sense of a function approximator. Now we want to make use of the property that our network became robust. This gives us the ability to make small changes on parameters such as weights and biases in order to improve the network. The algorithm how parameters need to be changed to increase the number of correct outputs is called backpropagation.

5.2 Backpropagation, Problems and Improvements

5.2.1 Backpropagation and Gradient Descent

Backpropagation was first introduced in 1960. About 20 years later people started to apply it to neural networks, which worked far faster than earlier approaches to learning. It's based on an optimisation problem using gradient descent in order to solve it.

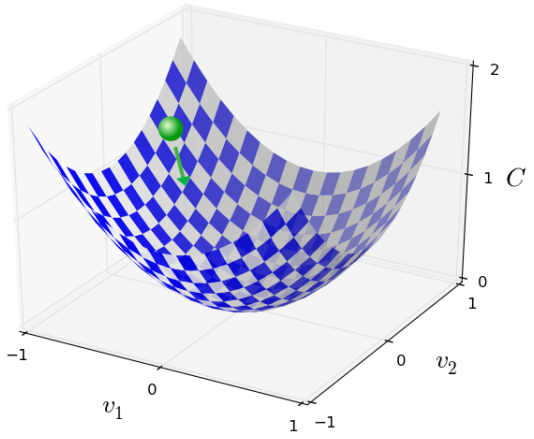

Figure 5.6: Visualisation of gradient decent

Gradient Descent

We want our network $f_\theta^*(x)$ to learn a given mapping $f(x)$ based on a finite set D of data $(x, f(x))$. Rather than increasing the number of samples where $f_\theta^*(x) = f(x)$ we decrease the error of a predefined error function (cost function). Minimising that cost function is done with gradient descent. For example using the mean squared error as cost function

$$C(\theta) = \frac{1}{2n} \sum_x \|f(x) - f_\theta^*(x)\|^2 \to min,$$

gradient decent is applied simply by updating the parameters

$$\theta' = \theta - \eta \nabla_\theta C(\theta),$$

with so called learning rate η giving the amount of update. Iteratively computing the gradient $\nabla_\theta C(\theta)$ with respect to the parameters and then moving to the opposite direction, can be visualised as walking down the slope of a valley (figure 5.6).

This update, seen as the learning, is slow when training set is large since it takes the sum over all samples. Therefore an adaptation by estimating the gradient on randomly drawn samples leads to the method called min-batch stochastic gradient decent.

Improved Methods for GD

The Mini-batch Stochastic Gradient Descent reduces costs for computing the gradient drastically. Set a number of epochs N and a mini-batch size n.

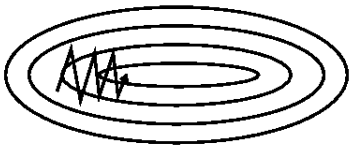

Figure 5.7: SGD without momentum

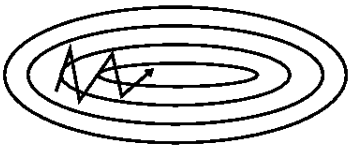

Figure 5.8: SGD with momentum

For every epoch shuffle all samples and update parameters for mini-batch of n randomly chosen training examples.

$$\theta' = \theta - \eta \nabla_\theta C(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$
$$y^{(i:i+n)} = f(x^{(i:i+n)}),$$

where $x^{(i:i+n)}$ denotes the n dimensional subvector $(x_i, x_{i+1}, ..., x_{n-1})$ of the vector x. This gives a very efficient approximation of the gradient, but still there are a lot of ways to improve learning. For example all parameters in each epoch use the same learning rate η, which is apparently not reasonable. Some parameters could already be near to the optimal value, while others still need bigger adjustments, or maybe we want to have a bigger learning rate in the beginning but throughout time, when the error got decreased, we want to take smaller steps to prevent jumping around the minimum. There is also the fact that SGD likely gets trapped in local minima for highly non-convex error functions and basically oscillates across the slopes, which slows down optimisation. There are methods like adding momentum to the gradient to reduce these oscillations.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta C(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$
$$\theta' = \theta - v_t$$

Momentum dampens oscillation and helps SGD to accelerate into the relevant direction (figure 5.7 and 5.8). This is done by adding a fraction γ of previous update vector v_{t-1} to the current update vector [20].

There are a lot of improved methods like Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop and Adam [20]. In figure 5.9 and 5.10 convergence of these methods are compared. Nesterov approximates the next step and thus corrects the momentum step for better convergence. Adagrad adapts the learning rate for each parameter based on the past gradients. Adadelta and RMSprop are extended versions of Adagrad also based on adaptive leaning rate. Adam (Adaptive Moment Estimation) also computes adaptive learning rates for each parameter, but in addition to storing exponentially decaying average of past squared gradients, it also keeps an exponentially decaying average of past gradients, similar to momentum. In each update they divide the learning rate by the average of past

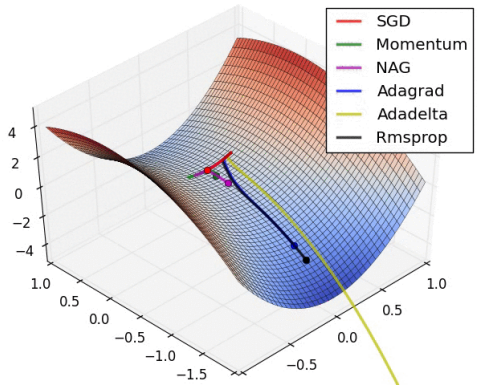

Figure 5.9: Method comparison on saddle point problem [20]

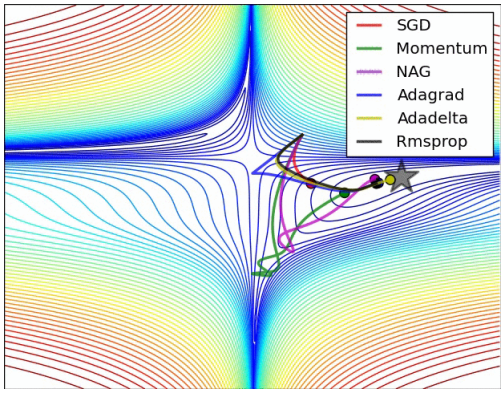

Figure 5.10: Method comparison on loss surface contours [20]

squared gradients and use the average of gradients for gradient update. According to [20] Adam might be the best overall choice, while Adadelta and RMSprop have similar good convergence.

Backpropagation

We now know a lot of possible optimisation methods, but we still don't know how to compute the gradients. The backpropagation algorithm provides an efficient method for computing gradients for each parameter. First we need to make some definitions for simplicity reasons.

Let C be an arbitrary cost function, σ an arbitrary activation function. Let w^l_{jk} denote the weight from node j of layer $l-1$ to node k in layer l. $W^l \in \mathbb{R}^{|l-1|\times|l|}$, where $|l|$ denotes the size of the l-th layer, be the matrix of the weights from layer $l-1$ to layer l. Let b^l_j be the bias in j-th node of l-th layer and b^l the vector of the biases of l-th layer. Then the input vector of the l-th layer can be written as $z^l = (W^l)^T\, a^{l-1} + b^l$. With $a^l = \sigma(z^l)$ as the output vector of the l-th layer, applying the activation function σ element wise. Finally we define the error vector of layer l as $\delta^l = \nabla_{z^l} C$.

Now we can compute the derivatives of the cost function with respect to each parameter by applying the chain rule, such that we obtain four fundamental equations of the backpropagation algorithm:

For the last layer L we get

$$\delta^L = \nabla_{z^L} C = \nabla_{a^L} C \odot \nabla_{z^L} \odot a^L = \nabla_{a^L} C \odot \nabla_{z^L} \odot \sigma(z^L) = \nabla_{a^L} C \odot \sigma'(z^L),$$

where $\odot$ denotes the Hadamard product (element wise multiplication).

For any other layer l we get

$$\begin{aligned}\delta^l &= \nabla_{z^l} C = (\nabla_{z^l} \cdot (z^{l+1})^T)\nabla_{z^{l+1}} C = (\nabla_{z^l} \cdot ((a^l)^T W^{l+1} + (b^{l+1})^T)\delta^{l+1} \\ &= (\nabla_{z^l} \cdot \sigma(z^l)^T)\, W^{l+1}\delta^{l+1} = \operatorname{diag}(\sigma'(z^l))(W^{l+1}\delta^{l+1}) = (W^{l+1}\delta^{l+1}) \odot \sigma'(z^l).\end{aligned}$$

Note that after applying the chain rule we get the matrix $\nabla_{z^l} \cdot (z^{l+1})^T$ since each input in layer $l+1$ is dependent on every input of the previous layer l.

Now we get simple expressions for the derivatives:

$$\begin{aligned}\frac{\partial C}{\partial b^l_j} &= \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j \\ \frac{\partial C}{\partial w^l_{jk}} &= \frac{\partial C}{\partial z^l_k}\frac{\partial z^l_k}{\partial w^l_{jk}} = \delta^l_k a^{l-1}_j\end{aligned}$$

Depending on the GD-Method, parameter updates are performed averaging the gradient over a batch of samples x.

$$\begin{aligned}b^l_j &\to b^l_j - \frac{\eta}{N}\sum_x \delta^l_j(x) \\ w^l_{jk} &\to w^l_{jk} - \frac{\eta}{N}\sum_x (\delta^l_k a^{l-1}_j)(x)\end{aligned}$$

We see that the error vector for each layer can be computed recursively beginning with the last layer. So while doing the feedforward for a sample, all input vectors z^l and output vectors a^l are stored for every layer l. When arrived at the last layer L we can start backpropagating the error in the network for the sample x [17].

5.2.2 Problems and Improvements

Vanishing and exploding gradient

A big problem that comes up with backpropagation is the vanishing or exploding gradient. For each error vector δ^l of layer l you get an additional factor of the derivative of the activation function. So the deeper the network the higher the exponents of the derivatives.

$$\begin{aligned}\delta^L &= \nabla_{a^L} C \odot \sigma'(z^L) \\ \delta^l &= (W^{l+1}\delta^{l+1}) \odot \sigma'(z^l)\end{aligned}$$

Now if derivatives are big, the gradient can explode, if derivatives are small the gradient can vanish. The exploding gradient problem can be prevent by just clipping the gradient if above threshold and is easier to detect. The vanishing

gradient problem however needs more sophisticated solutions and is also much harder to detect, because slow learning can have many reasons (close to local minimum, overfitting). One thing that can help preventing the vanishing gradient is to use special cost functions such as the cross entropy

$$C = -\frac{1}{n}\sum_x \sum_j [y_j \ln(a_j^L) + (1-y_j)\ln(1-a_j^L)], \quad \text{with } a_j^L = \sigma(z_j^L)$$
$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j).$$

The derivatives of the activation function $\sigma'(z_j^L)$ got cancelled out for the derivatives of the cost function with respect to the parameters in the last layer.

Another thing that helps is using different activation functions. For example the ReLU (rectified linear unit)

$$\sigma(z) = \max(0, z).$$

This activation function is quite different from the one we saw before but still, it can be shown, that they can compute any function [17]. It is piecewise linear such that its derivative is piecewise constant and nonzero on one part. To understand why this helps preventing a vanishing gradient, let's recall that the derivative of the sigmoid function approaches zero on both parts, i.e. $\lim_{|z|\to\infty} \sigma'(z) = 0$. Especially a saturated sigmoid function, i.e. weights $w \gg 1$, meaning that it is close to the heaviside step function, results in a zero derivative for even small inputs $|z| > \epsilon$. The derivative of the ReLU, in contrast, is always nonzero for $z > 0$. Now while backpropagating the error through the network, the gradients stop attenuating. For further details see [6]. In addition to that, these neurons can also help to prevent overfitting, since their zero part results in a sparse network, which simulates dropout layers. More details are given in the next section.

Overfitting

When training a neural network we actually want it to solve a general problem, like recognising and classifying all handwritten digits. But instead, we only have a finite number of noisy samples, where we can train our network on. Now if the network starts to even fit the noise on the data, such that it significantly performs better on the training data than on independent general data, this is called overfitting. Especially models with too many free parameters are very prone to overfitting. This makes it essential to estimate the optimal number of neurons in a neural network, called it's capacity, to a given problem.

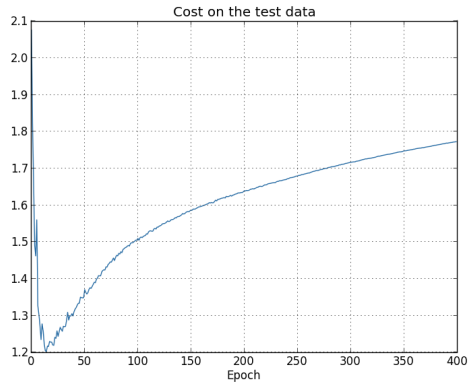

Figure 5.11: Cost function on test data [17]

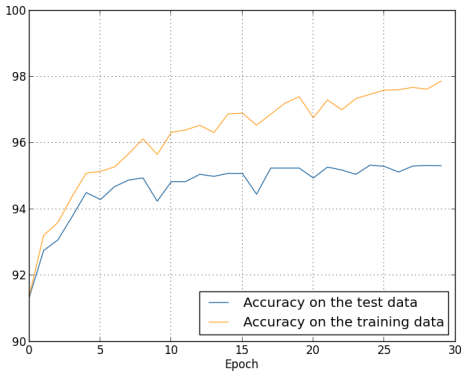

Figure 5.12: Comparison of accuracy of test data and training data [17]

Before starting the training, the set of samples is split into training data, test data and validation data. The test data is used as an independent set of samples to monitor the networks learning progress. It can be used to detect overfitting. The use of the validation data is described in the next section. Detecting overfitting can be done by either evaluating the cost function or calculating the accuracy of the trained network on the test data and compare those values to one on the training data. While the cost function on the training data is continuously minimised during the training, overfitting is indicated when the cost function on the test data stops decreasing (figure 5.11). On the other hand, overfitting can be detected if the accuracy of the network on the test data saturates on low level, while the accuracy on the training data is still increasing (figure 5.12). An ad-hoc method, called early stopping, to avoid overfitting is to stop the training as soon as it is detected. However this is not a satisfying solution, since we want our network to exploit the whole training data until a generalising solution is found. Another cutting method would be to reduce the size of the network. As mentioned before, the less parameters a model has, the less prone it is to overfitting. Nonetheless large networks have the potential to be more powerful than small networks.

Obviously the more training data available the better the networks ability for generalisation, unfortunately in most cases it is hard to gain more samples. Though, sometimes they can be generated by slightly transforming the available data in a natural way.

A very common and more sophisticated method is regularisation. The idea is to punish large weights when minimising the cost function. Small weights are somehow regularising the network. A smooth activation function, which approximates step functions, gets sharpened if the input has large weights. This makes

the network more rough. L^2 or L^1 regularisation is done by adding a penalty term (the L^2 or L^1 norm of the weights) to cost function $C(\theta)$ and using $\widetilde{C}(\theta)$ as the new cost function.

$$\widetilde{C}(\theta) = C(\theta) + \frac{\lambda}{pn} \sum_{i,j,k} \|w^i_{j,k}\|_{L^p},$$

where small values of λ prefer minimise the original cost function $C(\theta)$ and large values prefer small weights.

Another very common regularisation method is using dropout layers. It is radically different from L^2, L^1 regularisation. Rather than modifying the cost function, the network itself gets the ability to randomly and temporarily switch off some hidden neurons, then doing the feed forward before backpropagating the error. Thus, in each backpropagation, only a part of the network is trained from a mini-batch [17]. An heuristic idea why this might help is to see the dropout as training different networks in each step. Training different networks with mini-batch SGD and comparing their output gives more confidence about the possible correct output, since only a minority of the networks should produce an incorrect output.

Parameter Initialisation and Optimisation

When creating a neural network, parameters such as weights and biases have to be set. A first attempt would be a random initialisation with gaussian normal distribution with mean zero and a variance of one, but if we assume $w_{j,k}, b_i \sim \mathcal{N}(0,1)$ it follows that the input of each neuron $z_k = \sum_{j=1}^{n} w_{j,k}a_j + b_k$ has normal distribution with a variance of $n+1$. This leads to the problem of slow learning in the beginning, because, assuming an activation functions with flat ends, the derivative is close to zero at large input values. Thus this initialisation can be improved by setting the parameters $w_{j,k}, b_i \sim \mathcal{N}(0, \frac{1}{n+1})$.

Optimisation of hyper parameters such as the structure of the network, learning rate η, regularisation parameter λ and so on can be estimated on the validation data. The reason why they should be estimated on an independent set of samples is simply to hinder overfitting these parameters on the training data [17].

5.3 Convolutional Neural Networks

A convolutional neural network (CNN) is a special type of neural network, that is designed to preserve local information. Again, these networks were inspired by biological processes like the receptive field. Especially in problems where audio,

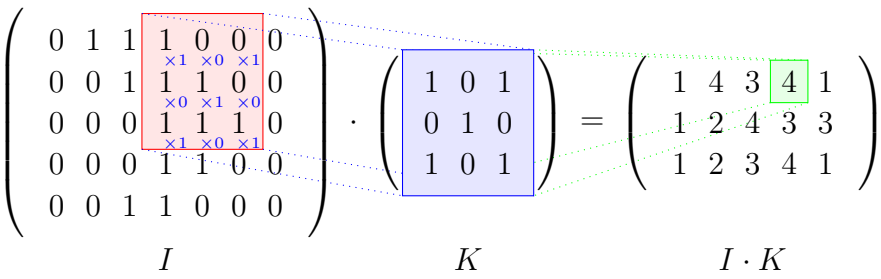

Figure 5.13: Example for a 2D convolution

images or even 3D data is processed, CNNs are extensively applied. In this section however, we will stick to CNNs that are used for image processing.

Typically CNNs consist of some convolutional layers followed by ordinary fully connected layers. The fully connected layers were already described in the previous sections. A convolutional layer consists of feature maps that are computed by a kernel. This kernel itself is a two dimensional matrix that holds the weights of the layer. The kernel is applied on the input or previous feature maps as a local convolution. Assume we have a 2D Matrix of size 5×7 as input, e.g. a small grey scaled image, then the convolution with a kernel of size 3×3 is computed (see figure 5.13). The fact that the same weights are used in the kernel in each step for this convolution is called parameter sharing. It would be possible to use different weights in the kernel for each convolution step, but that would blow up the number of parameters. After the convolution a bias is added and an activation function is applied as usual.

Additionally to the kernel, it is necessary to define the step size, called the stride, for the convolutional step. It gives the amount by which the filter is shifted in each step. Together with the kernel size, the stride determines the size of each resulting feature map.

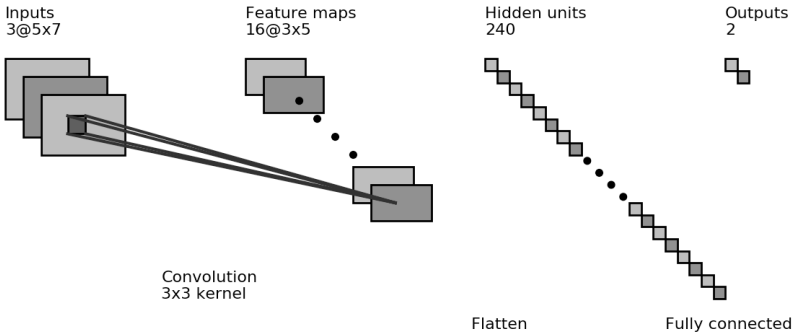

Figure 5.14: Example of a $16@3 \times 5$ convolutional layer with a 3×3 kernel applied on the inputs

Figure 5.14 shows a convolutional layer with 16 feature maps and a 3×3 kernel resulting in feature maps of size 3×5, if a stride of 1 is applied on the inputs of size 5×7.

In figure 5.14 an image with 3 channels is processed by the convolutional layer. Then the resulting feature maps are flattened into a one dimensional array which is taken as the input for the subsequent fully connected layers.

A convolutional layer can be summarised as proposed by [10]:

- Accepts a volume of size $H_1 \times W_1 \times C_1$
- Requires four hyperparameters:
  - Number of feature maps N
  - Size of the kernel $K \times K$
  - Stride S
- Produces a volume of size $H_2 \times W_2 \times C_2$
  - $H_2 = \frac{H_1 - K}{S} + 1$
  - $W_2 = \frac{W_1 - K}{S} + 1$
  - $C_2 = N$
- This gives $K \cdot K \cdot C_1$ parameters for each feature map and $(K \cdot K \cdot C_1) \cdot N$ parameters for the whole layer plus N parameters for the bias.
- Each feature map of size $H_2 \times W_2$ is the result of a convolution with stride S. The kernel of size $K \cdot K \cdot C_1$ is multiplied by a kernel-sized window of the input that has total size $H_1 \times W_1 \times C_1$.

6. Many Agent Reinforcement Learning in Agent Based Modelling

In the previous chapter we discussed some techniques how MDPs for a single agent can be solved. However, in ABM we have typically many agents that may be divided into groups of similar behaving agents. This chapter will combine the idea of reinforcement learning with agent based modelling and present a method how this can be accomplished in the example of a predator prey model.

In ABM one can model a single agent and its evolution as a partially observable Markov decision process (POMDP) [28]. POMDPs are a generalisation of ordinary MDPs, where the agent has to determine its action without having all informations about its state. This lack of information is modelled by a probability distribution over the state space. Though, we will only deal with a natural deterministic partial information of the world's state, that is observed by the agent, so we can use the same concepts presented in chapter 4.

For the classical approach the agent would act after a fixed policy, that was previously defined by the user, to generate trajectories. In this new approach we exploit the natural design of ABMs as POMDPs and introduce a reward function that is more or less equivalent to the classical policy definition, but with some fundamental advantages. When ABMs are used to imitate a complex emergent group behaviour, it often ends in an infinite trial and error loop where one tries to find the right agent's policy or parameters that lead to the desired group behaviour. This can be avoided in the RL approach by defining positive rewards one the desired outcomes, such that in the optimal case, the RL algorithm finds by design the wanted policy. Another problem, that is closely related to the previous one, is the user's bias, which affects the quality of the model in particular while defining the policies. Of course the reward function is biased as well but since it has a simpler structure than a policy the total model bias can be reduced.

Additionally we make use of the group structure in ABM. That means that agents with similar behaviour don't have to explore the state space individually.

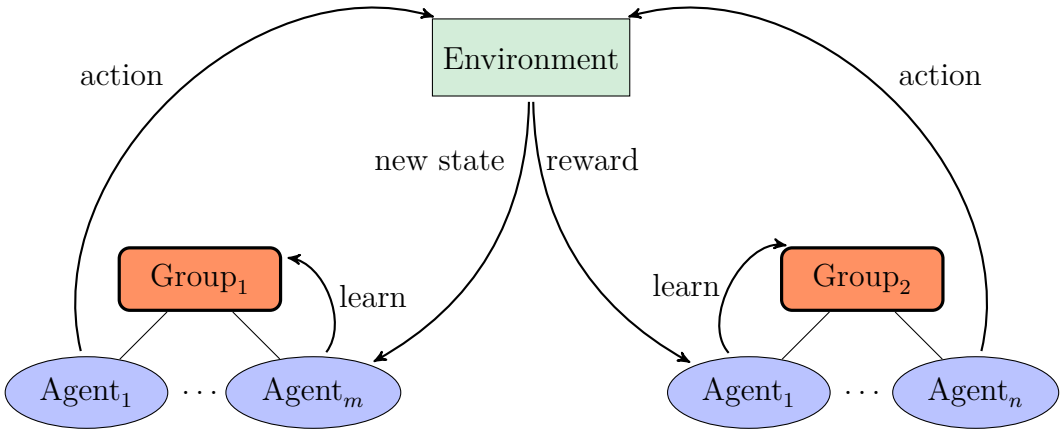

Figure 6.1: Environment with State-Action-Reward-State setting.

They share their experience in order to learn faster.

6.1 Reinforcement Learning in Agent Based Modelling

The first step, for a combination of agent based modelling and reinforcement learning, is to formulate the structure of ABM as a MDP, such that we can apply the technique of reinforcement learning.

As described in chapter 3, ABM has two principle components: the environment and the agents. Usually a MDP is formulated for a single agent. Now we have multiple agents that live in a common environment. Each agent in ABM naturally forms a MDP (see 'Case Studies for a Markov Chain Approach to Analyze Agent-Based Models' [11], where a Markov chain can be extended to a Markov decision process by considering actions). Therefore it is necessary to specify the state and the action space first. These spaces generally depend on the problem that needs to be modelled. In a classical ABM, where the agents are driven by rules, the specification of the action space is generated on the fly, containing all possible actions that a agent can take. The state space, for example, can contain possible environment features, i.e. position, velocity, set of neighbouring agents, or can be image like, representing the local view of an agent extracted from the environment. This gets more concrete in the next section. Another substantial part of a MDP is the reward function, but we will get to that later in more detail.

After the formal definition of the agents as MDPs, the environment comes into play. The environment takes the performed actions in each step and handles the

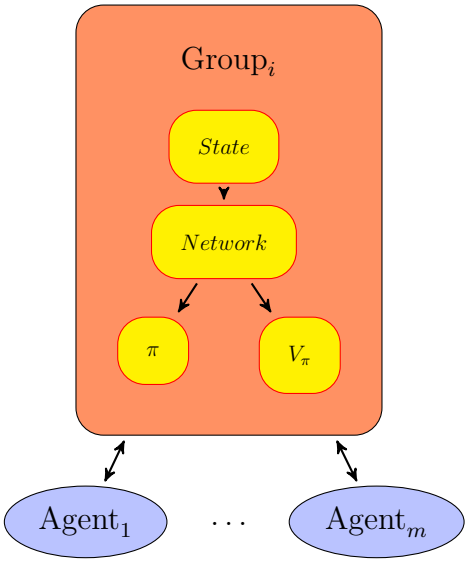

Figure 6.2: Unfold group control.

individual state transitions. It serves as a connection between these MDPs, since a agent's state is dependent on other agent's actions.

In this setting, assuming we have the reward functions, it is possible to solve all MDPs given by the agents. Since this can be an expensive task, we will exploit the agent's group structure for reasons of simplicity. In agent based modelling one usually deals with only few different groups of agents, where the agent of a group are characterised by a similar behaviour. We can use this logic to significantly reduce the number of MDPs that need to be solved. Instead of seeking the optimal policy for each agent individually, we only solve one MDP for each group. As a result we only have to formulate one reward function for each group. Figure 6.1 shows the general framework used for this RL-ABM approach.

The reward function should be independent of the agent's action and rather be defined on states only. Otherwise the agent would simply learn to favour rewarded actions and one would miss the purpose of using reinforcement learning in ABM. It would be just a long way round for the classical rule based approach. So in order to properly apply RL, it is necessary to define states of success and failure that correspond to the wanted behaviour for each group and its agents. Thereby finding the optimal actions, depending on the current state, so as to reach the states of success and avoid the states of failure. Generally speaking one has to define a real valued measure on the state space, the so called reward function, that gives the quality of success for a given state.

If the model was defined in this context, a pre-simulation, called the learning process, is needed. The learning process is a controlled simulation, in terms of

stability, that is run before the actual simulation can start. Most models require adjustments for a successful learning process (see section 6.2 for details).

For the learning process one has to choose a suitable reinforcement learning algorithm that finds the optimal policy. In theory any of chapter 4 could be used, depending on the complexity of the model, environment or reward function. State transition probabilities for the agents could be calculated if environmental states are determined by the agents action (i.e. policy) only, such that dynamic programming could be used. Else, if the state or action space is large, approximate solutions are in favour. For this thesis, an actor critic algorithm with neural network as function approximator is used, as it gives an efficient and highly flexible tool. Figure 6.2 shows the group control obtained by the reinforcement learning paradigm.

The following pseudo code shows the general algorithm for this framework.

```
// Initialise the environment and
// Actor Critic Neural Network for each group
environment.init()
for group in agents:
  ActorCritic[group] = Init_ActorCritic_Network(parameters)

// Start the learning process
for ep in range(MAX_EP):
  // Clear environment (e.g. delete dead agents)
  environment.clear()
  // Get agents' states for each group to select actions
  // and reset all buffers
  for group in agents:
    state[group] = environment.get_states_of(group)
    buffer[group] = {'s': [], 'a': [], 'r': [], 'd': []}

  // Create batch-like data (Action - State - Reward)
  // for the learning (backpropagation)
  for k in range(batchSize):
    for group in agents:
      actions = ActorCritic[group].choose_actions(state[group])
      new_states, rewards, done = environment.step(group, actions)
      state[group] = new_states
      buffer[group].append({'s': new_states, 'a': actions,
                            'r': rewards, 'd': done})
  // Learn from batch-like data
```

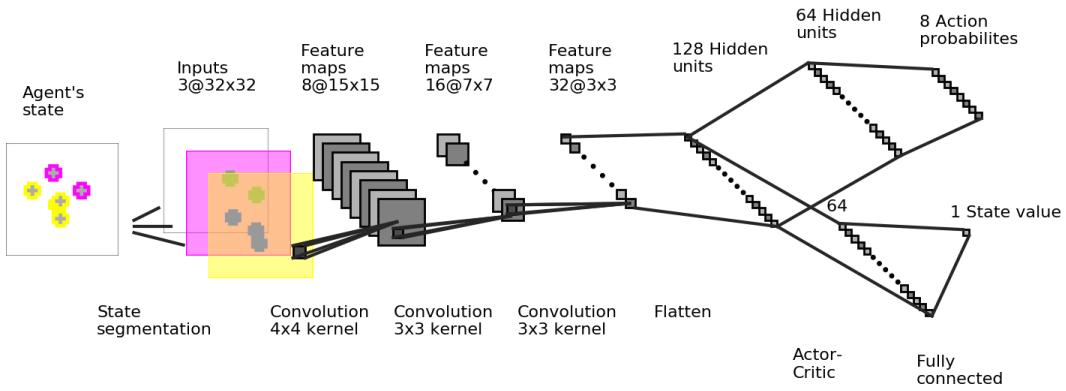

Figure 6.3: Actor Critic Network for group control

```
for group in agents:
  buffer[group]['r'] = getDiscountedRewards(buffer[group]['r'])
  ActorCritic[group].update(buffer[group])
```

In each step of the simulation, agents take their actions sequentially in groups. Based on the current states, that are fed into the corresponding network, the probabilities for each action (actor) and the value of these states (critic) are returned. In case of a discrete action space, as it is shown in figure 6.3, a multinomial distribution is used. When dealing with n-dimensional continuous action spaces, the actor has to return the mean and standard deviation for each dimension of the action space. This gives a set of normal distributions where actions are drawn in each dimension at the same time.

Their new states, drawn actions and rewards are collected groupwise in buffers. After a certain time, determined by the batch size, the experience τ_i is used to estimate the state values by $G_n^{\tau_i}(s_t)$ (discounted rewards). The update method calculates the gradients of the loss functions

$$\nabla_{\theta^V} J^V(\theta^V) \approx \frac{1}{n} \sum_{s_t \in \tau_i} (V_{\theta^V}^{\pi}(s_t) - G_n^{\tau_i}(s_t)) \nabla_{\theta^V} V_{\theta^V}^{\pi}(s_t) \tag{6.1}$$

$$\nabla_{\theta^\pi} J^\pi(\theta^\pi) \approx \frac{1}{n} \sum_{s_t, a_t \in \tau_i} (G_n^{\tau_i}(s_t) - V_{\theta^V}^{\pi}(s_t)) \nabla_{\theta^\pi} \log\left(\pi_\theta(a_t \mid s_t)\right) \tag{6.2}$$

and updates the parameters of the neural network accordingly. To encourage

exploration for the policy, an entropy term

$$H(\pi_\theta(a \mid s)) = -\mathbb{E}\left[\log\left(\pi_\theta(a_t \mid s_t)\right)\right]$$
$$\nabla_{\theta^\pi} H(\pi_\theta(a_t \mid s_t)) \approx -\frac{1}{n} \sum_{s_t, a_t \in \tau_i} \left(1 + \log\left(\pi_\theta(a_t \mid s_t)\right)\right) \nabla_{\theta^\pi} \pi_\theta(a_t \mid s_t)$$

is added as proposed in [15]. Therefore, the discounted rewards $G_n^{\tau_i}(s_t)$, the network's state value $V_{\theta^V}^\pi(s_t)$ and the action probability $\pi_\theta(a_t \mid s_t)$ for the corresponding taken action a are needed. $V_{\theta^V}^\pi(s_t)$ and $\pi_\theta(a_t \mid s_t)$ are obtained by feeding the network with the states s_t from the buffer.

Figure 6.3 shows the network used for this purpose. It consists of a convolutional part, that is shared between the actor and the critic, and some fully connected layers.

The states are given as an image from the local view of an agent. It is segmented into different channels, one for each group and one for the environment (obstacles, geometry information). Therefore convolutional layers are used, since they preserve local informations in images (see section 5.3 and [10]).

The parameters of the total network are updated by the combined gradients

$$\delta = \nabla_\theta J^\pi(\pi_\theta) + \alpha \nabla_\theta J^V(V_\theta) + \beta \nabla_\theta H(\pi_\theta) \tag{6.3}$$

with weighting parameters α and β. Parameter α is used to balance the importance of $\nabla_\theta J^\pi(\pi_\theta)$ and $\nabla_\theta J^V(V_\theta)$ for the parameter update and β gives the amount of exploration.

As an application and validation for this framework, a predator prey agent based model using reinforcement learning is implemented.

6.2 Reinforcement Learning on Predator Prey

Besides the well known Lotka-Volterra equations, it is possible to model a predator prey ecosystem with ABM. A basic predator prey ABM consists of only two groups. The preys that seek to survive and the predators that have to hunt. Preys can't starve but get hunted down, predators can starve but aren't hunted.

The environment for the model is a simple toroidal continuous $[0, 128] \times [0, 128]$ space. For visualisation purposes, the continuous space is projected onto a 512×512 pixel grid.

The initial number of preys is set to `prey.init_n = 80` and for the predators it is `predators.init_n = 30`. The agents are represented by a circle of radius 2. Groups are distinguished by individual colours. In this example, preys are coloured yellow and predators are magenta.

The state space is a $32 \times 32 \times 3$ discrete space. Therefore a local detail of size $[0, 32] \times [0, 32]$ is extracted from the environment at the agents position. Instead of using colours in the state space, groups are segmented into separate monochrome channels as in figure 6.3, resulting in $32 \times 32 \times 2$ for the groups and $32 \times 32 \times 1$ for the environment.

Both, the predators and the preys have the same discrete action space of size 8. Agents only navigate through the environment taking one of these 8 directions in each step, as showed in figure 6.4. The only difference between these two groups is the velocity. Preys are designed to be 10% faster than predators, to make the problem more interesting.

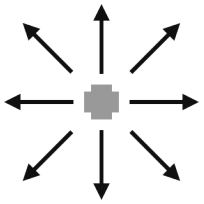

Figure 6.4: Action space of an agent

In order for the predators to attack, they only have to get close enough to a prey. This action is taken automatically, as soon as they overlap with each other, instead of adding it to the action space.

In the classical approach, behavioural rules are defined. The simplest rules would be that the predators seek for their closest prey and head for its direction. The prey in turn, tries to escape as soon as a predator approaches.

However, for this new approach, these rules should be found by the agents themselves with the help of a reward function. The reward function for the prey and the predators is defined as follows.

```
// Reward function for preys
def reward(prey):
  // get all predators that are within the agent's state
  near_predators = env.get_predators(prey.pos, prey.view_radius)
  reward = 0, overlapping = False

  // if prey is too close to a predator, give negative rewards
  // and decrease life points
  for pred in near_predators:
    if prey.overlapping(pred):
        reward -= 1.0
        prey.lp -= 5

  // if prey is dead, punish harder
  if prey.lp <= 0: reward -= 5
  return reward
```

```
// Reward function for predators
def reward(predator):
  // get all preys that are within the agent's state
  near_preys = env.get_preys(predator.pos, predator.view_radius)
  reward = 0, overlapping = False

  // if a prey is in attack range, give positive rewards,
  // else decrease the agent's life points
  for prey in near_preys:
    if predator.overlapping(prey):
      reward += 1.0
      predator.lp = max(predator.max_lp, predator.lp+3)
      overlapping = True
      break
  if not overlapping: predator.lp -= 1

  // if predator is starved, punish harder
  if predator.lp <= 0: reward -= 5
  return reward
```

Here, `prey.view_radius = predator.view_radius = 16`, is the maximum distance at which an opponent agent is recognised, as indicated previously in the size of the states.

Furthermore, preys have initial life points of `prey.lp = 20`, predators have `predator.lp = 90`. If agents die, i.e. life points less then or equal to 0, they are handled by `environment.clear()` et the end of this episode. There are two different approaches for handling dead agents. Whether they respawn again somewhere in the environment, or they just get deleted. In this example, both approaches were implemented. To switch between these two scenarios, the environment holds a property `environment.static = True`, that can be changed to `False`. A static environment means that groups keep their initial number of agents throughout the simulation, i.e. agents get respawned. Thus agents can't die off. Otherwise if `environment.static = False`, they get deleted. In this case it is necessary to implement an ability for the agents to breed, to prevent them from dying off too fast. So as long as they are alive, they have a counter `self.breed_counter` that gets decreased by one after each step. Each group has a different breed interval `prey.breed_interval = 90` and `predator.breed_interval = 130`. If the counter becomes zero for a specific agent, a new agent of the same group is created and placed at the parent agent's position.

For the simulation process, we set `environment.static = False`, since the

population dynamic is the purpose of this model. Though, this switch is needed for the learning process. As indicated in the previous section, the model sometimes needs to be adjusted for a successful learning.

Imagine we would set `environment.static = False` and start with the policy iteration, the population of a specific group could decrease dramatically or even die off. On the other hand a population could explode. Let's recall that every agent contributes their trajectories to the experience of the whole group. This happens when the buffer gets filled with the new states, rewards etc. in each step. If a population explodes, then so the buffer does and backpropagating all the experience from the buffer takes too long. While at the same time, the other group is typically close to extinction. Thus the buffer is kept small for this group and therefore the learning is slow. If a group dies off, the learning terminates. That is even for the other group, since they learn from interactions. This comes from the definition of the reward function. The reward of an agent only changes significantly on interactions with the other group. A stable learning process requires a stable simulation.

Development of the Learning Process

Before implementing the actor-critic method with neural networks, a simpler approach has been made. Q-learning in combination with a feature based linear function approximator (LFA) was used as it is proposed in [19]. Therefore basically the same parameter setting was applied as for the actor-critic model in the next section. The parameters that had to be changed are explained here.

A linear approximated Q-value function is written as

$$Q^{\hat{\pi}}_{\theta_k}(s_t, a_t) = \theta_k \cdot \psi(s_t, a_t),$$

where θ_k are the weights and $\psi(s_t, a_t)$ is the feature vector of the state-action pair (s_t, a_t). The feature vector is discussed in detail later. For now, as we want to approximate the optimal Q-value function, that fulfils the recursion

$$Q^{\hat{\pi}}(s_t, a_t) \approx r_t + \gamma \max_a Q^{\hat{\pi}}(s_{t+1}, a),$$

the loss function and the gradients become

$$\begin{aligned} J(\theta_k) &= \frac{1}{2}(r_t + \gamma \max_a Q^{\hat{\pi}}(s_{t+1}, a) - \theta_k \cdot \psi(s_t, a_t))^2 \\ \nabla_{\theta_k} J(\theta_k) &= -(r_t + \gamma \max_a Q^{\hat{\pi}}_{\theta_k}(s_{t+1}, a) - \theta_k \cdot \psi(s_t, a_t))\psi(s_t, a_t). \end{aligned}$$

Note that we use our approximative $Q^{\hat{\pi}}_{\theta_k}$ as an estimator for $Q^{\hat{\pi}}$ after the gradients were applied.

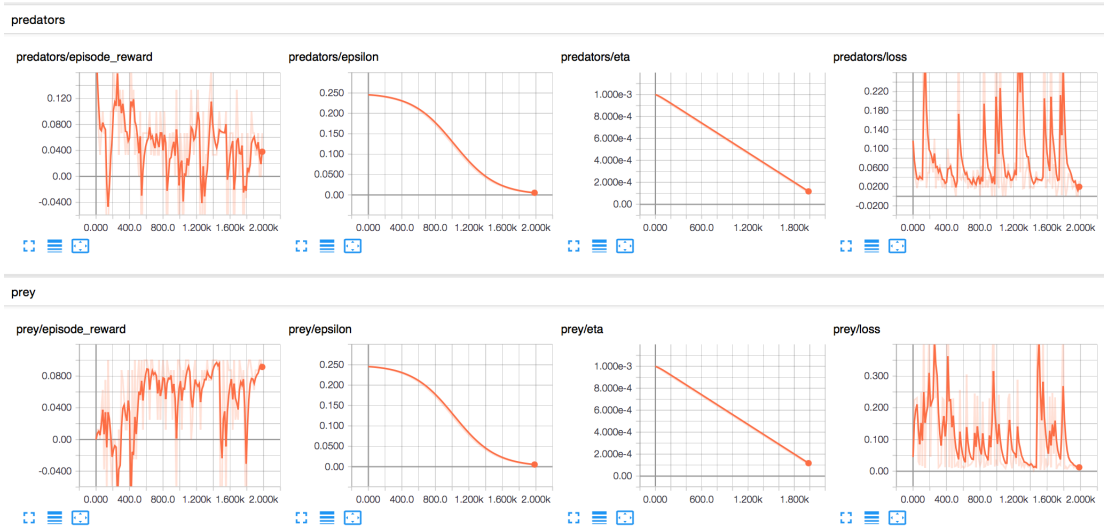

Figure 6.5: Monitoring learning progress for LFA using Tensorboard

Then the total parameter update becomes

$$\theta_{k+1} = \theta_k + \eta(r_t + \gamma \max_a Q^{\hat{\pi}}_{\theta_k}(s_{t+1}, a) - \theta_k \cdot \psi(s_t, a_t))\psi(s_t, a_t),$$

with the learning rate $\eta = 0.001$ and decay $\gamma = 0.95$ set as for the actor-critic method. The sign changes since we use gradient decent.

Q-learning is an off-policy method. To encourage exploration, a soft ϵ-greedy policy π_ϵ is used, where $\epsilon > 0$ is reduced over time. See figure 6.5 for the learning statistics.

The feature vector $\psi(s_t, a_t)$ is designed to provide enough information to the agent about the effect of taking action a_t while being in state s_t. It should be possible to evaluate the consequences of the action. The need to make choices about the design of state-action features hides a major weakness of linear function approximators, as we will see later.

For the predator prey model, the feature vector $\psi(s_t, a_t)$ has to provide informations about the environment that evolves if the agent takes action a_t, i.e. move in a specific direction, starting from his current position. Therefore it was necessary to discretise the state space that was described previously.

The 32×32 visibility field of the agents was divided into two grids of size 3×3, one for each group. These two grids also form the feature vector. Hence the grid was converted to a vector, where each entry contains the number of agents, that are located in that cell. After that, the vector gets normalised by the total number of agents in the grid. Now the feature vector is the concatenation of these two grid vectors. This means that we had 18 features in total.

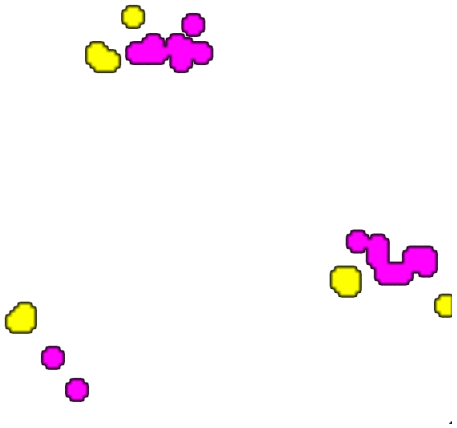

Figure 6.6: Scene of a predator prey simulation with Q-learning and LFA.

The problem that arose was that the grid was too coarse. This became a problem when the predator arrived in a cell, where a prey was located. The features of the 3×3 grid didn't give enough information to the predator about the exact location of the prey, that would have been needed to attack it. Thus, the grid had to be refined into a 5×5 sized grid.

As mentioned in [19], if the number of features increases the learning gets hindered when a linear function approximator is used. In our case, the number of features increases exponentially with the size of the grid.

Another problem that these features have is that they can't give a correct prediction about the effect of taking action a_t in state s_t. This comes naturally with the stochastic dynamic of the system. These features can't provide information about other agents behaviour, that would also be needed.

Despite these problems and after fixing the grid size, the agents successfully learned a predator prey behaviour by means of classical agent based modelling rules. After 2000 time steps, the prey had a mean life time of 125 time steps. The predators survived 219 time steps on average.

Interestingly the agents tend to gather in groups in only small areas. This can be observed in figure 6.6. All 30 predators and 80 preys are gathered on this scene. The agents on the one hand learned, that they are more successful in groups, but on the other hand they are not capable of evaluating the importance between grouping and hunting at the same time. This happens due to the simplicity of linear function approximators. More details about the weakness of feature vectors and linear function approximators can be read in [5, p. 414].

Nevertheless, this thesis aimed for a more flexible but complex reinforcement

learning framework that would even allow continuous action spaces and unbiased state representation (the definition of features can introduce additional bias).

Learning using Actor-Critic

Up to this point, a lot of debugging and parameter calibration was required for a successful learning process.

Things like proper array representation, segmentation of the states and input normalisation for the network turned out to be quite buggy. As well as finding optimal hyperparameters such as the learning rate and the right structure and size of the network.

However the following set of hyperparameters finally led to success. For discounting the reward, a decay factor of $\gamma = 0.95$ was used. The weights in the loss function 6.1 were set to $\alpha = 0.5$ and $\beta = 0.01$.

The neural network, showed in figure 6.3, is used for this model. It has $60,289$ parameters in total. A more detailed break down of the number of parameters in the layers is given below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   Connected to
=================================================================
Input (InputLayer)           (None, 32, 32, 3)         0
_________________________________________________________________
Conv2D-1 (Conv2D)            (None, 15, 15, 8)         392       Input[0][0]
_________________________________________________________________
Conv2D-2 (Conv2D)            (None, 7, 7, 16)          1168      Conv2D-1[0][0]
_________________________________________________________________
Conv2D-3 (Conv2D)            (None, 3, 3, 32)          4640      Conv2D-2[0][0]
_________________________________________________________________
Flatten (Flatten)            (None, 288)               0         Conv2D-3[0][0]
_________________________________________________________________
Dense-Flatten (Dense)        (None, 128)               36992     Flatten[0][0]
_________________________________________________________________
Policy-Dense (Dense)         (None, 64)                8256      Dense-Flatten[0][0]
_________________________________________________________________
Value-Dense (Dense)          (None, 64)                8256      Dense-Flatten[0][0]
_________________________________________________________________
Policy-Output (Dense)        (None, 8)                 520       Policy-Dense[0][0]
_________________________________________________________________
Value-Output (Dense)         (None, 1)                 65        Value-Dense[0][0]
=================================================================
Total params: 60,289
Trainable params: 60,289
Non-trainable params: 0
_________________________________________________________________
```

We used an Adam optimiser with default initial parameters to adjust the weights as proposed in chapter 5 and [20]. Both, predator and prey, have the same learning rate $\eta = 0.001$. The networks were trained for 500 episodes, each episode consists of a batch of 16 steps for each agent. It took 14 minutes to train both networks on a dual-core CPU $2,9$ GHz Intel Core i5 processor and 16GB ram with Tensorflow 1.7.0 built from source with SSE and AVX support without GPU.

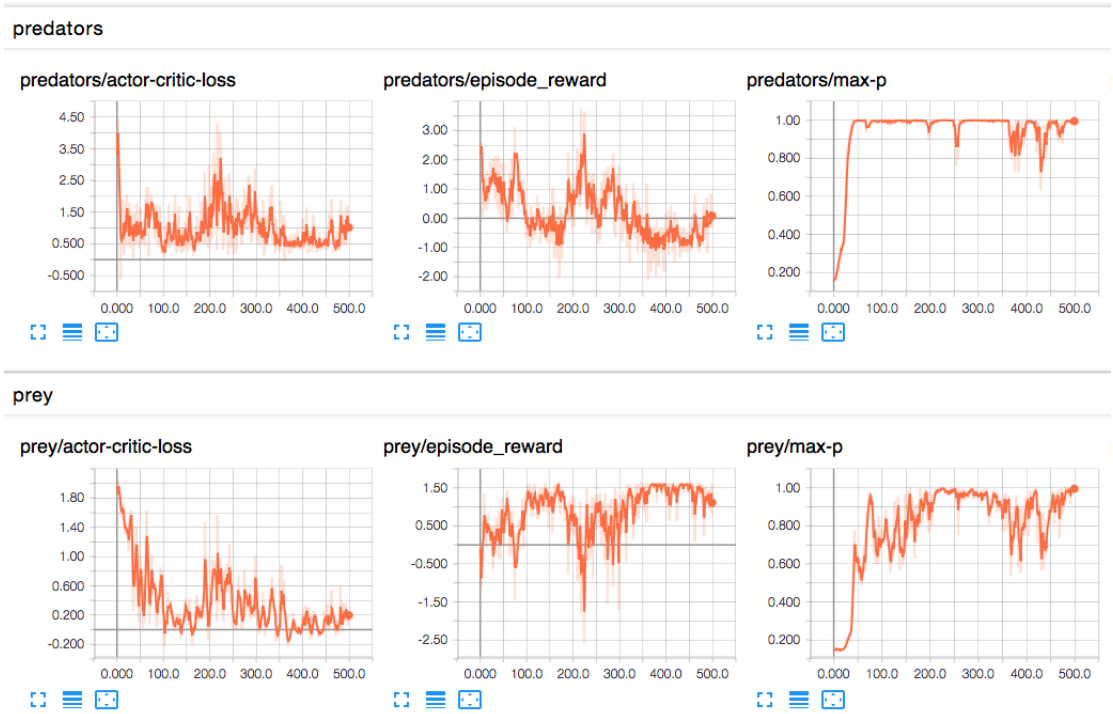

Figure 6.7: Monitoring learning progress with Tensorboard

Tensorboard is used for monitoring the training. Parameters such as total loss, mean episode reward and maximum probabilities provide enough information to check the learning progress and convergence. Figure 6.7 shows a typical training with these parameters.

The maximum probabilities, that are plotted for each batch, indicate the policy convergence. The more the maximum probability for the actions approaches one, the less randomly an agent behaves. Although predators and prey have the same learning rate, the preys' policy converges slower and less confident. A reason for this could be that preys are only forced to learn if they got attacked by predators, i.e. when predators already improved their policy. Generally speaking, the rivalry of groups adds a perturbation that avoids local minima such that the algorithm tends to converge towards global minima. Interestingly, in the first half of the training, before the mean reward of the predators reaches a peak, the maximum probability gets shortly reduced. That is observed at episode 60, 200 and 250.

Also it is distinctive that the mean episode rewards of predators and the preys are opposing. This comes naturally with the rivalry of the two groups.

During this pre-calibration loop, also problems with the model itself arose. As mentioned earlier the model parameter is set to `environment.static = True` for the training, such that dead agents respawn at a random new position in the environment at the end of each episode. Thereby the predators became lazy. Instead of exploring the environment for some preys, they didn't move until a previously killed prey respawned in their direct surrounding. Only now they started to pursue the preys. To encourage the predator's exploration, I restricted the area, in which they can respawn. More precisely, they respawn randomly in a 5×5-sized area around their previous position where they died.

Simulation and results

After a successful training, the environment parameter `environment.static` is switched to `False`, leading to a classical predator prey agent based model. The simulation is run over 1000 episodes. Figure 6.8 shows the population changing over 1000 episodes, starting with 30 predators and 80 preys with the same properties and parameters as in 6.2. The mean life time of the predators and the preys were approximately 165 and 180 time steps respectively. Naturally the population behaves similar to the solutions of the Lotka-Volterra equations, which proves consistency. The environment has a limit of 200 agents for each group, so that the computation time, especially the feed forward process of the neural network, is limited.

Figure 6.9 shows a detail of this simulation. To illustrate the dynamic, 5 sequential frames were stacked. The top most frame shows temporally the last one.

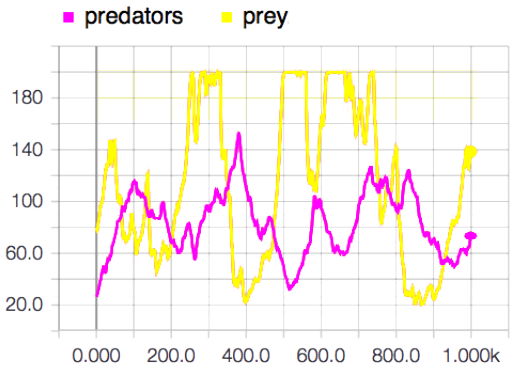

Figure 6.8: Population of predators and preys over 1000 episodes

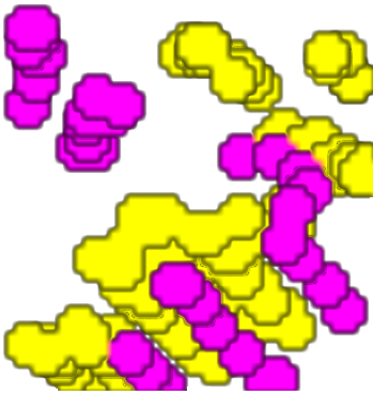

Figure 6.9: Detail of 5 sequential frames

The results were quite exciting. Not only that the predators and preys have learned the rules that would have been implemented in a classical agent based model. The predators also learned, that they can be more successful if they hunt together in groups. Due to the fact that the preys are 10% faster, a single predator won't be successful in hunting down a prey. This behaviour can be observed in figure 6.9, looking at the furthermost top right predator. Predators are represented by magenta coloured circles, preys are yellow. On the first frame, i.e. the bottom most frame, the predator of interest is hunting a single prey heading towards the right edge. Throughout these 5 frames one can see, that this predator loses interest of this prey as soon as one of the predators, which is hunting a group of preys along with two other predators, appears in his visual field. He joins the group in the hope of being more successful.

7. Conclusion and Prospects

The application of reinforcement learning in agent based modelling looks promising at first glance. With more computational power, it is possible to set up neural networks for each individual agent instead of using group control. Agents could evolve into individuals, which communicate with each other for planned actions and have a memory, e.g. using LSTMs [18]. Even with a simple setup like in section 6.2, it is possible to achieve remarkable results. Though, it always depends on the purpose of using agent based models. If an emergent behaviour is to be reproduced without knowing the agents individual behaviour, reinforcement learning can be a more straight forward approach than rule guessing and calibration.

It is not only about reducing the bias in the definition of rules, but also that the algorithm can find rules that were not even thought of. The definition of if-else conditions in the policy can be assumed by the algorithm. Let's recall the predator prey model of section 6.2, where the predators started to join other groups of predators instead of hunting on their own. If we had to define these behavioural rules in the classical approach, it could have been hard to determine some if-else conditions at some point. For example how and when should predators join the group without getting too close to the others. Overlapping predators wouldn't use the group's size at it's full capacity. These kind of behaviour is found automatically by the reinforcement learning algorithms.

A disadvantage of RL in ABM is the number of hyperparameters that have to be set for a successful learning process. Decisions concerning which RL paradigm or function approximator to choose, aren't obvious. There is not even a unique right decision, still there are multiple that don't lead to success. This can be a frustrating part. Another limitation is that this pre-simulation can take a lot of resources such as time or computational power, that isn't available in some situations. Especially if complex models require a complex reward function. The problem of defining a meaningful reward function is that there is only one dimension, i.e. it's codomain, in which the reward has to be expressed. If there are several instances of positive or negative rated states, it can be hard to correctly weight the rewards for a meaningful feedback. Reward functions always have to be designed in a way that the agents uniquely can find out which behaviour is desired and which isn't. The

more complex the model problem, the harder it is to find a meaningful reward function and the harder it will be to find functional hyperparameters.

Apart from these difficulties, reinforcement learning in agent based modelling can provide a new perspective on difficult problems, that makes it easier to solve them.

Especially on models that are dependent on data, that is only partially available. In this case, the sparse data could be just enough to extract a reward function that could help to find the rest of the data through reinforcement learning. This extraction of a reward function could be accomplished with inverse reinforcement learning (see [16] for details).

One concrete model, where this could be applied is the development of nevi and melanoma. This is still an open problem. While data about the cell movement of melanocytes and already developed nevi and melanoma is available, the question how and from which cell nevi develop remains [7].

Furthermore Wang et al. present a new model for the cell movement in the early phase of C. elegans, using Q-learning with neural networks [27].

List of Figures

Bibliography

[1] Robert Axelrod et al. 'The evolution of strategies in the iterated prisoners dilemma'. In: *The dynamics of norms* (1987), pp. 1–16.

[2] Eric Bonabeau. 'Agent-based modeling: Methods and techniques for simulating human systems'. In: *Proceedings of the National Academy of Sciences* 99.suppl 3 (2002), pp. 7280–7287.

[3] John Conway. 'The game of life'. In: *Scientific American* 223.4 (1970), p. 4.

[4] George Cybenko. 'Approximation by superpositions of a sigmoidal function'. In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989), pp. 303–314.

[5] Alborz Geramifard et al. 'A tutorial on linear function approximators for dynamic programming and reinforcement learning'. In: *Foundations and Trendső in Machine Learning* 6.4 (2013), pp. 375–451.

[6] Xavier Glorot, Antoine Bordes and Yoshua Bengio. 'Deep sparse rectifier neural networks'. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[7] James M Grichnik et al. 'How, and from which cell sources, do nevi really develop?' In: *Experimental dermatology* 23.5 (2014), pp. 310–313.

[8] Kurt Hornik, Maxwell Stinchcombe and Halbert White. 'Multilayer feedforward networks are universal approximators'. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: http://www.sciencedirect.com/science/article/pii/0893608089900208.

[9] Shin Ishii et al. 'A reinforcement learning scheme for a partially-observable multi-agent game'. In: *Machine Learning* 59.1-2 (2005), pp. 31–54.

[10] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/convolutional-networks/.

[11] Florian Kitzler and Martin Bicher. 'Case Studies for a Markov Chain Approach to Analyze Agent-Based Models'. In: *International Conference on Computer Science and Communication Engineering*. 2015.

[12] Joel Z Leibo et al. 'Multi-agent reinforcement learning in sequential social dilemmas'. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2017, pp. 464–473.

[13] Michael L Littman, Thomas L Dean and Leslie Pack Kaelbling. 'On the complexity of solving Markov decision problems'. In: *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 1995, pp. 394–402.

[14] Warren S McCulloch and Walter Pitts. 'A logical calculus of the ideas immanent in nervous activity'. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[15] Volodymyr Mnih et al. 'Asynchronous methods for deep reinforcement learning'. In: *International conference on machine learning*. 2016, pp. 1928–1937.

[16] Andrew Y Ng, Stuart J Russell et al. 'Algorithms for inverse reinforcement learning.' In: *Icml*. 2000, pp. 663–670.

[17] Michael A. Nielsen. *Neural networks and deep learning*. 2015. URL: `http://neuralnetworksanddeeplearning.com/`.

[18] Chris Olah. *Understanding LSTMs*. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Online; accessed 24th October 2018]. 2015.

[19] Megan M Olsen and Rachel Fraczkowski. 'Co-evolution in predator prey through reinforcement learning'. In: *Journal of computational science* 9 (2015), pp. 118–124.

[20] Sebastian Ruder. 'An overview of gradient descent optimization algorithms'. In: *Computing Research Repository* abs/1609.04747 (2016). URL: `http://arxiv.org/abs/1609.04747`.

[21] Thomas C Schelling. 'Dynamic models of segregation'. In: *Journal of mathematical sociology* 1.2 (1971), pp. 143–186.

[22] Satinder P Singh, Tommi Jaakkola and Michael I Jordan. 'Learning without state-estimation in partially observable Markovian decision processes'. In: *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 284–292.

[23] Felipe Petroski Such et al. 'Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning'. In: *arXiv preprint arXiv:1712.06567* (2017).

[24] Richard S Sutton and Andrew G Barto. 'Reinforcement learning: An introduction'. In: (2011).

[25] Richard S Sutton et al. 'Policy gradient methods for reinforcement learning with function approximation'. In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[26] V. B. Tadic. 'On the almost sure rate of convergence of linear stochastic approximation algorithms'. In: *IEEE Transactions on Information Theory* 50.2 (Feb. 2004), pp. 401–409. ISSN: 0018-9448. DOI: 10.1109/TIT.2003.821971.

[27] Zi Wang et al. 'Deep Reinforcement Learning of Cell Movement in the Early Stage of C. elegans Embryogenesis'. In: *arXiv preprint arXiv:1801.04600* (2018).

[28] Fayçal Yahyaoui and Mohamed Tkiouat. 'Partially observable Markov methods in an agent-based simulation: a tax evasion case study'. In: *Procedia Computer Science* 127 (2018), pp. 256–263.

[29] Yaodong Yang et al. 'An Empirical Study of AI Population Dynamics with Million-agent Reinforcement Learning'. In: *CoRR* abs/1709.04511 (2017). arXiv: 1709.04511. URL: http://arxiv.org/abs/1709.04511.