UNIVERSITY OF ŽILINA

FACULTY OF ELECTRICAL ENGINEERING
AND INFORMATION TECHNOLOGY

HABILITATION THESIS

ING. MICHAL GREGOR, PHD.

Towards Intelligent Agents using Deep Reinforcement Learning

Filing Number: 28260220195002

Žilina, 2019

UNIVERSITY OF ŽILINA

FACULTY OF ELECTRICAL ENGINEERING
AND INFORMATION TECHNOLOGY

HABILITATION THESIS

ING. MICHAL GREGOR, PHD.

Towards Intelligent Agents using Deep Reinforcement Learning

Field of Study: 5.2.14 Automation

Institution: University of Žilina, Faculty of Electrical Engineering and Information Technology, Department of Control and Information Systems

Žilina, 2019

Acknowledgements

I would like to express my sincere gratitude to all my friends and colleagues, who made this thesis possible. Special thanks go to Dušan Nemec – for the countless and long scientific discussions that often kept him from doing his own work. To Erickson Nascimento, Isabella Huang, Edson Roteia Araujo Junior, Cole Rose and other team members with whom I have collaborated on the psychoacoustic-annoyance-related feasibility study; to prof. Ruzena Bajcsy for advice, for leading the team and for hosting me at UC Berkeley. To my friends for understanding the fact that I could not drink beer with them and write this thesis at quite the same time.

Abstract

The habilitation thesis considers the problem of designing intelligent agents that has been at the core of the field of artificial intelligence and machine learning for many years. It discusses why and how this is a challenging multi-disciplinary problem and what components might be required to form an intelligent agent. After presenting the author's related prior work, the thesis proceeds to discuss the basics of deep reinforcement learning as a promising framework that could bring a lot of the necessary components together. It then introduces two case studies – one about the use of abstraction and visual attention, and the other about multisensory reinforcement learning. These demonstrate the flexibility of the deep reinforcement learning framework, but they also highlight the fact that to achieve good results, the learning system often needs to receive appropriate cognitive bias. This cognitive bias can often be provided to it through careful deep architecture design grounded in prior knowledge about the problem.

Keywords: machine learning, reinforcement learning, artificial intelligence, deep learning

Abstrakt

Habilitačná práca sa venuje problému návrhu inteligentných agentov, ktorý leží v jadre záujmu v oblasti umelej inteligencie a strojového učenia už mnoho rokov. Práca ukazuje prečo a v akom zmysle ide o náročný multidiscipilárny problém a tiež aké komponenty by mohli na vytvorenie takéhoto inteligentného agenta byť potrebné. Po tom, ako práca odprezentuje autorove súvisiace predchádzajúce výsledky, venuje sa základom hlbokého učenia s odmenou. Práve tieto metódy predstavujú v súčasnosti sľubný framework, ktorý by mohol byť schopný mnoho z potrebných komponentov integrovať. Práca ďalej uvádza dve prípadové štúdie – jedna sa venuje téme abstrakcie a vizuálnej pozornosti a druhá multisenzorickému učeniu s odmenou. Jednak pomocou nich demonštruje flexibilitu hlbokého učenia s odmenou, ale tiež ukazuje, že na získanie dobrých výsledkov je často potrebné systém vybaviť vhodnými kognitívnymi preferenciami. Tieto preferencie mu je často možné poskytnúť vhodným návrhom hlbokej architektúry, ktorý vychádza z predošlých znalostí o probléme.

Kľúčové slová: strojové učenie, učenie s odmenou, umelá inteligencia, hlboké učenie

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AI artificial intelligence, pp. 17–19, 21, 22

ANN artificial neural network, p. 20

CNN convolutional neural network, p. 135

DDPG deep deterministic policy gradient, pp. 3, XVI–XIX, XXI

DP dynamic programming, pp. 56, 57, 62, 68, 70, 72

DQN deep Q network, pp. 97–101, 114, 117, 118, 123, 134, 135, XVI–XIX, XXI, XXX, XXXI

ELU exponential linear unit, p. II

FCM fuzzy cognitive map, pp. 20, 21, 23

GP genetic programming, pp. 15, 17

GPS global positioning system, pp. 142, 144

GRU gated recurrent unit, pp. 106, 147, XXXVII

HMI human machine interface, pp. 14, 26

LSTM long short-term memory, pp. 106, 147, XXVII, XXXVII

MC Monte Carlo, pp. 70, 72

MDP Markov decision process, pp. 32, 34–36, 39, 40, 46, 56, 57, 62, 63, 62, 65, 70, 72, 130

MLP multi-layer perceptron, pp. 135, 138, 139, III

PA psychoacoustic annoyance, pp. 132, 135, 138, 135, 138–140, 142, 144, 145

PG policy gradient, pp. 86, XI, XII

POMDP partially observable Markov decision process, p. 130

PPO proximal policy optimization, pp. 3, 101, 114, XIV–XIX

RBF radial basis function, p. 82

ReLU rectified linear unit, p. II

RL reinforcement learning, pp. 2, 3, 16–18, 21–24, 31, 34–36, 38–40, 43, 57, 68, 71, 74, 81, 84–86, 93, 95, 97, 99, 101, 103, 104, 107–109, 118, 119, 123, 125, 126, 125, 127–130, 144, 145, XIX, XX

SAC soft actor-critic, pp. 3, 114, 118, XIX, XXXI

SARSA state action reward state action, pp. 72–75

SLAM simultaneous localization and mapping, p. 24

TD temporal difference, pp. 56, 57, 71–74, 76, 77, 79–81, IX, X

TRPO trust region policy optimization, pp. XII, XIV, XV, XVII

CHAPTER 1

INTRODUCTION

One of the key long-term focuses in the field of artificial intelligence is to design intelligent agents. Naturally, this task is not easy. It is certainly not made easier by the fact that we cannot even agree on what exactly constitutes intelligence – or agenthood for that matter. Leaving such philosophical questions aside though, designing intelligent agents is a very broad and multi-disciplinary problem. With the recent advances in deep learning, computer vision, reinforcement learning, speech and natural language generation and processing, we have come closer to the goal than ever before. Nevertheless, great challenges remain.

We have powerful deep learning systems, which can process audio and visual data. We are able to do image recognition and object detection, but these tasks require large amounts of manually annotated data. At the present state of the art, they also cannot necessarily provide sufficient performance or safety guarantees.

We have powerful speech recognition systems, but they still struggle to come up with the right answers when the acoustic conditions are not ideal, or the speaker's pronunciation diverges too far from what the system has been trained with. We are able to train ever-stronger language models and machine translation systems – but we can only get so far in this direction without the systems being able to reason logically about the statements that they are processing. Perhaps even more critically, it remains unclear how to ground natural language – so that the system not only models its structure, but that it can also attach some meaning to it: to connect the tokens with particular concepts that it has experienced.

Apart from perception and communication, an intelligent agent needs the ability to decide and act – to pick actions that will meet its goals and requirements. To perform short-term and long-term planning, when necessary and, naturally, to execute the actions that it decides upon.

Deep reinforcement learning provides a promising framework, which might be able to connect a lot of the aforementioned components into a single system in an unprecedented

manner. However, it has its own open problems, such as the low sample efficiency of existing deep reinforcement learning methods, which – for some tasks – can easily take 200 years of simulated time or more to learn a useful policy.

There is also the fact that the standard formulation of reinforcement learning requires a reward function, which would give positive feedback to the agent for performing the desired behaviour and negative feedback for behaving incorrectly. For some tasks, to specify such function can be comparably difficult to having to design the correct behaviour by hand.

However, several notable advancements in the field provide reasons for cautious optimism – it seems reasonably probable that at least some of these problems will be favourably resolved in the (hopefully near) future. To give some instances of these developments, there is the work in making reinforcement learning more efficient by pre-training the agents or some of their components so that they do not need to learn the target task from scratch.

These approaches include e.g. *(a)* representation learning, which would provide the agent with a pre-learned ability to perceive the relevant features of the problem; *(b)* meta reinforcement learning, i.e. agents, which are learning how to learn; *(c)* intrinsic motivation, such as curiosity, which helps the agent learn useful behaviours even without an external reward function; and more.

In this work, we set out to address the problem of designing intelligent agents using deep reinforcement learning. Our main hypotheses are as follows:

1. **Cognitive bias through architecture design:** We believe that a lot of useful prior knowledge about how a problem should be approached can be encoded into the agent through careful design of its architecture. This prior knowledge can provide useful cognitive bias to the agent.
2. **Abstraction and attention:** One way to introduce useful cognitive bias is to incorporate mechanisms such as abstraction and visual attention into the agent's architecture, thus increasing its efficiency.
3. **Multi-sensory RL:** Deep reinforcement learning is sufficiently flexible to handle several multisensory streams of data (such as audio, video, tabular data, natural language, ...). Such flexibility would have been extremely difficult to achieve in the past using different methods. However, we intend to show that in order to take full advantage of these capabilities, it is again necessary to provide the appropriate cognitive bias to the agent by selecting a suitable architecture.

In the remaining part of the thesis, we will attempt to present a brief introduction into the current state of the art in deep reinforcement learning and then to focus on the issues outlined above. The rest of the document is organized as follows (the list also describes some of the main contributions that the thesis intends to make):

- **Fundamental concepts:** An overview of the relevant fundamental terms in artificial intelligence and machine learning (chapter 2);
- **The context:** An overview of the various components that an intelligent agent might need, depending on the particular application (chapter 3);
- **Our prior work:** An overview of the work that we have done previously in this area, and how it all converges onto the problem of designing intelligent agents (chapter 3).
- **Reinforcement learning:** The fundamentals of reinforcement learning (chapter 4), including both: value-based (chapter 5) and policy-based (chapter 6) methods.
- **Deep learning:** A brief introduction to deep learning, including the fundamentals, the intuition for why deep architectures tend to be more expressive, and convolutional neural networks (appendix A).
- **Deep reinforcement learning:** A chapter on deep reinforcement learning (chapter 7), which explains the main difficulties of using deep models in reinforcement learning and shows how the deep Q-learning method deals with them. The chapter is continued in appendix B with other methods, such as DDPG, PPO and SAC.
- **Abstraction and attention:** Deep reinforcement learning is typically a lot less sample-efficient than conventional RL because in addition to the task, it also needs to learn feature extraction. In chapter 8 we show that it may be time to revisit the concepts of abstraction and attention in this context. Building visual attention and abstraction into the agent's architecture can provide it with useful cognitive bias, increasing the likelihood that it will generalize correctly, given a small amount of samples.
- **Multisensory RL:** Chapter 9 deals with agents that learn from multisensory observations. It describes how architectures that combine several different streams of multisensory data should be designed.It also presents a feasibility study, which applies deep reinforcement learning to a task, where the agent needs both: audio and tabular data to perform efficiently.

CHAPTER 2

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

In order to better acquaint the reader with the context of the thesis, it will be expedient to include a brief chapter, summarizing some basic concepts from the area of artificial intelligence and machine learning. The goal will be to not only introduce some of the basic terms, which the following chapters are going to build upon, but also to indicate in what ways the theory of reinforcement learning forms part of the broader area of artificial intelligence, machine learning and possibly other related areas.

2.1 | Types of Machine Learning: By Task

If we are now to focus on the area of machine learning, as one particular branch of artificial intelligence, it is necessary to remark there are several flavours of machine learning – each addressing a different task [1, 2]:

- supervised learning;
- reinforcement learning;
- unsupervised learning;
- semi-supervised learning.

For each of these types, we will now provide a short description.

2.1.1 Supervised Learning

The characteristic property of *supervised learning* is that the learning system receives feedback. It is first given the so-called training dataset, which has the following form:

$$(\mathbf{x}_1, \mathbf{y_1}), (\mathbf{x}_2, \mathbf{y_2}), ..., (\mathbf{x}_i, \mathbf{y_i}), ..., (\mathbf{x}_N, \mathbf{y_N}). \tag{2.1}$$

where $\mathbf{x}_i$ denotes the i-th input of the learning system and $\mathbf{y}_i$ denotes the desired output that corresponds to it. If the function of the system is f_θ, where θ is a vector of parameters,

the actual output of the system for input $\mathbf{x}$ will be $\hat{\mathbf{y}}$, such that:

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}). \tag{2.2}$$

The task of supervised learning can then be formally described as follows:

$$\theta^* = \arg\min_\theta \sum_{(\mathbf{x}_i, \mathbf{y}_i)} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i) \tag{2.3}$$

i.e. the goal is to find a vector of parameters θ, which will minimize the difference between the actual outputs of the learning system $\hat{\mathbf{y}}_i = f_\theta(\mathbf{x}_i)$ and the desired outputs $\mathbf{y}_i$ in terms of some predefined loss function $\mathcal{L}$ (e.g. the mean squared error, the cross-entropy etc.).

Supervised learning is typically used to solve one of the following two tasks

- *Classification:* the desired output is a categorical variable with a finite and relatively small domain.
- *Regression:* the desired output is continuous (it can also be multi-dimensional).

Other types of tasks, to which supervised learning can be applied, are usually derived from these two basic types in some way.

2.1.2 Reinforcement Learning

Reinforcement learning is typically applied to sequential decision making problems, i.e. problems, in which the learning system (the agent) makes decisions (chooses actions) in across several subsequent time steps. The system receives feedback – much like in the case of supervised learning. It is the character of this feedback that is different in the case of reinforcement learning. The desired outputs are no longer known directly. Instead, the system only receives a real-valued signal, that indicates how well it is doing. The system is rewarded for correct behaviour by being given rewards and punished for incorrect behaviour (by lower or negative rewards). The system is trying to find a policy, which would maximize the rewards in the long term.

The goal of reinforcement learning can again be described formally. However, there are several ways in which the problem can be framed and these result in different formal criteria for the goal. To give some idea of how these criteria are structured, we may give the following instance:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p(\tau|\pi_\theta, \mathcal{T})} \left\{ \sum_t r(s_t, a_t) \right\}. \tag{2.4}$$

The learnable parameters of the system are again denoted by θ here. Symbol τ denotes the potential state-action trajectories traversed by the agent. It depends on the actions taken by the agent on one hand (given by policy π_θ), and on the way in which the environment responds to those actions on the other hand (given by the transition function $\mathcal{T}$). The process which generates τ can then be jointly described using a probability distribution

$p(\tau|\pi_\theta, \mathcal{T})$, which depends on both factors. Function $r(a, s)$ is the reward function, which at every time step t gives the system reward $r(s_t, a_t)$ for performing action a_t in state s_t.

The goal of reinforcement learning then, as equation (2.4) shows, is to find a vector of parameters θ^* for the learning system, such that the expected value of the sum of all rewards is maximized. It is to be noted that most reinforcement learning methods do not optimize this objective directly, but typically use some surrogate objective. In order to go through equation (2.4) in more detail manner and to present a few practical reinforcement learning methods, we will need to discuss all of the related concepts in more detail – a task which we leave to a dedicated chapter (see 4).

2.1.3 Unsupervised Learning

In unsupervised learning – in contrast to the two above-mentioned paradigms – the learning system receives no feedback. Unsupervised learning is based on the principles of self-organization. Its goal depends on the particular method being used – it is usually related to identifying some interesting relationships or structure in the data [2]. Tasks of this kind may include, for an instance, clustering, anomaly detection, extraction of main characteristics, vector quantization, dimensionality reduction and other [2, 3].

Given that there are many kinds of unsupervised learning and each of them may have slightly different goals, it would not be practical to attempt to provide a common formal definition for all the types. Moreover, a great amount of variability is present also within the individual sub-types of unsupervised learning themselves. To make matters yet more complicated, not all unsupervised learning methods do in fact optimize utility functions that would be easy to formulate in a formal manner. For these reasons we will hereinafter only describe the selected kinds of unsupervised learning in an informal manner and we will not attempt precise formal definitions.

Clustering

The goal of most clustering methods is to identify clusters of samples in a dataset. A schematic example of this kind of task in a 2-dimensional space is given in Fig. 2.1.

Clusters can in most cases be informally characterized as groups of points, for which it is true, that the distances among the points within the cluster are smaller than the distances to points outside the cluster.

In general, clustering can be carried out over non-euclidean as well euclidean spaces. Indeed, many methods are able to handle spaces with very non-trivial distance metrics.

The form that the results of clustering take, may also differ according to the method that is being used. In the case of some methods, for an instance, it is necessary to specify the number of the clusters beforehand. Other methods are able to identify the number of

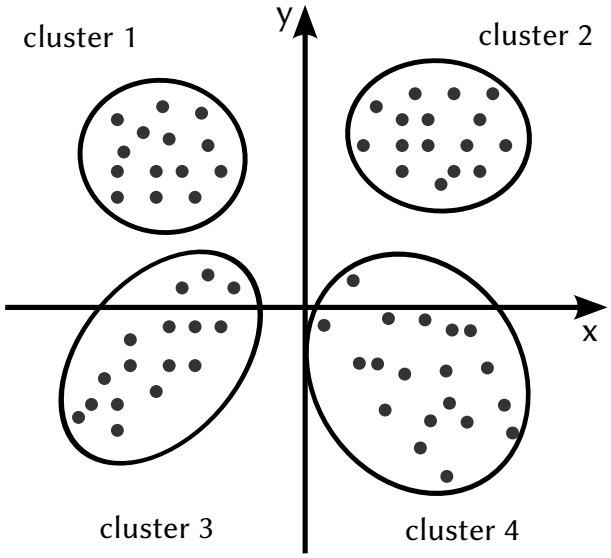


Fig. 2.1: Identification of 4 clusters in a dataset.

clusters automatically. The result of clustering will often be the assignment of every single sample from the dataset into one of the clusters.

Dimensionality Reduction

When doing dimensionality reduction, we transform vectors from the input space into a lower-dimensional space. The learning system is supposed to find a mapping from the one space to the other, which will minimizes the amount of useful information lost in the process as far as possible.

As an example of this kind, we may mention dimensionality reduction on the MNIST dataset of handwritten digits. The original data consists of black and white images of 28 $\times$ 28 pixels. Several randomly selected samples from the dataset are shown in Fig. 2.2 for illustration.

At the specified resolution and using a single colour channel, the input space will be 784-dimensional. It would therefore not be easy to visualize it without doing dimensionality reduction. However, if we first reduce the data into a 2-dimensional space – e.g. using UMAP [4] – visualization becomes easy. An example of what such visualization might look like is given in Fig. 2.3.

It is obvious that the original space does have some interesting structure – the reduced data are grouped into several distinct clusters. Given that there are exactly 10 clusters, we may expect that these will in fact correspond to the 10 individual digits. We can verify this by colour-coding the points in the plot according to which digit they represent. The result, shown in Fig. 2.4 clearly indicates that the clusters do indeed correspond to digits. However, unsurprisingly, there is also some noise in the data and a small number of the samples seem to be clustered with the wrong digits.

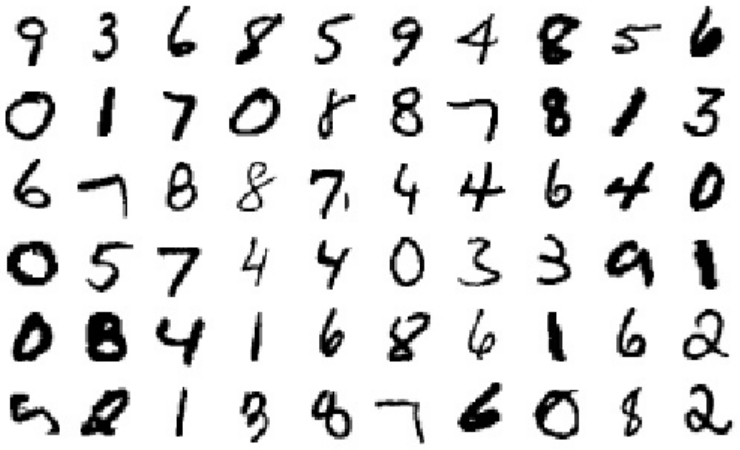


Fig. 2.2: Samples from the MNIST dataset.

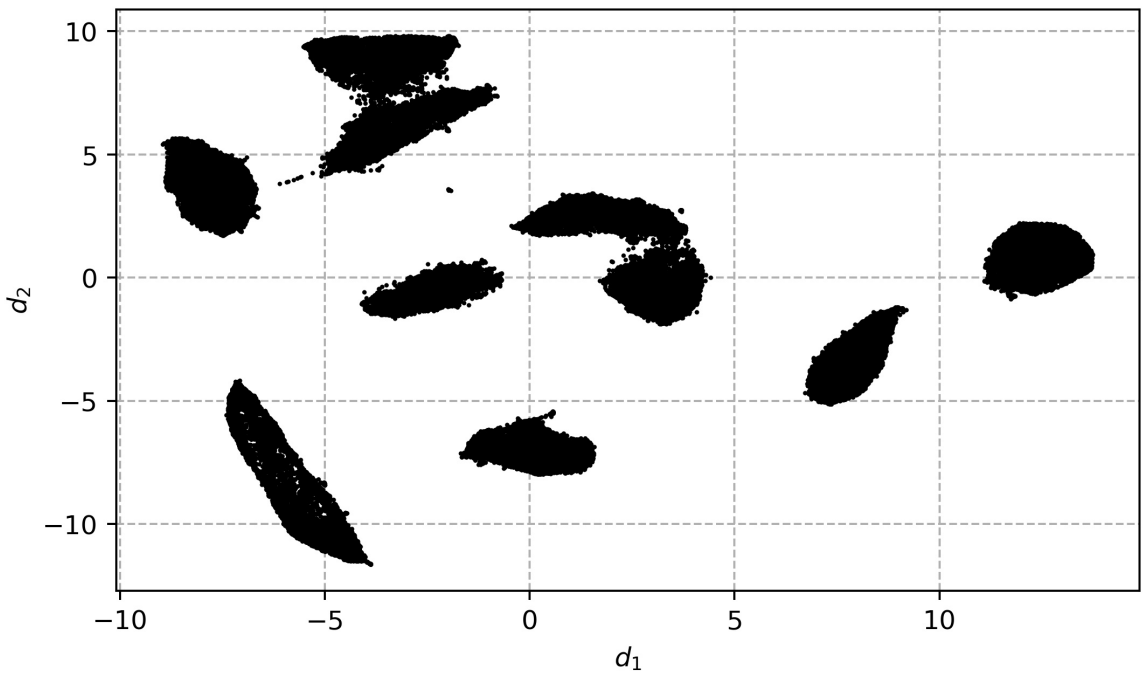


Fig. 2.3: Visualization of the MNIST dataset reduced into 2-dimensional space using UMAP.

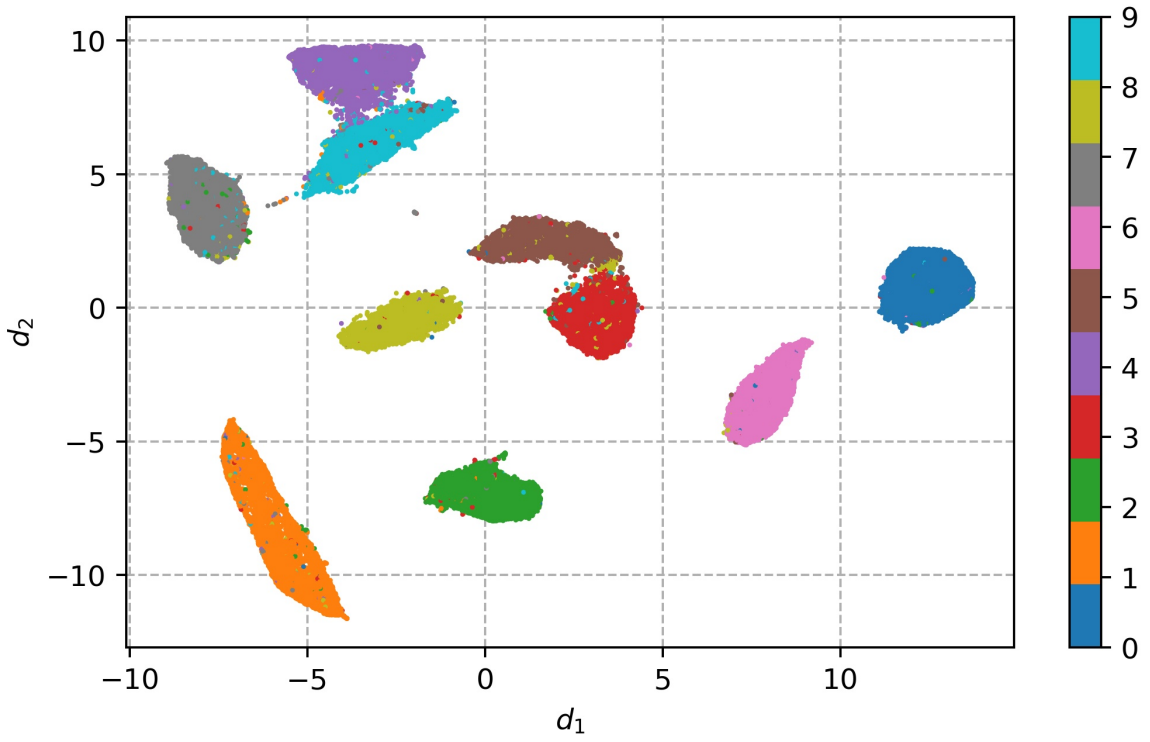


Fig. 2.4: Visualization of the MNIST dataset reduced into 2-dimensional space using UMAP, with points colour-coded according to which digit they represent.

2.2 | Generalization

Generalization is one of the key concepts in machine learning theory. The ability of a learning system to generalize is the ability to use the knowledge obtained through learning to correctly react to previously unseen inputs (i.e. such that the system did *not* encounter during learning).

A good example of why the ability to generalize is vital, may be taken from the area of image or speech recognition. It is obvious that if we train a system to perform speech recognition, we are not only asking it to be able to recognize the samples that we have shown it during training. Indeed, that would not be very useful. We probably want to make new recordings and have the system transcribe those into text, for an instance. if possible, we would also like the system to be robust to reasonable amounts of noise, and to generalize to different speakers, speech rates, microphone types, etc.

2.3 | Global and Local Generalization

In addition to the general information, which we have stated above, it will be useful to clarify a distinction that exists between two substantially different kinds of generalization, namely [5]:

- *local generalization*: concerns points from the input space, which are near to each other with respect to some simple distance metric – e.g. the euclidean distance;
- *global generalization*: it is able to recognize inputs that share the same structure, even though they may be very far from each other in the input space.

For a good non-technical example that aptly illustrates the distinction between local and global generalization, one may (somewhat surprisingly) turn to mushroom hunting. The ability to generalize locally, while mushroom hunting, would amount to realizing that wherever we have found one mushroom, there may be several – this is why we tend to search the immediate surroundings of each mushroom before moving on. We may even return to some of the good spots when we go mushroom hunting again next time.

Global generalization operates at a higher level. An experienced mushroom hunter will start to notice things that good mushroom-hunting spots have in common: they will start to perceive the structure of the task. They may, for an instance, start to give more of their attention to humid spots that have a bit of shade.

CHAPTER 3

THE CONTEXT OF THE THESIS

I have spent both – my studies and my later academic life – so far at the Department of Control and Information Systems (KRIS) at the Faculty of Electrical Engineering and Information Technology (FEIT) of the University of Žilina (UNIZA). I have also at various times participated in research and development activities involving artificial intelligence and machine learning at UNIZA's University Science Park (UVP) and at its Institute of Competitiveness and Innovations (UKaI), where my work still continues.

At DCIS we have been offering several courses related to artificial intelligence and machine learning. At present these include especially:

- Artificial Intelligence 1;
- Artificial Intelligence 2;
- Intelligent Transport Systems;
- Modelling of Telematic Systems.

Other similar courses were offered in the past, and have now been discontinued – such as Fuzzy Control, Expert Systems, Programming of Artificial Intelligence and others. The same scientific area has also formed an active part of our research – which has been inevitable because the main focus of our department has always been automation, which is constantly being informed by the progress in machine learning and artificial intelligence these days. Indeed, the progress in automation that we have witnessed in the past few years would be unimaginable without the recent breakthroughs in those fields.

In order to present the content of this work in its proper context, it will be expedient to give a brief overview of some selected relevant activities that I have participated in – both at KRIS and at UVP/UKaI. This will make clearer the way in which all these various lines of work tie in with each other and with our long-term goals. It will also help to explain the motivation and general vision behind some of our work.

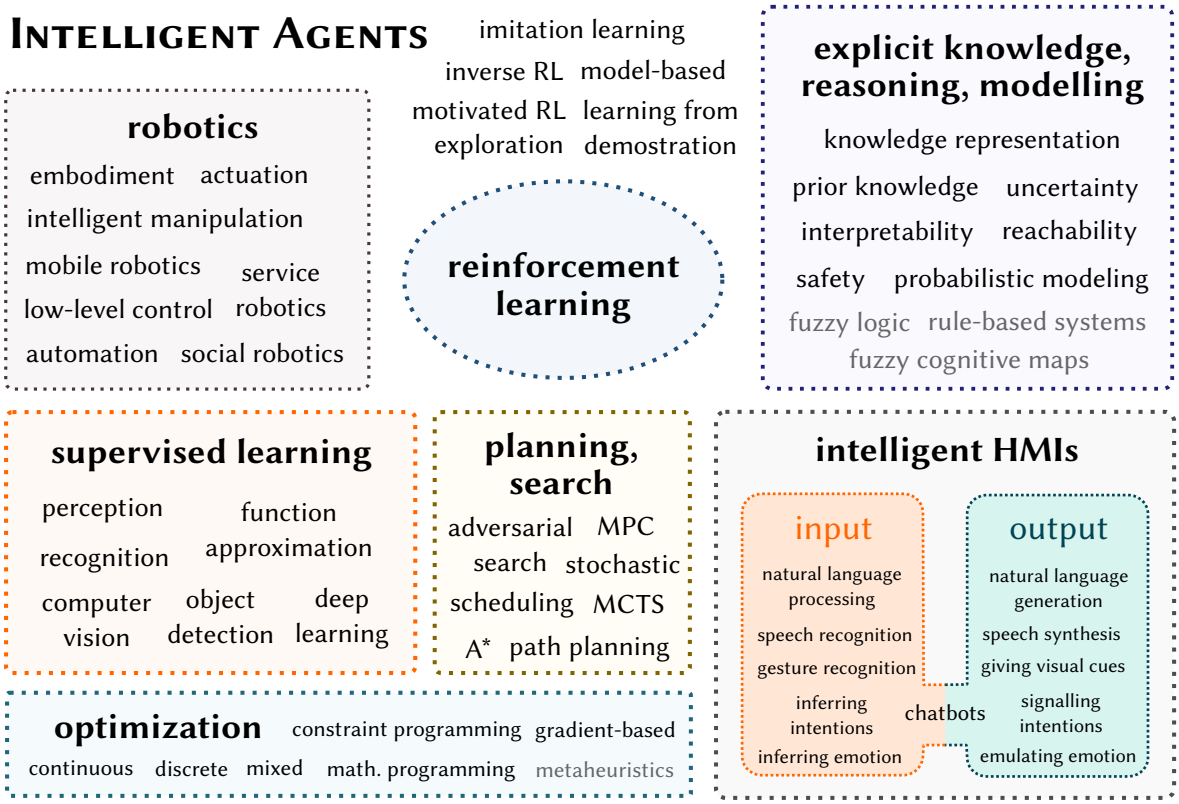


Fig. 3.1: Intelligent agents: related areas.

3.1 | Building Intelligent Agents is a Multi-Disciplinary Problem

One of the big questions in the field of artificial intelligence these days is how to build autonomous or semi-autonomous agents that can act intelligently in a given environment. The environments can be real-world, simulated or part of the general cyberspace (e.g. recommender systems and such).

Depending on the application, such agents may need to:

- learn from experience (whether online or from pre-collected data);
- do implicit/explicit reasoning;
- do planning and search for solutions of problems;
- communicate and interface with humans/other machines in effective ways;
- carry out physical actions that change the state of the real world;
- ...

It should be obvious that this skill set is quite diverse and that it encompasses a lot of different methods and approaches – indeed, that it is multi-disciplinary and spans many different areas. A subset of these is shown in Fig. 3.1, along with some of the relevant concepts.

At the centre is the area of reinforcement learning, which is concerned with learning how to act in order to meet some specified goals – and this is the main topic of the present work. However, there are many other areas – such as robotics, automatic planning, scheduling, optimization and more. The area of intelligent human machine interfaces (HMIs) is also very relevant in many applications and has been advancing quickly over the recent years – with increasingly powerful applications in natural language processing, speech recognition and synthesis, task-oriented chatbot systems and more.

The particular components in Fig. 3.1 that one needs to employ will, of course, vary considerably from application to application. However, in general, each of these plays a role and needs to be considered. We have previous work in several of these areas: the rest of this chapter will present some of it and place it in its proper context.

3.2 | Evolutionary Approaches

To provide the reader with the full context of this work, I need to mention that my first attempt to approach the problem of designing intelligent agents was actually not based on what we conventionally understand to be reinforcement learning. Even though the problem itself was essentially a reinforcement learning problem, I actually approached it using evolutionary computation – genetic programming (GP), to be more exact.

3.2.1 Problem Formulation and Performance Issues

In my master's degree thesis entitled "Applications of Artificial Intelligence Methods to Design and Control of Robotic Systems" I have arrived at a reasonably general version of the problem by reformulating the well-known artificial ant problem proposed by Koza [6]:

- Gregor, M. *Applications of Artificial Intelligence Methods to Design and Control of Robotic Systems*. Master's thesis, Žilinská univerzita v Žiline, Elektrotechnická fakulta, Katedra riadiacich a informačných systémov, 2011. Supervised by Juraj Spalek.

I have changed the mode in which the genetic program is executed and I have equipped the agent with explicit memory that the programs were able to access.

Unfortunately, this problem turned out to be very challenging for GP (actually, as it turns out, Koza's original formulation is similarly challenging – even though it is not as expressive). In order to successfully solve it, I had to design several additional mechanisms, such as the adaptive value-switching of mutation rate and the flood mechanism, which would adaptively modify GP's hyperparameters or carry out other reparation procedures once the evolutionary process got stuck.

These additional mechanisms allowed the GP to successfully arrive at a solution, but – in addition to the process being very sample-inefficient (meaning that a very large amount

of experience was necessary to make progress) – the resulting solution would not generalize very well.

3.2.2 Follow-up Work

I have addressed these issues further, in a number of follow-up publications, a small selection of which is listed here:

- Spalek, J. – Gregor, M. *Adaptive Approaches to Parameter Control in Genetic Algorithms and Genetic Programming*. Applied Computer Science: Improvements Methods in Manufacturing Design, Scheduling and Control, 7(1):38–56, 2011. ISSN 1895-3735. URL: <http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-f8a07e0a-7da5-43dd-afc3-5e0455daf098>;
- Gregor, M. – Spalek, J. *On Use of Node-attached Modules with Ancestry Tracking in Genetic Programming*. In 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems. 2012. ISBN 978-3-902823-21-2. URL: <https://doi.org/10.3182/20120523-3-CZ-3015.00027>;
- Gregor, M. – Spalek, J. *Using Context Blocks in Genetic Programming with JIT Compilation*. ATP Journal PLUS, (2), 2013. ISSN 1336-5010;
- Gregor, M. – Spalek, J. *Using LLVM-based JIT compilation in genetic programming*. In 2016 ELEKTRO, pp. 406–411. IEEE, 2016. ISBN 978-1-4673-8698-2. URL: <https://doi.org/10.1109/ELEKTRO.2016.7512108>.

These explore further ways to increase the scalability of the approach by introducing new adaptive mechanisms, several novel ways to implement reusable modules, and also by using just-in-time compilation to make execution of the genetic programs themselves faster.

However, even after adding all these extensions to the basic method, there are still issues trying to scale it to more challenging problems, which is why we have abandoned this direction and changed our focus to reinforcement learning.

3.2.3 The Underlying Issues

In fact, we now believe that this is a fundamental problem with genetic programming (and evolutionary methods in general). These approaches are extremely sample-inefficient by necessity, because they ignore virtually all information about the structure of the RL problem – they are in fact oblivious of the fact that the task involves sequential decision making at all.

After conducting an expensive many-step simulation, the evolutionary methods will collect a feedback signal consisting of a single scalar value. This extremely low-dimensional signal can be very uninformative – to compress the wealth of information produced during

each simulation run into a single scalar value is an impossible task. The problem is further aggravated when the fitness function is stochastic, in which case the scalar feedback signal can be very noisy.

Note also that each candidate solution needs to be evaluated at every epoch, so the expensive simulation has to be done many times, which is, of course, computationally very expensive, unless the simulation is extremely cheap to do.

Recently, new work has emerged, which uses evolutionary methods in conjunction with deep neural networks to do reinforcement learning. The pioneer work, based on evolutionary strategies, was done by OpenAI [12], with a follow-up by Uber AI Labs, which uses genetic algorithms instead [13].

Even though this approach is very different from applying GP to the problem, and despite the fact that the authors were able to report relatively good results, their work still raises the same basic issue: it suffers from terrible sample-inefficiency. This means that any such approach is only remotely useful for problems, where simulation is extremely cheap. The same is likely to remain true in the future, whenever a task is approached with evolutionary computation.

3.3 | Reinforcement Learning and Motivation

Following our largely unsuccessful efforts to use evolutionary approaches – especially genetic programming – as a way to design intelligent agents, I have turned my attention to reinforcement learning (RL). This branch of AI has proved to be much more promising – especially with the recent introduction of deep learning models into the field.

One of the long-standing issues in reinforcement learning is the sample efficiency of reinforcement learning methods. The term sample efficiency refers to how much experience an agent needs to collect, in order to successfully learn the required behaviour. Collecting experience can be very expensive, so naturally we would like an agent to be able to learn from as little experience as possible.

We have explored several ways to achieve this goal in our work concerning curiosity-driven exploration, novelty detection based on forecasting, optimistic exploration using a separate exploration value function, and more in:

- Gregor, M. – Spalek, J. *Novelty Detector for Reinforcement Learning Based on Forecasting*. In IEEE 12th International Symposium on Applied Machine Intelligence and Informatics: Proceedings, pp. 73–78. Herľany, Slovak Republic, 2014. ISBN 978-1-4799-3441-6. URL: <https://doi.org/10.1109/SAMI.2014.6822379>;
- Gregor, M. – Spalek, J. *Curiosity-driven Exploration in Reinforcement Learning*. In Proceedings of 10th International Conference, ELEKTRO 2014, pp. 435–439. Rajecké Teplice, Slovak Republic, 2014. ISBN 978-1-4799-3720-2. URL:

<https://doi.org/10.1109/ELEKTRO.2014.6848933>;

- Gregor, M. – Spalek, J. *The Optimistic Exploration Value Function.* In INES 2015: IEEE 19 th International Conference on Intelligent Engineering Systems, Proceedings, 19, pp. 119–123. Bratislava: IEEE, 2015. ISBN 978-1-4673-7938-0. URL: <https://doi.org/10.1109/INES.2015.7329650>;
- Gregor, M. *Control System of an Autonomous Robot for Solving Multi-objective Tasks.* PhD thesis, University of Žilina, 2014.

With the introduction of deep reinforcement learning, issues regarding sample efficiency have by no means been resolved – in fact they have been aggravated. Standard machine learning methods typically work on heavily preprocessed data and only need to learn how to produce the desired behaviour. Deep learning methods, on the other hand, try to learn both: the behaviour as well as the preprocessing and feature extraction. This makes deep learning much more effective in tasks, where it is difficult to design a good set of features, but it also means that the learning system is much more data-hungry.

For this reason, works that attempted to make classical RL more effective have been gaining new relevance. Indeed, some of these approaches are even now being re-examined in the context of deep reinforcement learning – a trend witnessed by a number of recent publications, such as the work of Pathak and Efros on curious RL agents [18, 19].

In a similar effort, we will also need to look into how our existing approaches could be transplanted from their original setting in the context of classical RL into that of deep RL. So far we have experimented with utilization of the concept of visual attention and tested its advantages in the game of Pac-Man:

- Gregor, M. – Nemec, D. – Janota, A. – Pirník, R. *A Visual Attention Operator for Playing Pac-Man.* In Proceedings of 12th International Conference, ELEKTRO 2018. 2018. ISBN 978-1-5386-4759-2. URL: <https://doi.org/10.1109/ELEKTRO.2018.8398308>.

Prior to that, there has been a thesis by Jakub Hanes, which also tackled Pac-Man using deep RL, but without using any of these more advanced concepts:

- Hanes, J. *Učenie s odmenou* [Reinforcement Learning]. Master's thesis, Žilinská univerzita, 2017. Supervised by Michal Gregor.

At the moment, we are also working on several applications of deep RL in the area of robotics.

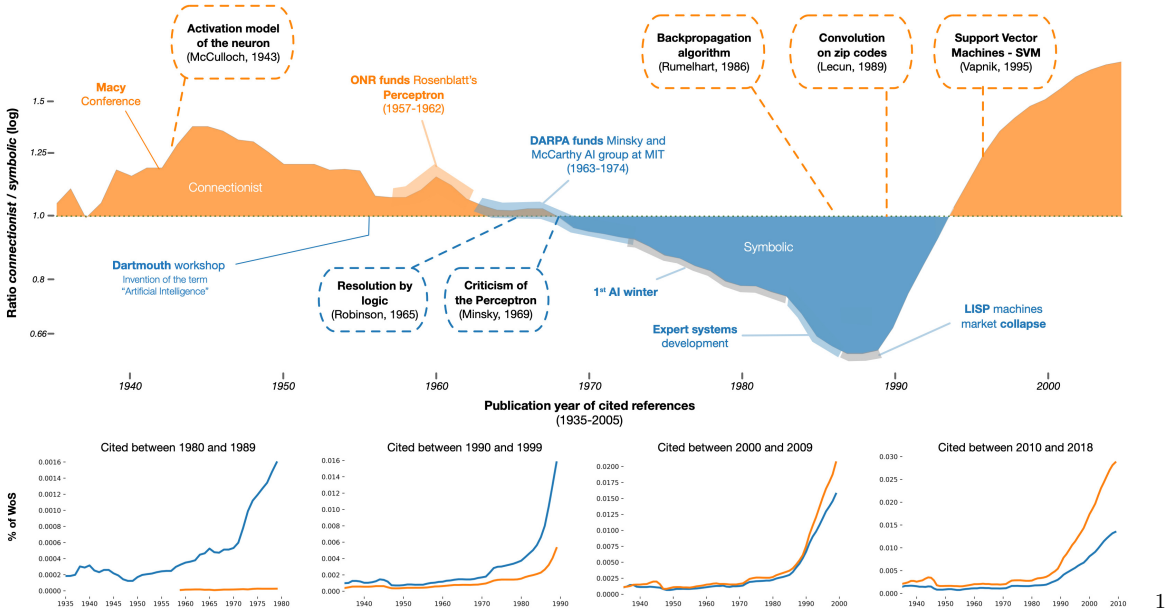


Fig. 3.2: Evolution of the academic influence of connectionist and symbolic approaches [22].

3.4 | Prior Knowledge and Explicit Knowledge Representations

In its long history, the area of artificial intelligence has seen a lot of interplay (and even some amount of competition) between two distinct types of approaches – symbolic AI (which works with representations based on logic) and sub-symbolic AI (a.k.a. connectionist AI, which uses approaches other than logic-based reasoning).

Over the years, different methods of symbolic and sub-symbolic AI have gained and lost traction and the balance between the two areas has shifted significantly multiple times. A rough graphical overview of this history has recently been given by Cardon et al. [22] and is presented in Fig. 3.2.

A further division, which is perhaps more useful, can be made between systems with explicit and implicit knowledge representations – whether those representations are implemented in a strictly symbolic way or not. That way, approaches such as fuzzy logic would count as systems with explicit knowledge representation, even though their actual implementation is sub-symbolic.

In any case, there is clearly a trade-off between methods with implicit and explicit representations. Implicit methods are typically geared towards machine learning: either from experience or from data and require less manual engineering. They are also good at learning to perform skills, which it is difficult to capture in high-level logic, such as riding a bicycle.

Methods with explicit representations, on the other hand, tend to be much more interpretable and their results are usually easier to reason about. They usually require a substantial amount of engineering: they very work with human-designed rule bases, which makes it difficult to scale them to very complex tasks, for which it may be very difficult to construct complete, correct and self-consistent rule-bases. On the other hand, it is very easy to make use of prior knowledge about the task when constructing explicit representations. It is not nearly as easy when using implicit methods.

3.4.1 Using Explicit Methods to Utilize Prior Knowledge

This last property: the fact that explicit methods make it easier to utilize existing prior knowledge about the task, has motivated me at one point to look into these methods. These approaches are currently in decline – and have been for a while – which can most likely be largely attributed to their failure to deliver upon the somewhat overblown promises that were made concerning them in the past. (This is something we would do well to remember whenever we are tempted to spread too much hype about the methods that are showing encouraging results now – however promising they may seem.)

Be that as it may, utilization of prior knowledge is still a consideration that cannot be taken lightly. In the context of reinforcement learning and intelligent agents in general, there are many kinds of knowledge that it would clearly be useful to convey to the agent before learning starts.

To see why this is a problem, let us consider an agent that is supposed to learn how to play chess. Clearly, if the agent is not made aware of what the rules of the game are beforehand, this will make the task much, much more difficult. In fact, it is dubious whether an average human would be able to learn to play the game under the same circumstances.

Fuzzy Cognitive Maps

In any case, as a way to address this important issue, we have explored several methods with more or less explicit representations. In 2013, during my 3-month Erasmus stay at the University of Patras, Greece, I have worked closely with the research group of prof. Peter Groumpos, who specializes in fuzzy cognitive maps (FCMs) [23–25].

The architecture of fuzzy cognitive maps closely resembles that of certain types of recurrent artificial neural networks (recurrent ANNs). The most important difference is that each node in the network represents a distinct concept and the connections between the nodes model the mutual relationships of those concepts. The representation is therefore relatively easy to interpret and in fact traditionally, FCMs would not be learned from data, but designed by experts.

My initial idea, therefore, was to try using an FCM as a value function approximator, which would allow us to incorporate the prior knowledge into the weights. At the time, the

FCM theory had been using several unsupervised learning methods originally designed for ANNs – so making use of the similarities between FCMs and ANNs. What we have explored instead was how FCMs could be trained using supervised, gradient-based methods – a topic we have explored with good results in several papers:

- Gregor, M. – Groumpos, P. P. *Tuning the Position of a Fuzzy Cognitive Map Attractor using Backpropagation through Time*. In Proceedings of The 7th International Conference on Integrated Modeling and Analysis in Applied Control and Automation (IMAACA 2013), pp. 78–86. Athens, 2013. ISBN 978-1-62993-488-4;
- Gregor, M. – Groumpos, P. P. *Training Fuzzy Cognitive Maps using Gradient-based Supervised Learning*. In Artificial Intelligence Applications and Innovations (9th IFIP WG 12.5 International Conference, AIAI 2013, Paphos, Cyprus, September 30 – October 2, 2013, Proceedings), IFIP Advances in Information and Communication Technology, pp. 547–556. Cyprus: Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41142-7. URL: <http://dx.doi.org/10.1007/978-3-642-41142-7_55>;
- Gregor, M. – Groumpos, P. P. – Gregor, M. *Using Weight Constraints and Masking to Improve Fuzzy Cognitive Map Models*. In Conference on Creativity in Intelligent Technologies and Data Science, pp. 91–106. Springer, 2017. ISBN 978-3-319-65551-2. ISSN 1865-0937. URL: <http://dx.doi.org/10.1007/978-3-319-65551-2_7>.

While this allowed us to train FCMs to a much higher level of precision, it did not solve the main problem: FCMs are not universal function approximators. Every node has its associated concept and there are no free nodes that could model complex relationships between them. If such relationships exist, the expert is required to manually insert all the intermediary concepts required to construct them. This proved to be especially impractical in the context of RL, which is why I have abandoned the idea later.

Some researchers work around the issue by inserting a number of free nodes into the FCM. These are then not associated with any specific concepts and are free to model the relationships. However, this approach effectively turns the FCM into a recurrent neural network and sacrifices the interpretability – thus eliminating the property we wanted to choose the FCM for in the first place.

Fuzzy Logic

Fuzzy logic is another way of explicitly representing and utilizing knowledge that has received a lot of attention in the past and which has also formed a very large part of the AI-related courses at our department at the time.

We have explored several ways in which these systems might be used. The main issue is that fuzzy logic systems are notorious for not being designed with support for machine learning in mind. Several approaches to tuning the membership functions and learning the

linguistic rules have been explored in the past – such as genetic fuzzy systems and fuzzy decisions trees.

We have experimented with some of these approaches ourselves, e.g. in:

- Gregor, M. – Miklošík, I. – Spalek, J. *Automatic tuning of a fuzzy meta-model for evacuation speed estimation.* In 2016 Cybernetics & Informatics (K&I), pp. 1–6. IEEE, 2016. ISBN 978-1-5090-1834-5. URL: <https://doi.org/10.1109/CYBERI.2016.7438594>;
- Kello, P. – Gregor, M. – Miklošík, I. – Spalek, J. *Learning a Fuzzy Model for Evacuation Speed Estimation using Fuzzy Decision Trees and Evolutionary Methods.* Zeszyty Naukowe Wyższej Szkoły Technicznej w Katowicach, (9):49–62, 2017. ISSN 2450-5552. URL: <http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-6fdc439d-1a22-4321-aaef-46237387713d>.

We have even started to develop a custom user interface that would integrate them:

- Kindernay, J. *Prestavba grafického užívateľského rozhrania pre knižnicu Fuzzylite* [Rebuilding of the graphical user interface for the Fuzzylite library]. Master's thesis, Žilinská univerzita, 2017. Supervised by Michal Gregor.

Unfortunately, in our experience these methods were neither very effective, nor did they offer much actual support for incorporating prior knowledge (which is not surprising, considering that they have been designed as an afterthought). Furthermore, they are less amenable to gradual and incremental learning than even artificial neural networks, which is a huge issue in RL applications. For these reasons, we did not proceed with the idea further.

Naturally, there is a host of other well-known objections to fuzzy logic, which could be raised. However, these do not seem especially relevant in the RL context and we will therefore refrain from discussing them.

First-order Logic and Logical Programming

Another branch of AI that is to be considered vis-à-vis explicit representations and reasoning is that of logic-based systems, such as systems that make use of first-order logic, description logics and more. The area of logical programming languages such as Prolog may also be relevant.

We have experimented with some of these approaches – especially with the Web Ontology Language (OWL), which is based on description logics, and with Prolog. Joined by researchers form UNIZA's University Science Park, we have implemented simple benchmark tasks, such as the well-known Zebra puzzle in both: Prolog and OWL, so as to compare how flexibly they could be modelled in each, and how efficiently they could be solved.

These results were published in several papers, such as:

- Gregor, M. – Zábovská, K. – Smataník, V. *The Zebra Puzzle and Getting to Know Your*

Tools. In INES 2015: IEEE 19th International Conference on Intelligent Engineering Systems, Proceedings, 19, pp. 159–164. Bratislava: IEEE, 2015. ISBN 978-1-4673-7938-0. URL: <https://doi.org/10.1109/INES.2015.7329698>;

- Bučko, B. – Hanušniak, V. – Joštiak, M. – Zábovská, K. *Inferring over functionally equivalent ontologies*. In Digital Information Processing and Communications (ICDIPC), 2015 Fifth International Conference on, pp. 201–206. IEEE, 2015. URL: <https://doi.org/10.1109/ICDIPC.2015.7323029>.

Unfortunately, the OWL reasoners we have experimented with, have proven to be extremely slow: the lack of speed was noticeable even on these simple toy problems. However, this is not surprising: it is well known that – depending on the expressions used – even a small OWL problem can easily become computationally intractable. For larger, more complex ontologies, reasoning can sometimes take days, or even not complete at all, depending on the reasoner.

However, the main problem with using these in the RL context is that there is no clear path from logic-based knowledge representations to the representations used in practical RL agents. With an FCM, for an instance, one could – at least in theory – initialize the weights using prior knowledge and then have RL take over. It is unclear what an equivalent of that could be when using representations based on logic (however, more on the interplay between implicit and explicit representations will be said in section 3.4.2).

3.4.2 The Interplay of Explicit and Implicit Knowledge

Even though implicit methods dominate the field of artificial intelligence and machine learning at this point, and methods that use explicit knowledge representations are in decline, this state of affairs is likely to change again at some point in the future. Explicit representations have their advantages – the ability to perform high-level reasoning is one of them, the ability to communicate knowledge (albeit imperfectly) is another, and there are many more.

In fact, results of several studies indicate, that mutual interaction of explicit and implicit knowledge representations forms an important part of human learning and intelligence. As an instance, one may mention the Hanoi tower experiment conducted by Sun et al. [34]. Two groups of human subjects have been tasked with solving the problem of Hanoi towers (a logic puzzle).

One of the groups was left to learn to solve the problem on their own, while the other group was encouraged to explain the reasons behind their individual moves in the process. That is to say, they were required to express their implicit intuitions in explicit form and to perform high-level reasoning. As a result, this second group was able to learn how to solve the puzzle more quickly.

It is likely that in the future we will see systems, which combine both: implicit and explicit representations, even though the explicit representations may well be implemented using sub-symbolic mechanisms – as is the case with human brains, after all.

3.4.3 Other Ways to Incorporate Prior Knowledge

Apart from using explicit representation, there is, naturally, a number of different ways to incorporate prior knowledge into reinforcement learning. These include methods such as:

- **Model-based reinforcement learning:** Prior knowledge about the task can be used to set up a model of the environment, which the agent can use to do planning or learning, anticipating effect of its actions without actual having to execute them in the real environment.
- **Imitation learning:** Provide instances of correct behaviours and have the agent learn them using supervised learning methods. The resulting agent can then either be used directly, or trained further using reinforcement learning.

  When using RL methods, which represent the policy implicitly, supervised imitation learning may not be a good fit. It may also be the case that we want to mix the demonstrated behaviours with experience in a common replay buffer. In that case the behaviours can be split up into their individual transitions, which can be injected with synthetic rewards so as to incentivize the agent to learn them. We have achieved good results with such approach (in a tabular setting) in the following work:
  - Gregor, M. – Spalek, J. *Log-learning: A Preliminary Study*. In 2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), pp. 235–238. IEEE, 2015. ISBN 978-1-4673-9282-2. URL: <https://doi.org/10.1109/ICUMT.2015.7382434>.
- **Architecture design:** An important way to incorporate prior knowledge about the task is to design a neural architecture that is well-suited to the incoming data. This may involve using 1D and 2D convolutional layers to preprocess sequential data and image data, the use of recurrent layers to provide the agent with memory and so on.
- **Using pre-trained preprocessors:** If it is expected that the agent could benefit from some high-level functionality such as being able to perform visual object detection, SLAM (simultaneous localization and mapping), speech recognition etc., the raw input can be preprocessed using neural architectures pre-trained using supervised learning.

  During reinforcement learning, these architectures may either remain frozen, or they may be fine-tuned. In the latter case, the approach amounts to doing transfer learning on certain sub-modules of the neural network.
- **Partial programming:** Another way in which prior knowledge can be incorporated into a system, is to use a technique known as partial programming, which has been

explored a while ago in papers such as [36–38]. In partial programming, the high-level logic of the system is hard-coded (usually in a specialized language such as ALisp – a version of Lisp augmented with some nondeterministic constructs [36]). This high-level code handles tasks, which can straight-forwardly be solved using standard techniques. The other tasks are delegated to lower-level agents, which learn to perform them using reinforcement learning [36].

Partial programming allows the programmer to incorporate a wealth of prior knowledge into the high-level agent. It does not really require a specialized language – it can, in principle, be done in any language. However, the approach has two notable downsides: It can only be applied to tasks, in which it is possible to divide the system into several clear-cut modules, some of which are written by hand and others are handled by learning agents. The other issues it that the hard-coded structure of the system is fixed: there is no way for the learning algorithm to tune it or modify it based on its actual experience.

- **Transfer reinforcement learning and meta reinforcement learning:** Supervised transfer learning is a well-known technique for training models in settings, where little data is available. The model is first trained on a large dataset, which is not specifically designed for the task, but it is from the same domain (e.g. the ImageNet dataset for image classification). The model uses this dataset to learn how to extract useful features. Afterwards, the model is fine-tuned on a smaller dataset (e.g. an image classification dataset with different labels), to learn the target task. Models trained in this way tend to generalize better than models trained directly on the small dataset.

  The principle of transfer reinforcement learning is similar. The agent is first trained on a source task, where it will hopefully acquire useful skills, which will help it to learn the target task much faster, once it is transferred to it.

  Meta learning is a bit more complex than transfer learning. It essentially adds another level of learning on top of the standard reinforcement learning system. The system uses a set of tasks to learn how to learn efficiently. The meta learner essentially treats all the individual tasks as data points.

  The connection of both – transfer reinforcement learning and meta reinforcement learning – with the incorporation of prior knowledge is very similar. In both cases the prior knowledge that the agent incorporates comes from its own past experience and not from any explicit expert formulation. However, prior human knowledge can still come into the design of the transfer and meta learning tasks themselves, which can make a lot of difference.
- **Natural language interfacing:** Perhaps the easiest way to communicate prior knowledge – such as facts about the environment and orders to the agent – would be

to express them in natural language. However, agents would then have to be trained to understand natural language and form associations between various abstract concepts and states of the environment, which is a very non-trivial task.

Nevertheless, recent research efforts involving both deep reinforcement learning and natural language understanding (such as [39]) do give grounds for cautious optimism – the task may not be as impossible as it would have seemed a few years ago. A further indication that this might be true is the early-stage work presented in [40]: the authors guide policy learning of a reinforcement learning agent using instructions and iterative corrections presented in natural language. The instructions and corrections are grounded using training on a set of tasks, for which ground-truth goals are specified using standard reward functions.

A more complete overview of approaches to prior knowledge incorporation would be out of scope for this present chapter. However, we will briefly mention some early-stage research we did in the area in the past and defer the rest of the discussion to later chapters, addressing the individual approaches as they become relevant.

3.5 | Intelligent HMIs and Natural Language

We have already mentioned natural language as one possible vehicle for conveniently communicating prior knowledge to future machine learning and artificial intelligence systems. However, natural language – together with the ability to recognize gestures, postures, facial expressions and other visual cues – also represents a very natural and convenient way to communicate intentions and instructions to machines.

While in some areas, such as industrial applications, more traditional HMIs are likely to remain in use for the foreseeable future, in other areas, such as personal computers, mobile devices, smart buildings, home robotics, intelligent transport systems and others, intelligent human-machine interfaces are even now being experimented with. As a case in point, there are all the virtual assistant applications and devices, such as Google Assistant, Alexa, Google Home, HomePod and more, which accept instructions in spoken language and reply in kind.

An intelligent voice user interface can have an architecture similar to that shown in Fig. 3.3: The user communicates using spoken language. The speech gets transformed to text using a speech recognition module and gets passed to a domain-specific chatbot system, which interfaces with the machine. It will communicate the appropriate commands to the machine, receive its feedback and transform that back to natural language. The resulting text will be transformed to speech using a speech synthesis module.

Recognizing the importance of intelligent HMIs in the overall context of designing intelligent agents, we have explored the area in several papers and theses, dealing with chatbot

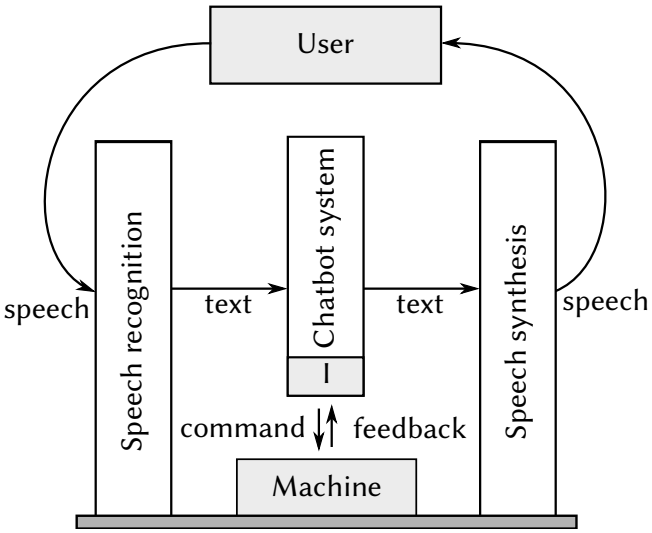


Fig. 3.3: An intelligent voice user interface.

systems:

- Gregor, M. – Gregor, M. *Chatbot systémy v inteligentných rečových užívateľských rozhraniach* [Chatbot Systems in Intelligent Voice User Interfaces]. ProIN, 16(4):50–55, 2015. ISSN 1339-2271;
- Jombík, M. *Chatbot systémy* [Chatbot Systems]. Master's thesis, Žilinská univerzita, 2014. Supervised by Michal Gregor;
- Slováček, L. *Chatbot systémy* [Chatbot Systems]. Bachelor's thesis, Žilinská univerzita, 2016. Supervised by Michal Gregor;

natural language processing and generation:

- Lokaj, M. *Kontrola pravopisu na báze rekurentných neurónových sietí* [Grammar Checking based on Recurrent Neural Networks]. Master's thesis, Žilinská univerzita, 2018. Supervised by Michal Gregor;

speech recognition:

- Gregor, M. *Systém rozpoznávania reči* [A Speech Recognition System]. Bachelor's thesis, Žilinská univerzita, 2009. Supervised by Tomáš Michulek;
- Michulek, T. – Gregor, M. *Remote Control of an Autonomous Mobile 3DLS System using ANN-based Automatic Speech Recognition.* In Metody i techniki zarządzania w inżynierii produkcji. Bielsko-Biała: Wydawnictwo akademii techniczno-humanistycznej w Bielsku-Białej, 2009. ISBN 978-83-60714-64-5;
- Gregor, M. – Michulek, T. *Intelligent Manufacturing Systems – Automatic Speech Recognition System.* In Applied Computer Science, vol. 5. Bielsko-Biała: Wydawnictwo akademii techniczno-humanistycznej w Bielsku-Białej, 2009. ISBN 978-83-60714-95-9. URL: <http://www.acs.pollub.pl/index.php?option=com_content&view=article&id=143:intelligent-manufacturing-systems--automatic-speech-recognition-

system&catid=47:vol-5-no-12009&Itemid=102>;

and intelligent voice user interfaces in a more general sense:

- Gregor, M. – Gregor, M. *Komponenty inteligentných rečových užívateľských rozhraní* [Components of Intelligent Voice User Interfaces]. ProIN, 16(3):48–52, 2015. ISSN 1339-2271;
- Gregor, M. – Gregor, M. *Aplikácie inteligentných rečových užívateľských rozhraní* [Applications of Intelligent Voice User Interfaces]. ProIN, 16(5-6):51–56, 2015. ISSN 1339-2271.

With the increasing amount of work in facial expression recognition, gesture recognition, emotion emulation and more, visual and corporeal aspects of intelligent human-machine interfaces may warrant closer attention as well in the future.

3.6 | Search and Planning

The ability to plan and search for solutions is an important aspect of intelligence. Related artificial intelligence methods have a number of practical applications in scheduling, planning, timetable creation and many other areas. There are also many model-based approaches that integrate search and planning with reinforcement learning.

In our work, we have explored several areas of search and planning, such as adversarial search in games, constraint programming in scheduling, or planning domain languages:

- Vít, M. *Implementácia umelých hráčov pre hru 2048* [Implementation of Artificial Players for 2048]. Bachelor's thesis, Žilinská univerzita, 2018. Supervised by Michal Gregor;
- Gregor, M. – Kovalský, M. – Gregor, T. *Rozvrhovanie I: Programovanie ohraničení* [Scheduling I: Constraint Programming]. ProIN, 18(1):26–30, 2017. ISSN 1339-2271;
- Gregor, M. – Kovalský, M. – Gregor, T. *Rozvrhovanie II: Nástroj MiniZinc* [Scheduling II: MiniZinc]. ProIN, 18(2):55–61, 2017. ISSN 1339-2271.
- Bujňák, D. *Implementácia a riešenie plánovacieho problému* [Implementation and Solution of a Planning Problem]. Master's thesis, Žilinská univerzita, 2016. Supervised by Michal Gregor;

Other researchers at our department have looked into applications in different areas, such as motion planning in mobile robotics – a good example of this would be the dissertation thesis of Dušan Nemec:

- Nemec, D. *Riadenie komplexných robotických mobilných systémov* [Control of Complex Robotic Mobile Systems]. Ph.D. thesis, Žilinská univerzita, 2018. Supervised by Aleš Janota.

3.7 | Teaching and Mentoring

In an effort to increase the involvement of students in artificial intelligence and machine learning research, I have co-taught several artificial intelligence and machine learning related courses, such as:

- Artificial Intelligence 1;
- Artificial Intelligence 2;
- Modelling of Telematic Systems;
- Artificial Intelligence;
- Expert Systems.

I have also worked on new, original content for some of these courses, including new lectures on neural networks and deep learning, evolutionary computation, reinforcement learning. I have also prepared new lab exercises for these and other topics.

In an attempt to provide the students with a more complete picture in an accessible way, I have co-authored several textbooks, such as:

- Gregor, M. *Umelá inteligencia 1* [Artificial Intelligence 1]. Žilina: CEIT, a.s., 2014. ISBN 978-80-971684-1-4;
- Gregor, M. – Nemec, D. – Hruboš, M. – Spalek, J. *Umelá inteligencia 2: Hlboké učenie* [Artificial Intelligence 2]. CEIT, a.s., 2017. ISBN 978-80-89865-03-1;
- Gregor, M. – Hruboš, M. – Nemec, D. *Umelá inteligencia, skriptá I: Návody na vybrané cvičenia* [Artificial Intelligence, Lecture Notes I]. CEIT, a.s., 2017. ISBN 978-80-89865-02-4;
- Gregor, M. – Janota, A. – Hruboš, M. *Kompendium vybraných metód umelej inteligencie*. EDIS: Vydavateľstvo Žilinskej univerzity, 2018. ISBN 978-80-554-1539-0.

I have also participated in a number of other educational and popularization activities as well as in mentoring of individual students – as their thesis advisor and otherwise. It is hoped that all these activities will help us to train highly qualified graduates, who will then be instrumental in driving our work in the field forward.

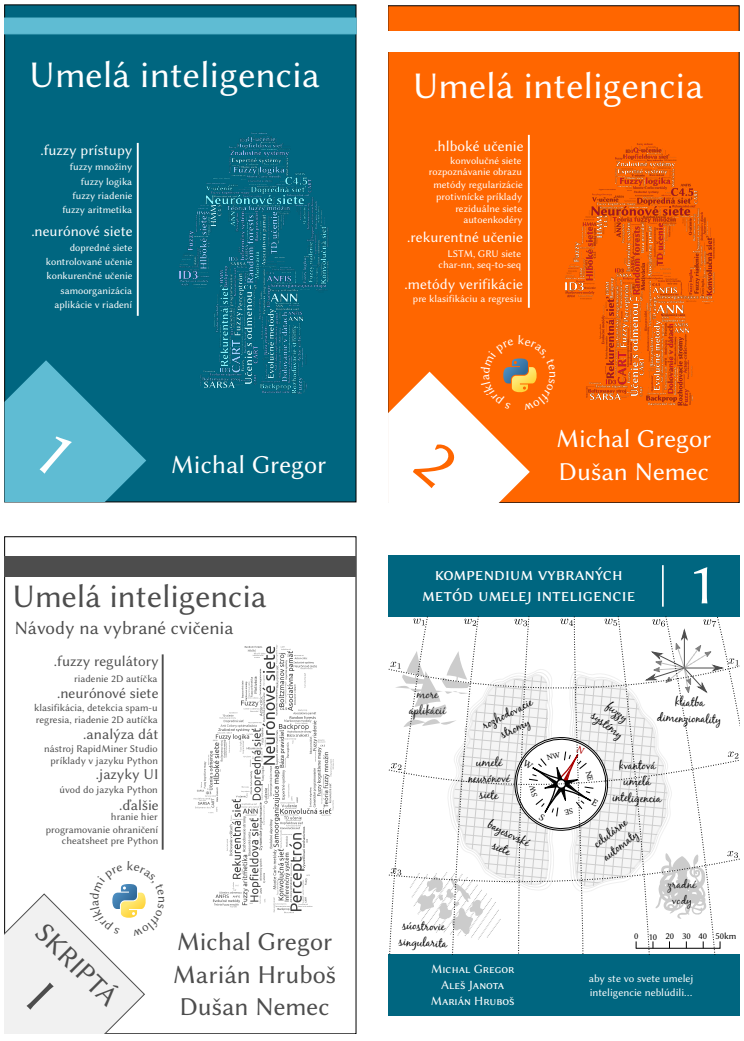


Fig. 3.4: Covers of our textbooks.

CHAPTER 4

THE REINFORCEMENT LEARNING PROBLEM

As we have mentioned hereinbefore, (see 2.1), reinforcement learning (RL) – much like supervised learning – involves a certain kind of feedback. In contrast to supervised learning, where for each input we also know the desired output, in the case of reinforcement learning the feedback has a weaker form. The learning system only receives rewards (usual real numbers), which characterize how well it is doing.

Reinforcement learning is typically applied to sequential decision making problems, i.e. to tasks, where the learning system makes decisions over time. In the context of reinforcement learning, we will refer to the learning system as the *agent*.[2]

An illustration of the principle of reinforcement learning is shown in Fig. 4.1. The figure shows two interacting entities – the agent and the environment. We can see that the agent is able to perceive the state of the environment in some fashion (the arrow labelled "state observation"). It can also influence the state by selecting and carrying out actions (the arrow labelled "action"). Finally, the agent receives certain rewards for its behaviour

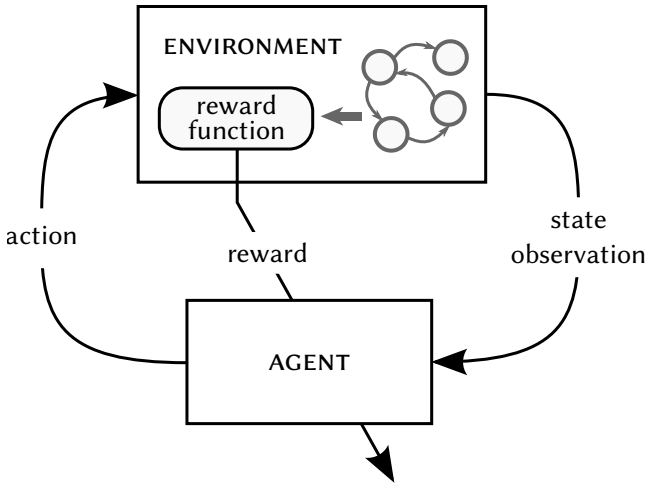


Fig. 4.1: The principle of reinforcement learning.

(arrow labelled "reward"). The goal of the agent is to choose actions that maximize the total rewards received.

It is important to note that maximization of the total rewards cannot in general be achieved by acting greedily – that is to say, by maximizing immediate rewards at every time step. The agent will often have to sacrifice rewards in the short term in order to maximize the total rewards in the long term.

In the remaining part of the chapter, we will try to formalize the concepts introduced above. These introductory pages will be somewhat notation-heavy. They will, however, allow us to lay down what will hopefully be a consistent way of referring to all the essential concepts and properties concerning reinforcement learning problems.

4.1 | Markov Decision Processes

The most common way to formalize reinforcement learning tasks is to describe them as *Markov decision processes* (MDP). MDPs represent a class of probabilistic models related to Markov chains, but augmented with actions – in addition to the environment, there is now an agent, which influences state transitions. The MDP can also be interpreted as a special case of a Bayesian influence network.

The MDP is defined using an n-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{T}, r \rangle$, where:

- $\mathcal{S}$ is the set of states that the environment can be in (the state space);
- $\mathcal{A}$ is the set of actions, which the agent can select from (the action space);
- $\mathcal{I}$ is the initial state distribution;
- $\mathcal{T}$ is the transition function, which determines the probability of a transition to a state, given the current state and the agent's action;
- r is the reward function, which determines what reward the agent will receive for carrying out a certain action in a certain state.

In order for a decision process to qualify as a Markov decision process, it also has to satisfy the *Markov property*. Stated informally, this means that the process must be memory-less. In other words, the next state and reward should only depend on the current state and action – not on any previous states and actions.

Although in general an MDP can be continuous in time, the theory of reinforcement learning focuses almost solely on discrete-time MDPs. A convention of the field is that, even though the time is discrete, it is still denoted with t. The reason we mention this expressly, is that readers familiar with certain branches of system and control theory (in which t is used solely for continuous time and discrete time steps are denoted in a different way), might otherwise find the notation confusing.

More detailed information, including the formal notation for all these MDP components

will be given hereinafter – in separate subsections.

4.1.1 Sets $\mathcal{S}$ and $\mathcal{A}$

The state space $\mathcal{S}$ and the action space $\mathcal{A}$ are both domain-specific. Both can be either discrete or continuous. The choice of a suitable reinforcement learning method may be strongly influenced by the size of these spaces and by some of their other properties. Some methods are, for example, designed to work with discrete action spaces, while others are well-suited to handling continuous actions.

In the following text, states will be denoted with $s \in \mathcal{S}$. The state, that the environment finds itself in at time step t will be denoted with s_t. Analogically to this, actions will be denoted with $a \in \mathcal{A}$, or with a_t whenever we refer to an action at a specific time step t.

4.1.2 The Reward Function

As we have already mentioned, a distinguishing feature of reinforcement learning is that the system receives feedback in the form of a reward signal. The rewards express how well the agent is performing its task. The reward that the agent receives for selecting an action in a particular state, is determined by the *reward function*, which is the following kind of mapping:

$$r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}. \tag{4.1}$$

That is to say, the reward function assigns a real-valued reward $r(s, a)$ to action $a \in \mathcal{A}$ carried out at state $s \in \mathcal{S}$ and provided that the environment transitions to state s_{t+1}. In keeping with our other notation, the immediate reward received at time step t, will be denoted with r_t, i.e. $r_{t+1} =_{\text{def}} r(s_t, a_t, s_{t+1})$.

In general, the reward function may be stochastic. In that case it would be a mapping

$$r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to G(\mathbb{R}). \tag{4.2}$$

to some distribution G over the set of real numbers $\mathbb{R}$.

To save space and make equations more readable, we will also use $\mathcal{R}^a_{ss'}$ as a short-form notation for the expected reward received for performing action a in state s and transitioning to state s', i.e.:

$$\mathcal{R}^a_{ss'} =_{\text{def}} \mathbb{E}\left[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right] = \mathbb{E}\left[r(s, a, s')\right]. \tag{4.3}$$

4.1.3 Trajectories and Histories

By taking actions at discrete time points, the agent triggers certain state transitions. It can, in consequence, be thought of as describing a state-action *trajectory* τ over the spaces $\mathcal{S}, \mathcal{A}$:

$$\tau = (s_0, a_0), (s_1, a_1), ... \tag{4.4}$$

If we want to record the full *history* of the MDP, we may also record the immediate rewards received at every time step. Thus, the history η of the agent can be defined as the sequence

$$\eta = (s_0, a_0, r_0), (s_1, a_1, r_1), ... \tag{4.5}$$

Naturally, if we are given the reward function $r(s_t, a_t)$, it can be used to transform a trajectory τ into a full history η by computing a reward $r_t = r(s_t, a_t)$ for every time step t in τ – always provided that the reward function is deterministic.

4.1.4 The Initial State Distribution

An MDP can start from a number of initial states. The probability of starting from any particular state s is given by the initial state distribution $\mathcal{I}$, which is a mapping of the following kind:

$$\mathcal{I} : \mathcal{S} \rightarrow [0, 1], \tag{4.6}$$

i.e. a mapping from a state to a probability. The actual definition of $\mathcal{I}$ is as follows:

$$\mathcal{I}(s) =_{\text{def}} P(s_0 = s), \tag{4.7}$$

where s_0 is the state at the initial time step $t = 0$.

In many RL tasks, the agent will always start from the same initial state, so there will be exactly one state s for which $\mathcal{I}$ will be 1 and for all other states it will be 0.

4.1.5 The Transition Function

Since the MDP will, in the general case, contain stochastic elements, the transition function cannot map the current state s_t and action a_t directly to the next state s_{t+1}. Instead of that, it determines the probabilities of state transitions. Its form is therefore as follows:

$$\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]. \tag{4.8}$$

The actual definition of the transition function in terms of transition probabilities, can be written as follows:

$$\mathcal{T}(s, a, s') =_{\text{def}} P(s_{t+1} = s' | s_t = s, a_t = a). \tag{4.9}$$

That is to say that $\mathcal{T}(s, a, s')$ expresses the probability of the transition from state s to state s' when action a is applied. Note that the transition probabilities only depend on states and actions – there is no direct dependence on the rewards.

In order to make the notation more compact in complicated expressions, we will some-

times use $P^a_{ss'}$ to refer to the probability of transitioning from state s to state s' when action a is applied, i.e.:

$$P^a_{ss'} =_{\text{def}} \mathcal{T}(s, a, s'). \tag{4.10}$$

If an MDP is purely deterministic and does not contain any stochastic elements, the following stricter formulation will also hold:

$$\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \{0, 1\}, \tag{4.11}$$

because the state transition will either happen with certainty, or not happen at all.

4.1.6 The Markov Property

In order for the above-stated definition of the transition function to make sense, the corresponding decision process must be *Markovian* – i.e. it must satisfy the *Markov property*. Informally speaking, this means that the process must be memory-less. The next state and reward should only depend on the current state and action – not on any previous states or actions.

A formal statement of the same, for a discrete-time MDP, can be done in the following way:

$$\begin{aligned} &P(s_{t+1} = s, r_{t+1} = r | s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = \\ &= P(s_{t+1} = s, r_{t+1} = r | s_t, a_t), \end{aligned} \tag{4.12}$$

where s_t denotes the state at time step t and r_t denotes the reward at time step t.

If the Markov condition did not hold, we would have to condition the transition probabilities on the entire trajectory $\tau_{\leq t}$ (up to and including time step t):

$$\mathcal{T}(\tau_{\leq t}, a, s') =_{\text{def}} P(s_{t+1} = s' | \tau_{\leq t}). \tag{4.13}$$

4.2 | Environment Models

When selecting an appropriate reinforcement learning method, one of the things we need to consider is whether a model of the environment is available or not. There are two distinct types of reinforcement learning methods:

- *model-based:* they require that a model be available and provided that it is, they can use it to some advantage;
- *model-free:* they do not require a model.

In practice, we want to have both kinds of methods at our disposal. If an accurate model of the environment is available, it can make solving a task much easier. The agent can anticipate effects of its actions without actually having to execute them. It can solve its task by planning, by learning from simulated experience, or in any number of other, more

complex ways. This is why model-based RL methods are useful.

On the other hand, there are tasks for which no model is available and to create it would be comparably or even more difficult than to solve the original task in the first place. It is therefore also useful to have efficient model-free methods at our disposal.

4.2.1 Sample and Distribution Models

It is important to note that there are two significantly different kinds of environment models, i.e. [59]:

- *Distribution models:* To know a distribution model is essentially equivalent to knowing the transition function of the decision process. For every state of the environment (and an action), a distribution model will specify the full distribution over the next state: i.e. all possible next states together with their respective probabilities.
- *Sample models:* Sample models are simpler and less expressive. They do not describe the full distribution – they can only draw samples from it.

A distribution model can easily be used in place of sample models, but the reverse is not true: we can draw samples from a distribution model, but we cannot easily reconstruct the full distribution from samples. This shows that distribution models are the more powerful class. However, it is often much more difficult to get a distribution model than to get a sample model [59].

4.3 | Policies

The problem of solving an MDP can be framed as the problem of searching for the optimal policy. The policy is the component of an RL agent, which decides, what action to choose, given some perception of the current state. The policy π may be either deterministic, or stochastic, so the general form for the kind of mapping it performs would be:

$$\pi : \mathcal{S} \to F(\mathcal{A}), \tag{4.14}$$

where $F(\mathcal{A})$ is some probability distribution over the action space $\mathcal{A}$.

To simplify the notation, one can use the following to indicate the probability that action a will be selected in state s when following policy π:

$$\pi(s, a) =_{\text{def}} p(a \mid s, \pi). \tag{4.15}$$

4.4 | Episodic and Continuing Tasks

Some RL tasks can continue to run indefinitely –we call these *continuing* tasks –, while others will tend to auto-terminate at some point – these are referred to as *episodic* tasks.

Episodic tasks usually involve some kind of a *goal state*, or – more generally – a *termi-*

nation condition. In certain cases, episodes may need to be terminated forcefully, instead of waiting for them to terminate naturally. The most common termination criterion would be to specify the maximum number of time steps that an episode can take – but one can naturally set up other, more complex criteria.

As far as continuing problems are considered, an agent that is supposed to collect and remove garbage from a street can be given as an instance. Presumably any people, that live in the area, will generate more garbage over time. The task of the agent is therefore ongoing – it is supposed to go on collecting and removing garbage indefinitely, as soon as new garbage accumulates.

Now, having discussed the concept of episodes and steps, we can describe the sequential character of reinforcement learning using pseudocode Algorithm 1. The three underlined steps in the algorithm correspond to what the agent does: *(a)* observing the environment; *(b)* selecting an action in accordance with the current policy and applying the selected action; *(c)* updating the policy, the value estimates, etc., once the immediate reward has been received. The rest depends on how the environment has been set up.

Algorithm 1: The sequential character of reinforcement learning.

```
1  Initialization;
2  repeat (for each episode)
3      Initialization of a new episode;
4      repeat (for each step)
5          Observing the environment;
6          Selection and application of an action;
7          State transition;
8          Issuing a reward;
9          Policy improvement, ...
10     until the termination condition is satisfied;
11     The end of the episode;
12 until the maximum number of episodes has been reached;
```

4.5 | Immediate Rewards and Total Returns

We have already stated hereinbefore that the goal of reinforcement learning is to optimize the total return of the agent in a long-term sense (the long-term rewards), not just the immediate reward at the next time step. But how are these long-term returns to be defined formally?

We already have notation for immediate reward r_t at time step t. The total return following time step t and onwards will be denoted with R_t. There are several ways to define this total return:

- the infinite-horizon model;
- the finite-horizon model;
- the discounted infinite-horizon model;
- the average reward model.

We will now go through these in order.

4.5.1 The Infinite-Horizon Model

When computing the total return, we may sum up the rewards at all the time steps following t, which will essentially yield a sum that goes on to infinity – hence the name infinite-horizon model:

$$R_t^{\text{inf}} =_{\text{def}} r_{t+1} + r_{t+2} + r_{t+3} + ... = \sum_{k=t}^{\infty} r_k. \tag{4.16}$$

This kind of model does not easily apply to tasks in general, but there are applications, in which it can work well. If, for an instance, the task is naturally episodic, there will actually be some final time step T and the sum will not go to infinity, yielding the following return formulation instead [59]:

$$R_t^{\text{inf}} =_{\text{def}} r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T = \sum_{k=t}^{T} r_k. \tag{4.17}$$

There may also be specific RL problems, where by the nature of the reward signal, the infinite series will have a finite sum. However, this will, of course, not be true in general.

4.5.2 The Finite-Horizon Model

The finite horizon model, as its name implies, only looks at a finite horizon. It only considers rewards, which are finitely close in the future: closer than a specified horizon h. The definition of the total return will then be as follows [59]:

$$R_t^{\text{fin}} =_{\text{def}} r_{t+1} + r_{t+2} + ... + r_{t+h} = \sum_{k=t}^{t+h} r_k, \tag{4.18}$$

where $h \in (0, \infty)$.

Even though the finite-horizon model eliminates some of the issues connected to the infinite-horizon model, it introduces problems of its own. One significant issue is how to choose the horizon h. Obviously we need it to be long enough to at least accommodate delayed rewards. Otherwise the agent will, for example, not be able to learn complex behaviours, which may take more than h steps to execute or at least to get rewarded for.

4.5.3 The Discounted Infinite-Horizon Model

The problems connected with selecting the optimal horizon h seem to suggest that rather than cutting off rewards further than h time steps in the future discretely, it might be a better idea to instead decrease the weight of the rewards softly, so that the further the

reward is in the future, the less importance is attached to it. In this way, the influence of any rewards that are very far in the future will be negligible, but there will be no discrete cut-off point and thus no need to select h.

The discounted infinite-horizon model builds upon these ideas and formalizes them in the following way [59]:

$$R_t^{\text{dis}} =_{\text{def}} r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1}, \tag{4.19}$$

where $\gamma \in [0, 1]$ is the discount factor, which determines the rate at which future rewards are discounted.

The intuition behind the discounted infinite-horizon model is actually concisely expressed by the saying "A bird in the hand is worth two in the bush". The rewards that the agent gets in the near future should be assigned a greater weight simply because we are more certain that the agent will indeed be able to receive them. Conversely, rewards very far in the future tend to be much less certain. Over a large number of time steps, many unforeseen things may happen (such as changes in the environment, the agent getting damaged, etc.) that might prevent the agent from accruing the rewards.

In the following text, we will actually assume that the discounted infinite-horizon model is being used, unless we explicitly state otherwise.

4.5.4 The Average Reward Model

For tasks, where it does not make sense to discount future rewards, it may be more sensible to use the average reward model instead:

$$R_t^{\text{avg}} =_{\text{def}} \lim_{h \to \infty} \frac{1}{h} \sum_{k=0}^{t+h} r_{t+k+1}. \tag{4.20}$$

4.6 | The Goal of Reinforcement Learning

The goal of reinforcement learning, as we have mentioned before, is to find a policy for the agent, which maximizes the total rewards received by the agent in the long term. In general, then, the RL goal can be expressed as finding the optimal vector of parameters θ^* for a parametrized policy π_θ, so that some criterion $J(\theta)$ is maximized:

$$\theta^* = \arg\max_{\theta} J(\theta). \tag{4.21}$$

The criterion will generally involve the *expected value* of the total return (since both the MDP and the policy can be stochastic, the actual return may vary considerably among multiple runs of the same algorithm). Given that there are several different ways to define the total return, there will naturally also be several ways to define criterion $J(\theta)$.

4.6.1 The Discounted Infinite-Horizon Formulation

When using the discounted infinite-horizon return R_0^{dis}, RL goal can be formalized as maximizing a criterion based on the return from time step $t = 0$ onwards [60, 61]:

$$\begin{aligned} J^{\text{dis}}(\theta) &=_{\text{def}} \mathbb{E}\left\{R_0^{\text{dis}} \middle| \pi_\theta\right\} \\ &= \mathbb{E}\left\{\sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| \pi_\theta\right\}. \end{aligned} \tag{4.22}$$

The same criterion can also be expressed as an expected value over the state-action space [61]:

$$\begin{aligned} J^{\text{dis}}(\theta) &=_{\text{def}} \sum_{s\in\mathcal{S}} \rho^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\mathcal{R}_s^a \\ &= \mathbb{E}_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta}\left\{r(s,a)\right\}, \end{aligned} \tag{4.23}$$

where $\rho^\pi(s)$ is the (improper) discounted state distribution, which expresses how likely the agent is to land in state s and with how much discounting [61]:

$$\rho^\pi(s') =_{\text{def}} \sum_{s\in\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{I}(s) p(s \to s', t, \pi), \tag{4.24}$$

Expression $p(s \to s', t, \pi)$ denotes the density at state s' after transitioning for t time steps from state s. Also recall that $\mathcal{I}(s)$ is the initial state distribution.

4.6.2 The Average Reward Formulation

For the case of a continuing problem, the average reward formulation may be preferable. The criterion has the following form [60, 62]:

$$\begin{aligned} J^{\text{avg}}(\theta) &=_{\text{def}} \mathbb{E}\left\{R_0^{\text{avg}} | \pi_\theta\right\} \\ &= \lim_{h\to\infty} \frac{1}{h} \mathbb{E}\left\{r_1 + r_2 + \ldots + r_h | \pi_\theta\right\}. \end{aligned} \tag{4.25}$$

This can again be expressed as an expectation over the state-action space [60]:

$$J^{\text{avg}}(\theta) =_{\text{def}} \sum_{s\in\mathcal{S}} \rho^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\mathcal{R}_s^a, \tag{4.26}$$

where $\rho^{\pi_\theta}(s)$ is now the stationary distribution of states under π [62]:

$$\rho^\pi(s) =_{\text{def}} \lim_{t\to\infty} P(s_t = s | \pi), \tag{4.27}$$

which is supposed to exist for every policy π and to be independent of the initial state s_0.

Naturally, all the equations with sums over the state/action space only hold for the case when that space is discrete. If one or both of the spaces are in fact continuous, the corresponding sum will be replaced with an integral.

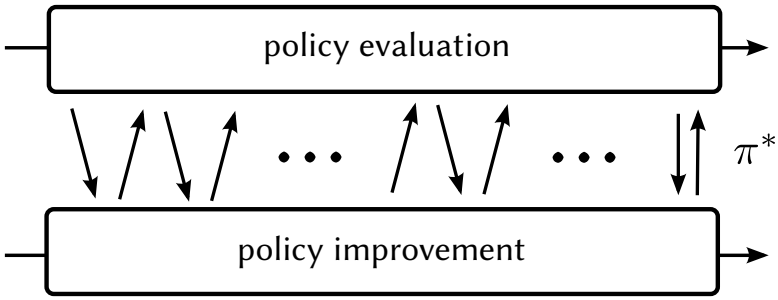


Fig. 4.2: Generalized policy iteration.

4.7 | Reinforcement Learning Methods

This introductory chapter on RL has described the reinforcement learning problem, as it is commonly stated using the MDP framework. We have introduced the necessary notation and shown several ways to formalize the goal of maximizing total returns. We are now ready to introduce the common principles behind reinforcement learning methods and to list several distinct types of them.

Virtually all reinforcement learning methods are well described using a framework known as *generalized policy iteration*, the concept of which is illustrated in Fig. 4.2. In this framework, there are two mutually interacting processes:

- **Policy evaluation:** determines how good certain states / actions / behaviours are in the long term;
- **Policy improvement:** uses the information from policy evaluation to make rewarding behaviours more likely under the policy.

Policy evaluation is done in order to get the information necessary to improve the policy. Once the policy has been improved, it needs to be re-evaluated, since its behaviour will have changed. The re-evaluation will yield new insights on how to improve the policy further and so on. Normally, when the process converges to the optimal policy, evaluation will no longer yield any useful information and the policy will stop changing.

The policy evaluation and policy iteration steps may be implemented in a number of ways – ranging from very simple to extremely complex. Depending on way they are implemented, we differentiate the following 3 types of reinforcement learning methods:

- **Value-based methods:** Methods based on the use of value functions, i.e. of functions that estimate the long-term effects of being in a state or executing an action. These methods do not represent the policy explicitly, but rather derive it from the value function: e.g. by always picking the action with the greatest expected total return. Naturally, this approach is not practical when the action space is very large or continuous.
- **Policy-based methods:** Methods that represent the policy explicitly (e.g. by a neural

network), which makes them applicable to continuous action spaces and allows them to represent stochastic policies in a natural way. They use relatively simple ways to estimate long-term rewards associated with behaviours – often just by observing a single sample return. For this reason, they tend to suffer from issues with high variance.

- **Actor-critic methods:** Methods, which combine both – explicit policy representation and a value function. They combine the advantages of value-based and policy-based methods: like value-based methods, they tend to have much lower variance and like policy-based methods they are good at representing stochastic policies and they can handle continuous action spaces.

CHAPTER 5

VALUE-BASED RL METHODS

We will start our discussion of reinforcement learning methods with value-based methods. We will first define what value functions are and then proceed to derive several value-based algorithms.

Note that some concepts presented in this chapter, such as the exploration vs. exploitation trade-off, apply to all three: value-based, policy-based, and actor-critic methods.

5.1 | Value Functions

As we have already mentioned, in a typical RL task, the agent has to go beyond greedily maximizing immediate rewards, if it is to maximize its total return. This is mainly because:

- Rewards may be delayed – they may occur several time steps after the relevant action has been performed;
- Rewards usually do not relate to any single action, but rather to some more complex behaviour – a sequence of many actions.

If we had a function, which – instead of immediate rewards – would map directly to the total expected returns, that would make the task of an RL agent much easier. This is because actions greedily maximizing this new function would actually be optimal in the long term.

Functions of this kind are called *value functions*. There are two different kinds:

- the *state-value function* $V^\pi : \mathcal{S} \to \mathbb{R}$;
- the *action-value function* $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

The definitions we give below (including section 5.2 on Bellman equations) are written under the assumption that we are using the discounted infinite-horizon model. In the case of the average reward model, differential value functions are used instead. However, to prevent confusion, we will not cover them at this point. The reader may refer to [62] for a

more detailed introduction.

5.1.1 The State-Value Function

The state-value function $V^\pi(s)$ maps a state s to its value – i.e. to the total return that the agent can expect to obtain in the future, after it has been in state s, provided that it follows policy π:

$$V^\pi(s) = \mathbb{E}_\pi\{R_t | s_t = s\}. \tag{5.1}$$

That the state-value function should always depend on a particular policy π should be obvious: the total return of the agent will generally depend on how it is going to behave.

If we assume the discounted infinite-horizon model, we can also write [59]:

$$V^\pi(s) = \mathbb{E}_\pi\{R_t | s_t = s\} = \mathbb{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s\right\}, \tag{5.2}$$

5.1.2 The Action-Value Function

The action-value function $Q^\pi(s, a)$ (also frequently called the Q-function due to the notation) returns the total return that the agent can expect in the future, in consequence of choosing action a in state s, provided that it follows policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi\left\{R_t \,|\, s_t = s, a_t = a\right\}. \tag{5.3}$$

When we plug in the discounted infinite-horizon model, we get [59]:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi\left\{R_t \,|\, s_t = s, a_t = a\right\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a\right\}. \end{aligned} \tag{5.4}$$

The reader should note that once we have an action-value function, it is very easy (at least in the discrete-action case) to recover a policy from it – the agent can just greedily select the action with the highest Q-value. This will correspond to selecting the action with the highest expected return. In the case of continuous actions the situation is not quite as straight-forward and different approaches must be taken (see chapter 6 for more details).

5.1.3 Value Function Intuitions

In order to make the above-defined concepts more tangible, we will now explore the notion of value functions in a simple grid-world setting. The map of the grid-world is shown in Fig. 5.1. The dark-grey cells correspond to walls, which the agent cannot traverse. Symbol P at position $(0, 9)$ indicates the initial state of the agent. Symbol G marks the goal state.

The agent has 4 distinct actions at its disposal: to go *(a)* up, *(b)* down, *(c)* left, *(d)* right. The agent is supposed to pick a sequence of actions, which will get it to the goal state G,

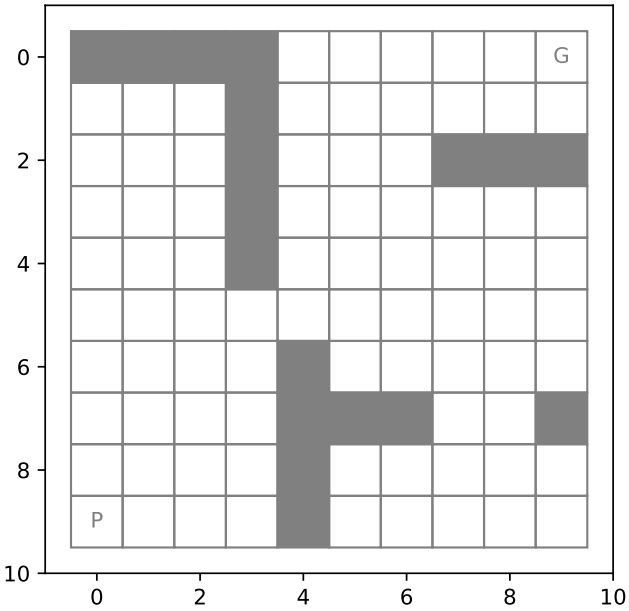


Fig. 5.1: Map of the grid-world.

in which it will receive reward of $r_G = 100$. In all other states, the agent will receive the reward of 0.

State-Value Function Intuitions

Let us say that we want to evaluate, how useful it is for the agent to be in each of the available states $s \in \mathcal{S}$, which in our case correspond to the various positions within our grid-world. The value of the goal state should – perhaps a bit counter-intuitively – be 0. This is because the episode ends there are the agent receives no subsequent rewards thereafter.

The values of the other states will very much depend on the model of total returns that we are using. If we use the discounted infinite-horizon model, the agent will assign greater value to states close to the goal state. This is because the further the agent is from the goal state, the more time steps it will take to reach it and – consequently – the more discounted the reward will become. For a state, which is, say, 5 steps from the goal, the reward will be discounted just 5 times, whereas for a state 10 steps away it will be discounted 10 times.

The state-value function, as computed using value iteration (for information on value iteration, see section 5.5.2) with the discount factor of $\gamma = 0.9$ and 50 backups, is shown in Fig. 5.2. It can be clearly seen that the values of the states decrease as one moves further from the goal state. Note that the goal state's value is zero as we expected and that the value of states removed by 1 step is $\gamma \cdot 100 = 90$.

As we can see, even though we do not explicitly punish the agent for loitering, it will still learn to move towards the goal state because of the discounting. If we switch to the undiscounted version of the infinite-horizon model (i.e. set $\gamma = 1$), the situation changes.

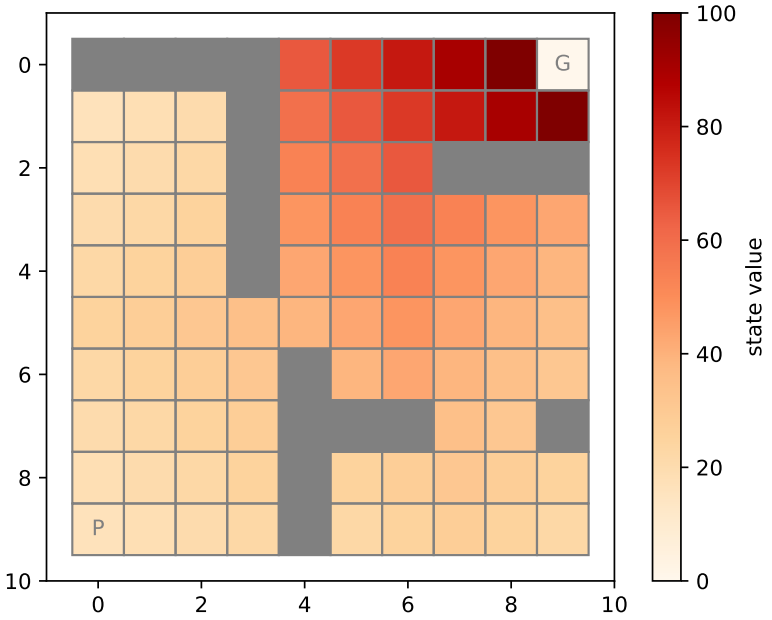


Fig. 5.2: The grid world's state-value function ($\gamma = 0.9$).

Since the episode does not end until the agent reaches the goal state and receiving a reward a 1000 time steps in the future is just as good as receiving it immediately, there will be no incentive for the agent to actually go anywhere in any finite horizon. That situation is depicted in Fig. 5.3 – as we can see, all states have the same value, which is 100.

Given a state-value function and the MDP's transition function, we can already select actions. If the agent goes over all the actions and all possible next states (1 step lookahead), it can compute which action has the greatest expected return. This principle will be stated formally later on. In our grid-world example, we can do this graphically – by always moving to the darkest square we can reach (ties can be broken arbitrarily). In our case, this would lead to the path depicted in Fig. 5.4.

Naturally, if we try to use the undiscounted state-value function to select actions, things will break down. In our case there is no incentive for the agent to move towards the goal state in any finite number of steps – the agent will just wander off anywhere as shown in Fig. 5.5 (the exact direction only depends on how we break ties). In our case, even though the agent would theoretically be guaranteed to receive the reward once the episode ends, the episode would actually never end, thereby rendering the guarantee useless.

Action-Value Function Intuitions

As we have already mentioned, the action-value function returns how much reward an agent can expect in the future for selecting action a in state s. As such, it can easily be turned into a policy (at least if the action space is discrete and not too large) – an agent merely needs to pick the action with the greatest value in any given state.

Although we made a similar statement about the state-value function, which too can be

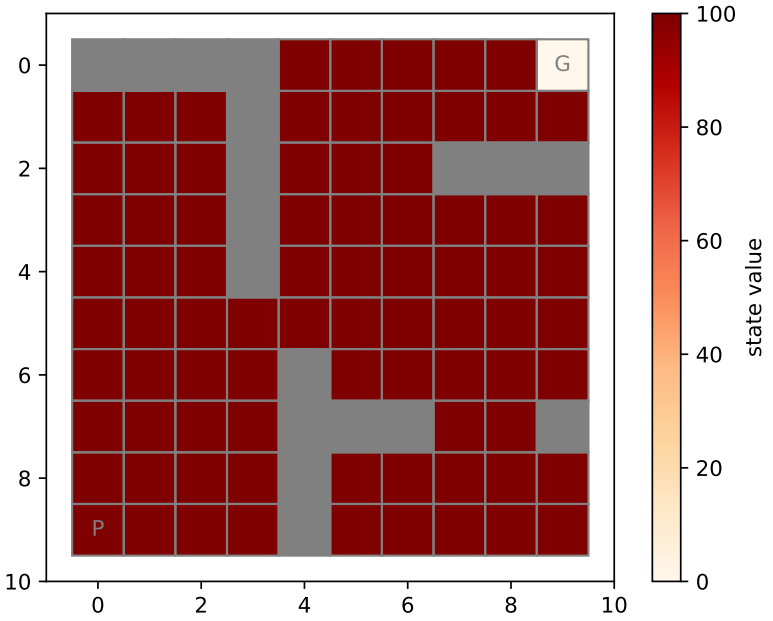


Fig. 5.3: The state-value function for the undiscounted model (i.e. $\gamma = 1$).

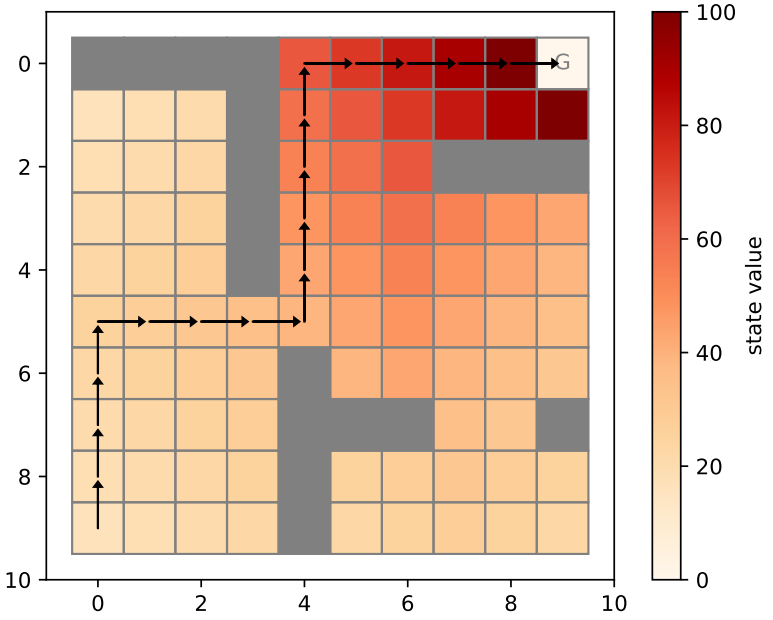


Fig. 5.4: The path that the agent will take, if following the value function for $\gamma = 0.9$.

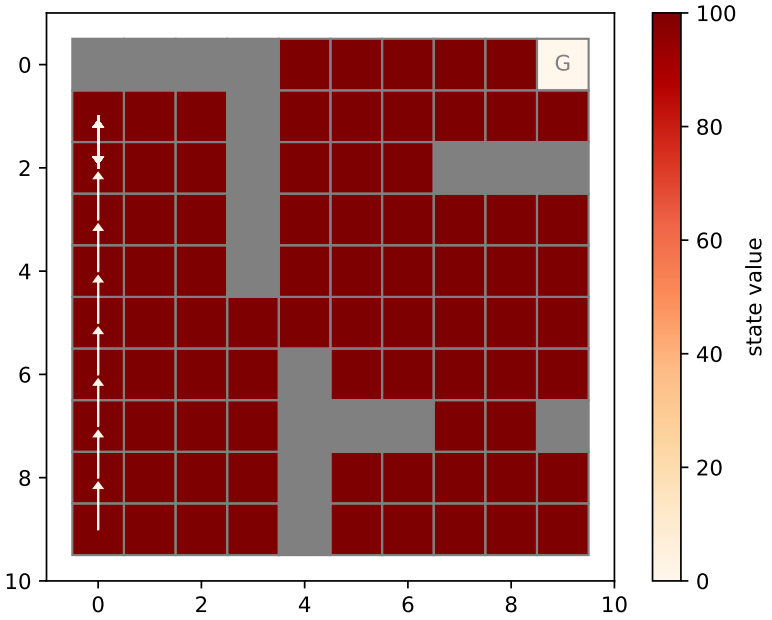


Fig. 5.5: The path that the agent will take, if following the value function for $\gamma = 1$.

used to derive a policy, note that in the case of the action-value function, we do not require the knowledge of the transition function. We also do not have to consider all possible next states. The action-value function already factors all that context in.

The action-value function for our grid-world example is depicted in Fig. 5.6. For every state, we indicate the relative values of all four actions: larger and darker arrows correspond to actions with greater values.

Finally, provided that we use the undiscounted infinite-horizon model, all actions will have equal values – this case is illustrated in Fig. 5.7.

The Expected Value Depends on the Policy

At this point, let us reiterate that the expected values will almost always depend on the policy that the agent chooses to follow. In the case of our grid-world example, if the agent chooses to stay in place, it will never receive any rewards. If it instead walks towards the goal and reaches it, it receives the reward of 100. This is why a value function is always computed with respect to some policy (although that policy may be implicit – for an instance, see section 5.5.2 on value iteration).

5.2 | The Bellman Equations

One very useful and important property of the value functions is that they are recursive. This is mainly because the return can generally be decomposed into two parts – the immediate reward and the rest of the return. With the discounted infinite-horizon model this

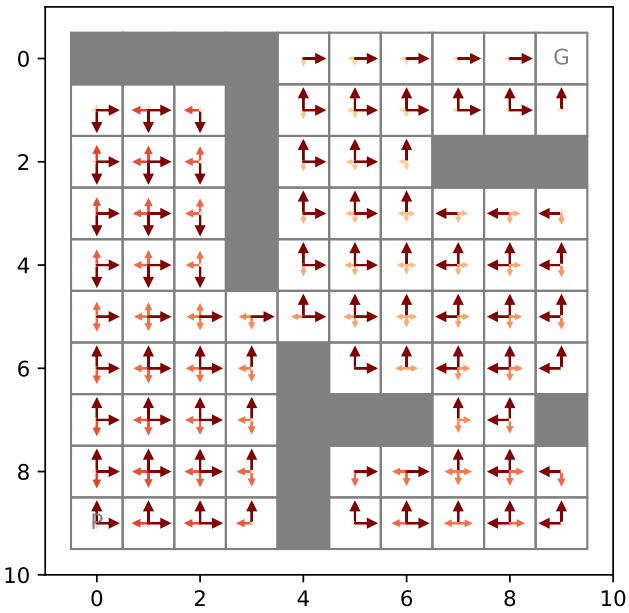


Fig. 5.6: The action-value function for $\gamma = 0.9$. The larger, darker arrows correspond to actions with greater values.

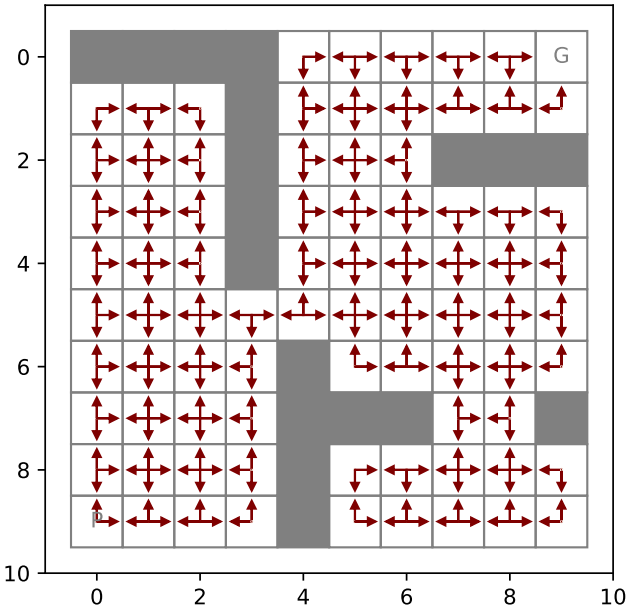


Fig. 5.7: The action-value function for $\gamma = 1$. The larger, darker arrows correspond to actions with greater values.

means the following:

$$\begin{aligned} R_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \\ &= \gamma^0 r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+1+k} \\ &= r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k}, \end{aligned} \tag{5.5}$$

which can, in turn, be rewritten as:

$$R_t = r_{t+1} + \gamma R_{t+1}. \tag{5.6}$$

This enables the formulation of the so-called Bellman equations, which express how the value at the current state relates to values at all possible next states. In the remaining part of this section, we will go over these relationships and show how the Bellman equations arise.

5.2.1 The Bellman Expectation Equation

Let us first recall that the state-value function is defined as

$$V^{\pi}(s) = \mathbb{E}_{\pi}\{R_t | s_t = s\} \tag{5.7}$$

and that we can plug in the discounted infinite-horizon model to get

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s \right\}. \tag{5.8}$$

If we now decompose the return according to (5.5), we get

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s \right\} \\ &= \mathbb{E}_{\pi}\left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_t = s \right\}. \end{aligned} \tag{5.9}$$

In order to pull r_{t+1} out of the expectation, we substitute in the action probabilities $\pi(s,a)$ and the transition probabilities $P^a_{ss'}$ [59]:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_{t+1} = s' \right\} \right], \tag{5.10}$$

where $\mathcal{R}^a_{ss'} =_{\text{def}} \mathbb{E}\left[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right]$ is the expected value of reward r_{t+1} provided that the agent takes action a in state s and transitions to state s'. Naturally, this notation is only correct provided that both the action space $\mathcal{A}$ and the state space $\mathcal{S}$ are discrete – otherwise the sums over $\mathcal{A}$ and $\mathcal{S}$ would need to be replaced by integrals.

The interesting thing here is that the expected value that we are left with is actually

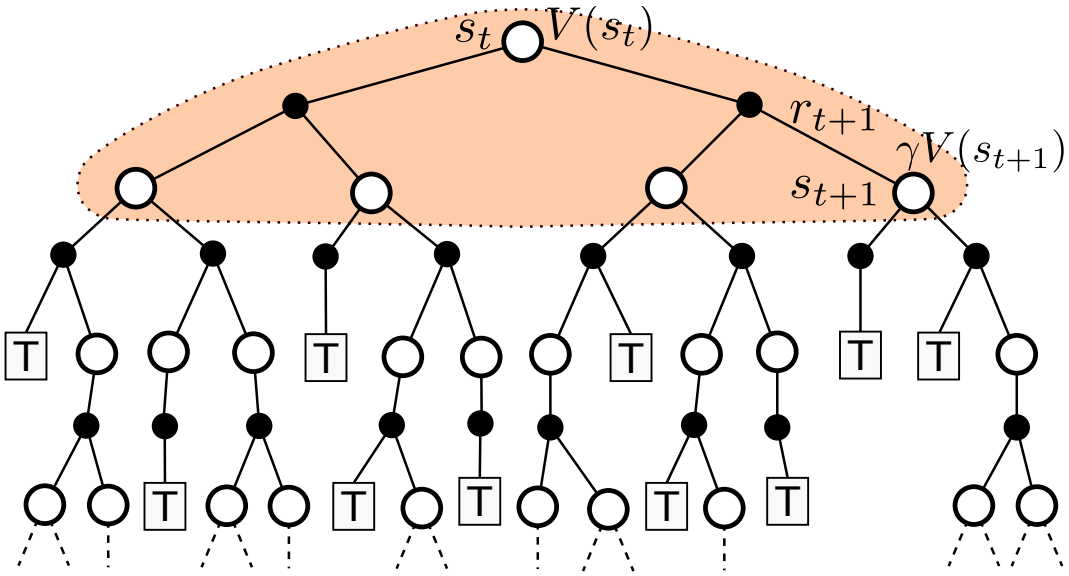


Fig. 5.8: A schematic illustration of the Bellman backup, resulting from the Bellman expectation equation. The value of state s_t can be computed by considering the values of all transitions from it, weighted by their respective probabilities. (Figure inspired by [63].)

exactly equal to the state-value $V^\pi(s')$ of state $s_{t+1} = s'$, i.e.:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right]. \tag{5.11}$$

This relationship is known as the *Bellman expectation equation.* To summarize the entire procedure once again, we can write [59]:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left\{ R_t \,|\, s_t = s \right\} \\ &= \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s \right\} \\ &= \mathbb{E}_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_t = s \right\} \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_{t+1} = s' \right\} \right] \\ &= \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right]. \end{aligned} \tag{5.12}$$

A Bellman Backup Diagram

A simple way to illustrate the Bellman expectation equation is shown in Fig. 5.8. It shows how the value $V(s_t)$ of state s_t can be computed by considering all possible transitions from it. The corresponding immediate reward r_{t+1} and discounted value $\gamma V(s_{t+1})$ are added up and multiplied by the probability of the transition. The results are then summed up to form $V(s_t)$.

Note that the larger, empty circles correspond to the agent's decisions – there would be one branch for each action that the agent can take. The smaller, black circles correspond to the dynamics of the environment – each branch corresponds to one possible outcome of taking one particular action. The rectangles labelled with "T" correspond to terminal states.

In Terms of the Action-Value Function

An analogical Bellman equation can be derived for the action-value function Q^π:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi \left\{ R_t \,|\, s_t = s, a_t = a \right\} \\ &= \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a \right\} \\ &= \mathbb{E}_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_t = s, a_t = a \right\} \\ &= \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \sum_{a' \in \mathcal{A}} \pi(s', a') \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_{t+1} = s', a_{t+1} = a' \right\} \right] \\ &= \sum_{s' \in \mathcal{S}} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \sum_{a' \in \mathcal{A}} \pi(s', a') Q^\pi(s', a') \right]. \end{aligned} \tag{5.13}$$

5.2.2 The Optimal Policy

As we have already stated, the goal of reinforcement learning is to maximize long-term rewards of the agent. We can now reformulate this goal in a different way – by introducing the concept of the optimal policy. We can then state what the Bellman equations say about that optimal policy, which will give us some indications of how to find it.

Policy Ordering

We say that policy π is better than or equal to π', if its expected rewards are greater than or equal to the expected rewards of π' [59]. That is to say, $\pi \geq \pi'$ holds if and only if [59]:

$$V^\pi(s) \;\geq\; V^{\pi'}(s) \; \forall s \in \mathcal{S}. \tag{5.14}$$

The Optimal Policy: The Formal Definition

The policy, which is better than or equal to all other policies, is called the optimal policy and denoted π^*, i.e. the following holds for its state-value function [59]:

$$V^{\pi^*}(s) = \max_\pi V^\pi(s) \; \forall s \in \mathcal{S}. \tag{5.15}$$

The value function of the optimal policy is also referred to as the optimal value function, denoted $V^*(s) =_{\text{def}} V^{\pi^*}(s)$.

The same concept can be equivalently expressed using the action-value function [59]:

$$Q^{\pi^*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \; \forall s \in \mathcal{S} \wedge a \in \mathcal{A}, \tag{5.16}$$

where the same notation applies: $Q^*(s,a) =_{\text{def}} Q^{\pi^*}(s,a)$.

Note that if the problem is Markovian (refer back to section 4.1.6 for the precise formulation of the Markov condition), there will always be a deterministic policy, which will satisfy the above-stated optimality conditions [62]. In non-Markovian cases, however, the optimal policy may have to be stochastic.

5.2.3 The Bellman Optimality Equation

If we now substitute (5.15) for $V^{\pi}(s)$ in the Bellman expectation equation (5.12), we obtain an equation that describes the optimal policy – the so-called Bellman optimality equation [59]:

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}} Q^*(s,a) \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} \left\{ R_t \,|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi^*} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+2+k} \,\middle|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}} \mathbb{E} \left\{ r_{t+1} + \gamma V^*(s_{t+1}) \,|\, s_t = s, a_t = a \right\} \\
&= \max_{a \in \mathcal{A}} \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s') \right].
\end{aligned} \tag{5.17}$$

Note that at one point, the expectation $\mathbb{E}_{\pi^*}$ becomes $\mathbb{E}$. This is because the optimal value function V^* has been plugged in, and thus the expression no longer depends on the policy π^* directly. The expectation is then only over the transition model: i.e. what the next state s_{t+1} and reward r_{t+1} will be.

By analogical reasoning, one can derive the following for the optimal action-value function [59]:

$$\begin{aligned}
Q^*(s,a) &= \mathbb{E} \left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \,\middle|\, s_t = s, a_t = a \right\} \\
&= \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right].
\end{aligned} \tag{5.18}$$

We will draw on these definitions in a later section, when discussing various ways to learn the optimal policy.

5.3 | The Greedy Policy

Once we have a value function, we can easily turn it into a policy (at least provided that the states space and the action space are discrete and relatively small). One sensible way to do this would be to examine the values of all the actions at every state and select the action with the maximum value.

A policy, which selects actions according to this procedure is called the greedy policy. Formally, we can express this by writing:

$$\pi_{greedy(Q)}(s) =_{\text{def}} \arg\max_{a \in \mathcal{A}} Q(s,a). \tag{5.19}$$

where $greedy(Q)$ denotes the policy being greedy w.r.t. a value function Q.

In a similar way, a state-value function can also be used to derive a policy – except that in this case, one needs to do 1-step lookahead for which the knowledge of the environment's transition function is required:

$$\pi_{greedy(V,\mathcal{T})}(s) =_{\text{def}} \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]. \tag{5.20}$$

5.3.1 The Policy Greedy w.r.t. the Optimal Value Function

A policy, which is greedy w.r.t. the optimal value function, will be optimal in its turn, i.e. we can write that:

$$\pi^*(s) = \pi_{greedy(Q^*)}(s), \tag{5.21}$$

or using the state-value function:

$$\pi^*(s) = \pi_{greedy(V^*,\mathcal{T})}(s). \tag{5.22}$$

Note though, that these equations are only guaranteed to hold in the Markovian case. In non-Markovian problems the optimal policy may have to be stochastic.

5.4 | Exploration vs. Exploitation

We have described, how a greedy policy can be derived from a value function and we have stated that a policy greedy w.r.t. the optimal value function will be the optimal policy, provided that the problem is Markovian.

However, the agent is generally not given the optimal value function – it is its task to learn it. The process will typically starts from some initial estimate of the value function. In the case of a tabular agent, the value tables may be initialized to zero or to some other constant. The problem with deriving a greedy policy with respect to such initial value function is that the values have no correspondence with reality – they are mostly noise.

Consequently, if the initial value for action a_1 is very low, the greedy policy may never

even attempt to execute that action. This is a problem, because the value might be drastically wrong. Action a_1 might very well actually be the best action for that state – the only way to find out is to try it, which the greedy policy would not do.

We might say that if the greedy policy is derived from a value function that has recently been arbitrarily initialized and not from the optimal value function, it is basing its decisions on prejudice rather than knowledge and experience.

This is because a greedy policy is always trying to *exploit* the knowledge encoded in the values to the maximum. However, at the beginning of learning, there is not yet any knowledge to exploit. What the agent needs to do instead is to perform *exploration*. The trade-off between exploration and exploitation is always present in reinforcement learning. Should the agent sometimes attempt actions, which it believes are suboptimal? In order to do so, it must, of course, sacrifice a portion of the reward that it believes it might otherwise get. But there is always a chance that it will discover some new, more rewarding behaviour.

We also need to keep in mind that the environment might be stochastic and the same action might result in different outcomes. Therefore it might be reasonable to retry actions multiple times. There are many sophisticated approaches to controlling how much exploration the agent does and what actions it tries..

In the remainder of this section, we will look at two very simple ways that can help to introduce some amount of exploration. These are both very straight-forward, but they actually work relatively well in practice, provided that the task is not especially exploration-intensive.

5.4.1 The ε-greedy Policy

The best known exploration policy would without doubt be the so-called ε-greedy policy. In general, if learning is to progress, all actions must have a chance of being selected. On the other hand, if the agent is to perform any useful task, it is necessary that it follow the greedy policy most of the time.

The idea behind the ε-greedy policy is that the agent should mostly behave greedily, but sometimes – with some relatively small probability ε – it should select the action randomly (by sampling from a uniform distribution):

$$\pi_{\varepsilon-greedy(Q)} =_{\text{def}} \begin{cases} \arg\max_{a \in \mathcal{A}} Q(s,a) & \text{if } z = 0: \ z \sim \text{Bernoulli}(\varepsilon) \\ \mathcal{U}(A) & \text{otherwise,} \end{cases} \tag{5.23}$$

where $\mathcal{U}(\mathcal{A})$ denotes the uniform distribution over the action space $\mathcal{A}$.

Parameter ε is often annealed – learning starts with a high exploration rate and then, as learning progresses, ε is gradually reduced. The annealing schedule may be linear, exponential or any of a large number of other choices.

5.4.2 The Softmax Policy

Another natural approach to resolving the exploration vs. exploitation trade-off would be to select actions stochastically all the time, but to bias the probability distribution towards actions with greater values.

Policies based on this principle are called *softmax* rules. The most common softmax rule is based on the Boltzmann distribution, which is defined as follows [59]:

$$\pi_{softmax(Q)}(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q(s,a')/\tau}}, \tag{5.24}$$

where $\tau > 0$ is the thermal parameter, which determines to what degree actions with greater values are preferred to actions with smaller values. At low temperatures τ, the value of the action will have a very strong influence on the outcome. Conversely, at very high temperatures, the policy will approach uniform sampling.

Analogously to the ε-greedy policy, parameter τ of the Boltzmann softmax is often annealed. Learning starts with a high temperature, which drives the agent to explore more aggressively and then as the temperature is gradually decreased, the agent becomes more conservative and starts to exploit its knowledge more and more.

5.5 | Dynamic Programming

There are several fundamentally different approaches to solving reinforcement learning problems using value-based methods. We will follow Sutton and Barto's approach and start by contrasting three different approaches [59]:

- Dynamic programming (DP);
- Monte Carlo learning;
- Temporal difference learning (TD learning).

This section will discuss how MDPs can be solved using dynamic programming, provided that the transition function is known and the state-action space is discrete and not too large.

Dynamic programming is a set of techniques in software engineering, applications of which are broad and include areas other than reinforcement learning. These techniques can be used to good effect when solving tasks with recurring sub-problems. A naïve approach – often based on recursion – would have to solve every one of those sub-problems again and again as it encounters them. Dynamic programming only solves each of these sub-problems once and then reuses the solutions as many times as the sub-problem recurs. Either of the following two techniques can be used to achieve this:

- **Tabulation:** The problem is approached in a bottom-up manner. The sub-problems are solved first and stored in some kind of table. The overall solution is then built up

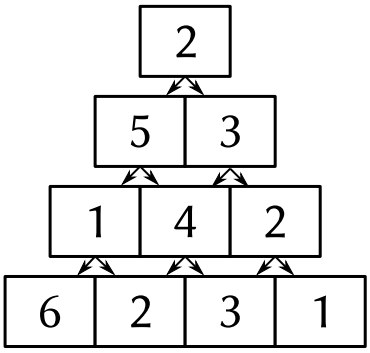


Fig. 5.9: An instance of the max-sum path pyramid problem.

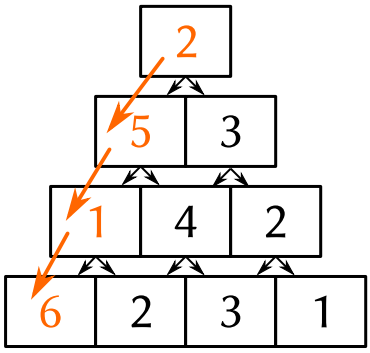


Fig. 5.10: The solution of the max-sum path pyramid problem.

from these partial solutions.

- **Memoization:** The problem is approached in a top-down manner (e.g. using recursion). Each sub-problem is only solved the first time it is encountered. After that its solution is stored in some sort of cache – usually implemented as a hash map. When the same sub-problem occurs again, the solution is retrieved from this cache.

The common feature of these two techniques is obvious: each recurring sub-problem is only solved once. If the overall structure of the problem is such that tabulation can easily be used, this tends to be slightly more efficient than using memoization, because of the overhead connected with using a hash map.

All in all, dynamic programming is very useful in the context of standard software engineering. However, its utility in the context of reinforcement learning is not large. The principal reason for this is that in reinforcement learning, DP requires multiple iterations over the entire state-action space and over all possible transitions, which is prohibitively expensive even for relatively small problems. In addition, the transition function of the MDP must be known, which is often not the case. The main reason for discussing DP, then, is to prepare ground and to provide motivation for the other two approaches: Monte Carlo learning and TD learning.

5.5.1 DP in Software Engineering – An Example

To explicitly show, which of the concepts related to dynamic programming are generic and which are RL-specific, we will first present a simple non-RL example. Although this is essentially a textbook problem, one often encounters practical problems with the same general structure. Let us have a small pyramid, as shown in Fig. 5.9. The task is to find the maximum-sum path from the top of the pyramid to its bottom. Arrows indicate which paths of traversal are open. From every node it is only possible to move downwards: to one of the two adjacent nodes in the row just below it. The correct solution is shown in Fig. 5.10. The sum along the indicated path is 14.

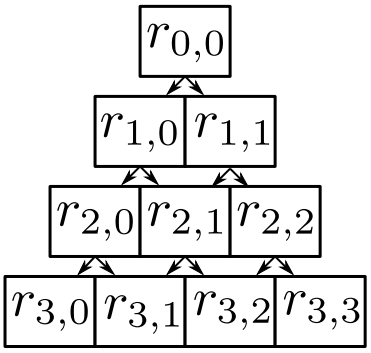


Fig. 5.11: General notation for a 4-row pyramid.

If we want to describe the problem using some more general notation, we can think of every one of the squares as a distinct state. We can number the rows of the pyramid – starting from the top (row 0); and the columns in each row – starting from the left (column 0). Thus, for each square we will have a state $s_{i,j}$, where i is the number of the row and j of the column. Similarly, the value of a square will be denoted $r_{i,j}$. For a pyramid with 4 rows, this will give us the graph shown in Fig. 5.11. The entire pyramid – as a set of all the squares – can be denoted with

$$P = \{s_{i,j} \,|\, j = 0, ..., i-1 \;\; \forall\, i = 0, 1, ..., d-1\}\,, \tag{5.25}$$

where d is the depth of pyramid P.

Solving the Pyramid Problem by Top-Down Recursion

One very natural way to approach the problem would be to start from the top of the pyramid and use simple recursion to search along all possible paths to find the one with the maximum sum. The algorithm is shown in Algorithm 2. Symbol $\frown$ denotes concatenation of sequences.

Algorithm 2: Solving the pyramid problem using top-down recursion.

```
1  function top_down(P, row = 0, col = 0) begin
2  |  if row + 1 ≥ d then return ⟨r_{row,col}, (col)⟩ ;
   |  /* The path through the bottom left square.                    */
3  |  left, left_path ← top_down(P, row + 1, col);
   |  /* The path through the bottom right square.                   */
4  |  right, right_path ← top_down(P, row + 1, col + 1);
5  |  if left ≥ right then
6  |  |  return ⟨r_{row,col} + left, (col)⌢left_path⟩;
7  |  else
8  |  |  return ⟨r_{row,col} + right, (col)⌢right_path⟩;
9  |  end
10 end
```

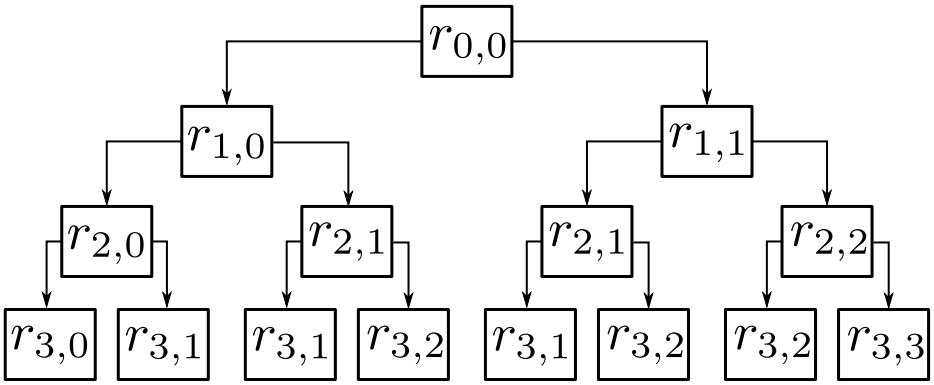


Fig. 5.12: The tree traversed by top-down recursion.

This procedure will result in summing up along the branches of the tree shown in Fig. 5.12. While this will eventually yield the correct result, note that the tree is larger than the original pyramid and that it has repeating subtrees – e.g. node $r_{2,1}$ and its subtree appear both along the left and the right path from $r_{0,0}$. This means that an algorithm based on naïve top-down recursion will have to compute the value of that subtree twice – even though the same sub-tree has already been encountered before.

For a pyramid of depth 4, the full search tree is not that much larger. However, as new rows are added, the difference in size will grow quickly and it will soon be too expensive to recompute the values of all repeating sub-trees again and again.

Solving the Pyramid Problem using Dynamic Programming

A more efficient way to solve the pyramid problem is to use dynamic programming. We can either keep the top-down approach and use memoization, or proceed from the bottom up using tabulation. We will describe the second approach, because it seems less obvious and it will also provide more useful intuitions for when we discuss reinforcement learning using dynamic programming. We will iteratively compute values $V(s_{i,j})$ for the all squares. Value $V(s_{i,j})$ is the value of the maximum-sum path from node $s_{i,j}$ downward.

The algorithm will first compute the values for all the squares at the bottom of the pyramid. This is trivial – the values are equal to the values of the individual squares:

$$V(s_{d,j}) = r_{d,j} \quad \forall\, j = 0, ..., d-1. \tag{5.26}$$

These values can then be backed up to the row above using the following formula:

$$V(s_{i-1,j}) = r_{i-1,j} + \max\{V(s_{i,j}), V(s_{i,j+1})\} \quad \forall\, j = 0, ..., i-1. \tag{5.27}$$

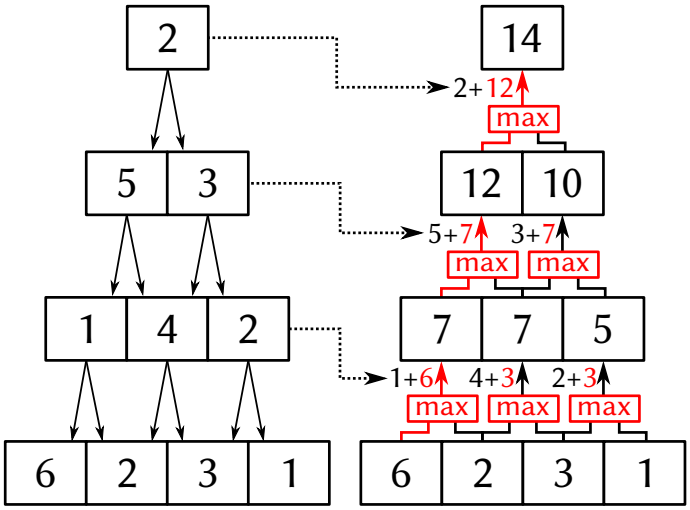


Fig. 5.13: A graphical illustration of the backup formula.

For an instance, value $V(s_{2,1})$ would be:

$$\begin{aligned} V(s_{2,1}) &= r_{2,1} + \max\{V(s_{3,1}), V(s_{3,2})\} \\ &= 4 + \max\{2, 3\} \\ &= 4 + 3 = 7. \end{aligned} \tag{5.28}$$

A graphical illustration of how the backup formula is applied is shown in Fig. 5.13.

We continue applying that same backup rule until we get to the top of the pyramid, at which point the value of the maximum-sum path will be known. The path itself will also be known, provided that we kept track of it while doing the backups. The complete pseudocode is outlined in Algorithm 3. For a sequence denoted paths, notation paths_j refers to the j-th element of the sequence and $\frown$ is the concatenation operator.

Given that we are computing the values of the nodes from the bottom up, whenever we are computing a value at row $i-1$, the values at row i have already been computed. We can reuse them multiple times without having to compute them again and again. Also, the structure of the pyramid problem is such, that once the values for row $i-1$ are known, we no longer need to store the values for row i – we can throw them away.

Comparing the Empirical Run Times

To compare the computational expense of simple recursion and the dynamic programming solution, we can run both algorithms on pyramids of various depths and compare the empirical run times – the results of such experiment are shown in Fig. 5.14. It is obvious, that for pyramids with depth greater than 5, dynamic programming is significantly faster. Note also that the vertical axis is logarithmic. The naïve recursive solution is not scalable at all.

Algorithm 3: Solving the pyramid problem using dynamic programming.

```
1  function pyramid_dp(P) begin
2      V(s_{d,j}) ← r_{d,j}  ∀ j = 0, ..., d − 1;
3      paths ← ((0), (1), ..., (d − 1));
4      for i = d − 1, d − 2, ..., 1 do
5          updated_paths ← ();
6          for j = 0, 1, ..., i − 1 do
7              if V(s_{i,j}) ≥ V(s_{i,j+1}) then
8                  V(s_{i−1,j}) ← r_{i−1,j} + V(s_{i,j});
9                  updated_paths ← updated_paths⌢((j)⌢paths_j);
10             else
11                 V(s_{i−1,j}) ← r_{i−1,j} + V(s_{i,j+1});
12                 updated_paths ← updated_paths⌢((j)⌢paths_{j+1});
13             end
14         end
15         paths ← updated_paths;
16     end
17     return ⟨V(s_{0,0}), paths_0⟩;
18 end
```

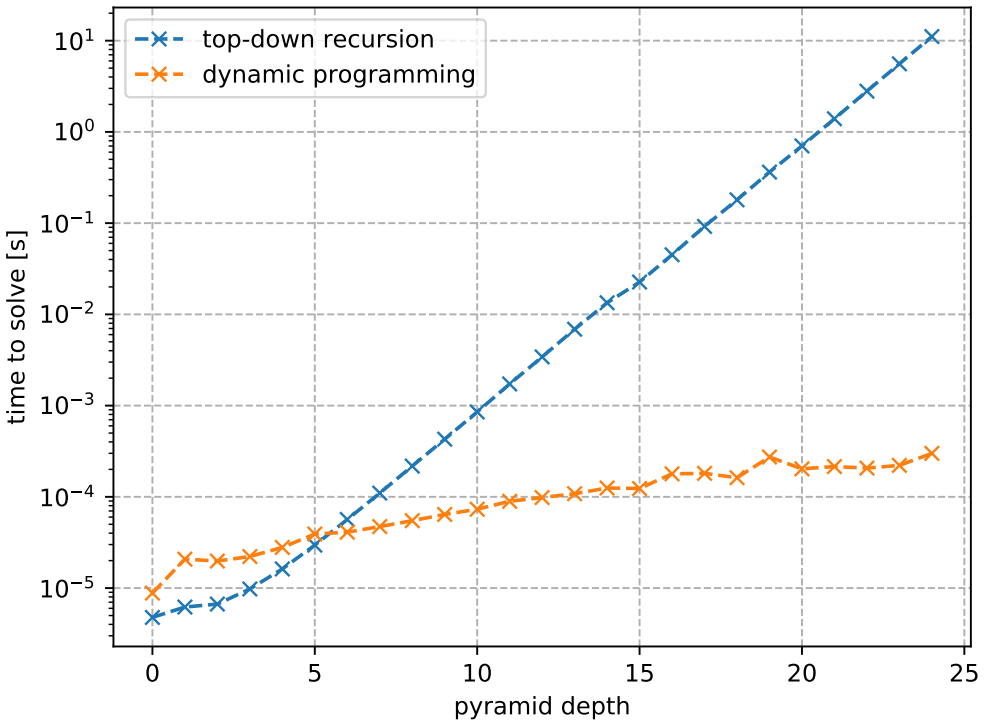


Fig. 5.14: Comparison of empirical run times: top-down recursion vs. dynamic programming.

5.5.2 Value Iteration

In the context of solving Markov decision processes, dynamic programming can be used to compute the value function, provided that the we can enumerate all states and actions and that we know the transition function. This means that we need to have a perfect model of the environment at our disposal and that the task should be discrete and the spaces not too large, lest the enumeration of all states and actions become computationally intractable.

We will start our discussion of dynamic programming for MDPs with a method called *value iteration*. The idea behind value iteration is to turn the Bellman optimality equation into an update rule and use that to find the optimal value function. Let us recall that according to the Bellman optimality equation, the following holds for the optimal value $V^*(s)$ of state s:

$$\begin{aligned} V^*(s) &= \max_{a \in \mathcal{A}} \mathbb{E}\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \,|\, s_t = s, a_t = a \right\} \\ &= \max_{a \in \mathcal{A}} \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s') \right]. \end{aligned} \tag{5.29}$$

According to this, in order to compute the optimal value $V^*(s)$, we need to know the optimal values $V^*(s')$ of all states s' reachable from s. In our pyramid example, this requirement was easy to meet. The pyramid was acyclic – there was a clear direction in which the values could be propagated. The algorithm would start at the bottom of the pyramid and work its way up layer by layer.

In an MDP, the situation is a bit more complicated. There is usually no clear ordering to the states. Also, cycles frequently arise: it is often the case that the agent can leave and reenter the same state. Value iteration work around this problem by bootstrapping the value function – it starts from a value function V_0, which is initialized in some arbitrary way*. It then gradually refines this initial estimate by iteratively applying an update rule derived from the Bellman optimality equation. This yields estimates V_1, V_2, ..., which gradually converge to the optimal value function.

The update rule is derived by substituting the optimal value function V^* with its current estimate V_k in (5.29). This yields:

$$\begin{aligned} V_{k+1}(s) &= \max_{a} \mathbb{E}\{ r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s, a_t = a \} \\ &= \max_{a} \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k(s') \right], \end{aligned} \tag{5.30}$$

This rule is applied iteratively, going over all states $s \in \mathcal{S}$ in each iteration.

*The only requirement is that the value of each terminal state should be set to 0 [59] – this needs to be so, because by definition no further rewards are received after termination.

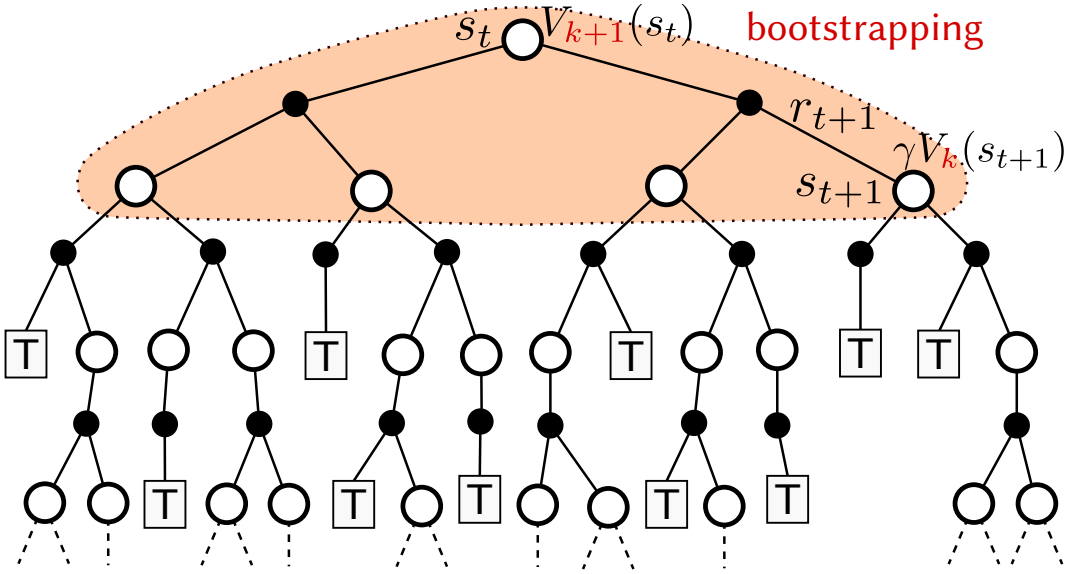


Fig. 5.15: Dynamic programming in an MDP. We approximate the Bellman expectation equation: not knowing the true values $V(s_{t+1})$ yet, we apply bootstrapping and use the current estimate $V_k(s_{t+1})$ of the value instead. (Figure from [63] with slight modifications.)

5.5.3 A Dynamic Programming Backup Diagram

The principle of dynamic programming, when applied to an MDP, is further illustrated in Fig. 5.15. DP is doing what we call a full 1-step backup – it looks one step into the future and considers all possible future transitions. Thus, the figure is very similar to the illustration given for the Bellman expectation equation earlier (see Fig. 5.8). The difference is that when doing DP, we do not yet know what the true values of the various states s_{t+1} are. Therefore we must use bootstrapping: we substitute our current estimate $V_k(s_{t+1})$ for the true $V(s_{t+1})$.

5.5.4 Value Iteration in a Grid-World Example

A straight-forward way to illustrate the recursive nature of the value functions, as expressed by the Bellman equations, is to turn back to our grid-world example. Let us suppose that we want to determine value $V_{k+1}(s)$ of a certain state in the grid-world. An illustration of such situation is given in Fig. 5.16.

It is obvious, that if the states surrounding s have certain values, then the value of s should be closely related to them, because it is possible to reach those surrounding states in a single step.

As usual, the value will also depend on the agent's policy. If we know, for an instance, that the agent will always choose to go to the right, value of state s will only depend on immediate reward $r_{\rightarrow}$ and the value of state $s_{\rightarrow}$ – because the agent will never transition into any other state.

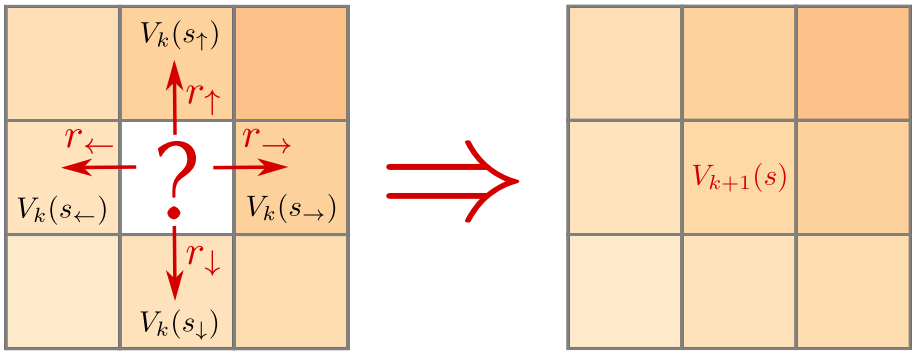


Fig. 5.16: Value iteration in a grid-world.

If we assume that rewards obtained farther in the future matter less than more immediate rewards, we can use the infinite-horizon discounted model and write that

$$V^\pi(s) = r_\rightarrow + \gamma V(s_\rightarrow), \tag{5.31}$$

always provided that $\pi(s, a_\rightarrow) = 1$. Constant $\gamma \in [0, 1]$ is the discount factor.

In the more general case, when policy π admits any one of the actions, the agent may transition into any one of the 4 neighbouring states, receiving some immediate reward in each case. All the corresponding notation is summarized in Table 5.1.

Table 5.1: Actions and their consequences in our grid-world example.

action	action's probability	resulting state	immediate reward
$a_\uparrow$	$\pi(s, a_\uparrow)$	$s_\uparrow$	$r_\uparrow$
$a_\downarrow$	$\pi(s, a_\downarrow)$	$s_\downarrow$	$r_\downarrow$
$a_\leftarrow$	$\pi(s, a_\leftarrow)$	$s_\leftarrow$	$r_\leftarrow$
$a_\rightarrow$	$\pi(s, a_\rightarrow)$	$s_\rightarrow$	$r_\rightarrow$

The value $V^\pi(s)$ can then be expressed in the following way:

$$\begin{aligned} V^\pi(s) =& \pi(s, a_\uparrow)(r_\uparrow + \gamma V^\pi(s_\uparrow)) \\ &+ \pi(s, a_\downarrow)(r_\downarrow + \gamma V^\pi(s_\downarrow)) \\ &+ \pi(s, a_\leftarrow)(r_\leftarrow + \gamma V^\pi(s_\leftarrow)) \\ &+ \pi(s, a_\rightarrow)(r_\rightarrow + \gamma V^\pi(s_\rightarrow)). \end{aligned} \tag{5.32}$$

That is to say, that the value of state s equals the sum of the γ-discounted values of the neighbouring states $V^\pi(s.)$, increased by the corresponding immediate rewards $r.$, given that the sum is weighted by the probability that the agent will transition into that state when following policy π.

If our maze was stochastic (the same action could lead into multiple states), we would, of course, have to consider all possible next states and weight them by their respective probabilities.

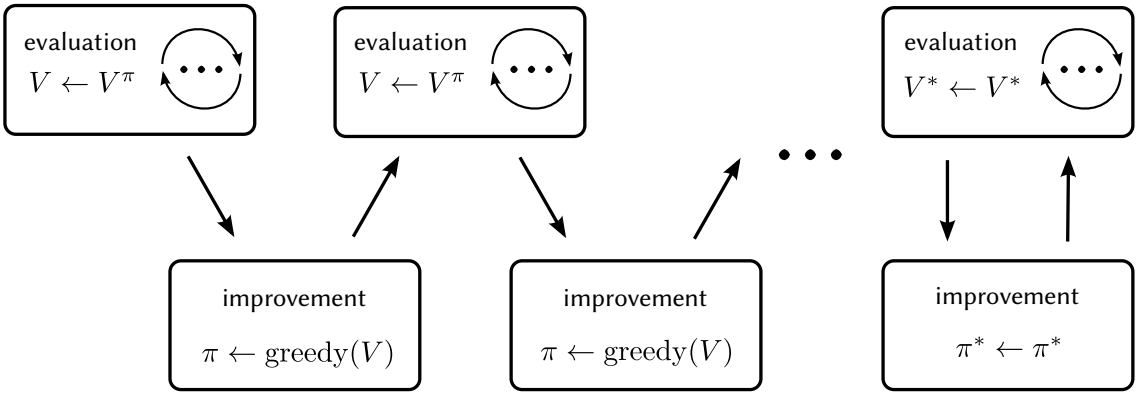


Fig. 5.17: Policy iteration.

What Policy Does the Value Function Correspond to?

The way in which value iteration computes its updates is interesting, because it does not involve any explicit formulation of a policy – it goes directly from a value function to a value function. However, we have stated hereinbefore that every value function depends on some particular policy π, because – quite naturally – rewards, that an agent obtains in the future, will generally depend on how the agent behaves. Therefore the value iteration algorithm begs the following question: What policy does any k-th estimate V_k of the optimal value function actually correspond to?

The answer is somewhat surprising: there may be no such policy. It is, of course, always possible to derive a policy greedy w.r.t. any V_k. However, that policy will have its own value function, which might not be identical to V_k.

5.5.5 Policy Iteration

Apart from value iteration, there is another way in which dynamic programming can be applied to MDPs, and which considers policies more explicitly. It is called *policy iteration* and it is based upon two distinct procedures:

- policy evaluation;
- policy improvement.

The idea of policy iteration is to start from some initial policy π_0 and to evaluate it, i.e. use the Bellman expectation equation to compute its value function V_0 (an iterative process). Once the value function is known, it can be used to improve the policy – by modifying it so that it chooses actions, which are greedy w.r.t. the value function.

By iterating over these two steps, the initial policy π_0 and its value function V_0 will gradually converge to the optimal policy π^* and the optimal value function V^*. A visual depiction of this procedure is illustrated in Fig. 5.17.

We will now turn to the two steps – policy evaluation and policy improvement – in

more detail.

Policy Evaluation

Policy evaluation is the process of finding the value function V^π that corresponds to policy π. The solution is typically computed iteratively. The process starts from an initial value function V_0 and it uses an update rule derived from the Bellman expectation equation to gradually refine this estimate until it approaches the true value function V^π of policy π at convergence.

The update rule of iterative policy evaluation is as follows [59]:

$$\begin{aligned} V_{k+1}(s) &= \mathbb{E}_\pi \left\{ r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s \right\} \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k(s') \right], \end{aligned} \tag{5.33}$$

for all $s \in \mathcal{S}$. It can be shown that $V_k \to V^\pi$ when $k \to \infty$.

To terminate the iteration, one may look at the maximum difference between the value estimates for all the states across two steps and terminate if it goes below a certain threshold, i.e. [59]:

$$\max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k(s)| \leq \delta. \tag{5.34}$$

Policy Improvement

Policy improvement is the process, which, given the value function V^π of policy π, derives a new policy with greater expected rewards. Indeed, policy evaluation is done expressly with the goal of improving the policy afterwards.

For the sake of simplicity, we will first consider the case of deterministic policies. Let us suppose that the current policy π would select action a at state s, i.e. $\pi(s) = a$. Let us further suppose that there is an action a', which has a greater value at state s, i.e.

$$Q^\pi(s, a') > Q^\pi(s, a). \tag{5.35}$$

This means that if the agent selects a' at s instead of a and afterwards continues to follow π, it will actually get a greater expected return.

Now, if this is true, then it would seem that – instead of following π afterwards – it would be better still to choose a' instead of a every time the agent gets into state s. The so-called *policy improvement theorem*, – which holds for both deterministic and stochastic policies – establishes that this actually does hold. It states the following: Let π and π' be two policies, such that [59]:

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \; \forall s \in \mathcal{S}. \tag{5.36}$$

It follows then, that policy π' must be at least as good or better than π, i.e.:

$$V^{\pi'}(s) \geq V^{\pi}(s). \tag{5.37}$$

Note that the requirement is for the condition to hold for all states. and that the action-value function $Q^{\pi}(s, \pi'(s))$ is taken w.r.t. policy π, not π'.

For stochastic policies, if there are ties when selecting the maximizing action, they do not have to be resolved in favour of a single action: each maximizing action can be given a portion of the probability in the new greedy policy. In fact, the probability can be distributed arbitrarily among the actions – as long as submaximal actions are given zero probability, the policy improvement theorem will still hold [62].

If we now extend our reasoning to all states $s \in \mathcal{S}$, we can form a new policy π', which will be greedy w.r.t. the action-value function $Q^{\pi}(s, a)$: [59]:

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^{\pi}(s, a) \\ &= \arg\max_a \mathbb{E}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= \arg\max_a \sum_{s'} P^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')\right]. \end{aligned} \tag{5.38}$$

This procedure is called *policy improvement*. Policy π' derived in this way satisfies the policy improvement theorem by construction – it is therefore guaranteed that it will in fact be an improvement upon π. Moreover, if the improved policy π' is as good as, but not better than the original policy π, then it follows that both π and π' must be optimal policies [62].

5.6 | Generalized Policy Iteration

Policy iteration, as described above, can be extremely time-consuming. This is because every policy evaluation step involves a long iterative process. Once this process completes, a single policy improvement step is taken and then another evaluation step has to start. Fortunately, in practice it is often not necessary to run the evaluation process until its completion – a few steps are usually sufficient.

This gives rise to the *generalized policy iteration* algorithm, which we have mentioned earlier: in section 4.7. Recall that it too consists of two mutually interacting processes: policy evaluation and policy improvement (as illustrated in Fig. 5.18). However, these may be implemented arbitrarily and may interact with any degree of granularity – one does not have to end before the other one begins.

The evaluation step may, for an instance, compute the full value function of the policy, but it is also free to make a very rough estimate of the value, based on just a single sample. The same holds for policy improvement: it can be as simple as deriving a greedy policy w.r.t. the current value function, or much more complex, such as optimizing parameters of

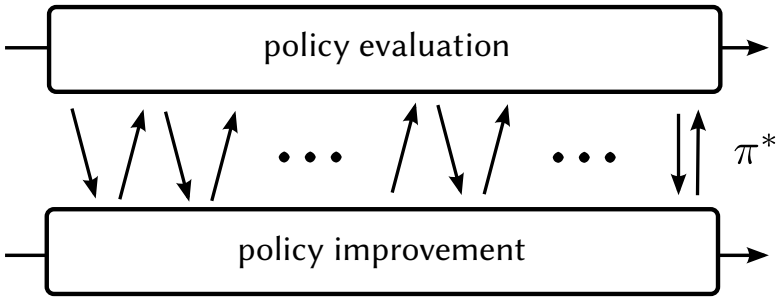


Fig. 5.18: Generalized policy iteration.

a deep neural network.

Given all this flexibility, virtually all reinforcement learning methods are well-described using the generalized policy iteration framework [62]. This certainly holds for standard policy iteration and value iteration, which both arise as special cases: when we apply full, or 1-step policy evaluation based on DP and do policy improvement by simply deriving a greedy policy.

5.7 | Monte Carlo Learning

Dynamic programming, which we have been discussing so far, has two crucial disadvantages: it requires a full distributional model (see 4.2.1) of the environment and it is extremely computationally expensive. These two properties are actually closely related: DP is expensive precisely because it needs to iterate over all possible states and compute backups by considering all state transitions that are possible according to the model.

We will now move to model-free RL methods. The first method that we will consider is called Monte Carlo learning. In order to derive it, we will start from the definition of the state-value function:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left\{R_t | s_t = s\right\}. \tag{5.39}$$

5.7.1 Using the Empirical Mean

In order to turn this equation into a practical algorithm, we will need to replace the expected value with an empirical mean, which we can compute using transitions, that the agent has actually experienced. Let us say that we have run the agent for N episodes and we have collected a set $\{\eta_1, \eta_2, ..., \eta_N\}$ of histories η_i.

Every-visit Monte Carlo Estimation

Having collected full histories, we can easily compute the actual return R_t from each time step t onwards – we already know what all the future rewards have been. We can then

approximate the expected value in (5.39) using the empirical mean as follows:

$$V^{\pi}(s) \approx \frac{1}{|D|} \sum_{R_t \in D} R_t, \tag{5.40}$$

where D is the multiset of returns R_t from transitions in all our histories η_i, which match $s_t = s$, i.e.:

$$D = \bigcup_{\eta_i} \{R_t = g(\eta_i, t) \,|\, s_t = s\}, \tag{5.41}$$

where g is a function, which computes the return R_t, given a history η and a time step t:

$$R_t = g(\eta, t). \tag{5.42}$$

Just to recount, for the infinite-horizon discounted return, the function would be:

$$R_t^{\text{dis}} = g^{\text{dis}}(\eta, t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^T r_{t+T}, \tag{5.43}$$

where T is the number of remaining times steps until the end of episode η and r_i is the immediate reward at time step i, as recorded in history η.

This way of estimating the value is called *every-visit* Monte Carlo estimation, because the estimate considers every time at which s was visited.

First-visit Monte Carlo Estimation

Besides every-visit Monte Carlo, there is an alternative, which only considers the first time each state has been visited and uses that return, i.e.:

$$D_{\text{1st}} = \bigcup_{\eta_i} \left\{ R_t = g(\eta_i, t) \,\middle|\, \begin{array}{c} s_t = s; \\ \nexists\, (s_{t'}, a_{t'}, r_{t'}) \in \eta_i : t' < t \end{array} \right\}. \tag{5.44}$$

5.7.2 Batch Monte Carlo Learning

Now that we have a way to estimate the action-values using past experience, we can formulate a batch version of Monte Carlo learning. We do this by turning (5.40) into an assignment:

$$V^{\pi}(s) \leftarrow \frac{1}{|D|} \sum_{R_t \in D} R_t. \tag{5.45}$$

This way, given a set of histories, we can estimate action-state values of all action-state pairs that were visited.

5.7.3 Incremental Monte Carlo Learning

Batch Monte Carlo learning gives us a way to make an agent learn from experience. However, it requires that all the histories be collected before the agent learn anything. This is a concern not only because we want the agent to start learning as quickly as possible – there is a more serious issue. How do we generate the transition data? If we simply pick

actions randomly, the agent is likely to behave so poorly, that it might not even see the states/actions, from which it could learn the optimal policy.

One way to solve this problem is to compute the empirical mean in (5.45) incrementally. That way we start with some initial value function (in the tabular case it will typically be zero-initialized) and gradually update this initial estimate based on experience. For all state-action pairs in a history η, we will apply the following update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[R_t - V(s_t)\right], \tag{5.46}$$

which moves the current estimate of $V(s_t)$ in the direction of the observed return R_t. The size of the step depends on the learning rate α.

This gives us a way to update the state-value function after every episode. Using an incremental mean has one additional effect – it means that past experience will eventually fade out and get replaced by what the agent has experienced more recently. This can be useful if the task is non-stationary, i.e. the MDP changes over time and the policy may need to adapt accordingly.

5.7.4 MC Learning Backup Diagram

We can again use a backup diagram – given in Fig. 5.19 – to illustrate the main idea. In contrast to DP, Monte Carlo learning does not do a full backup, but a sample backup – it only considers the transitions that actually happened. The other difference is that it does not do 1-step backups, but instead considers all transitions until the end of the episode (we could say that it applies ∞-step backups). No bootstrapping is necessary, because we use the actual return R_t as a sample estimate of value $V(s_t)$.

5.7.5 Using the Action-Value Function

The method, as stated, still relies on a state-value function. In order to derive a policy from a state-value function, 1 step lookahead is required – we need to probe all the possible next states to determine, which action has the greatest expected value. A distributional model of the environment is required in order to do this.

This is not ideal: recall that one of the main motivations behind deriving Monte Carlo learning in the first place, was to learn in a model-free fashion. However, this can be achieved, if we work with the action-value function instead of the state-value function. The action-value function lets us access the action values directly – we do not require a model to compute them. The update rule will be very similar to that for the state-value function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_t - Q(s_t, a_t)\right], \tag{5.47}$$

Note that in contrast to dynamic programming, Monte Carlo learning now:

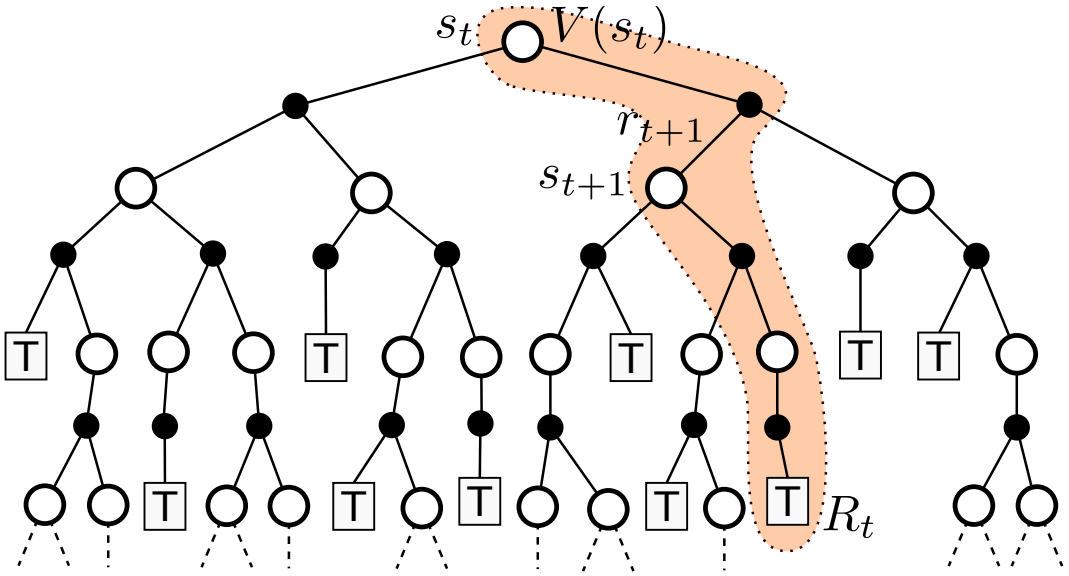


Fig. 5.19: Monte Carlo learning in an MDP: ∞-step sample backups are applied and the actual return R_t is used as the sample estimate of $V(s_t)$. (Figure from [63] with slight modifications.)

- does not require a model;
- does not need to iterate over the entire state space, and over all possible transitions: only the states-action pairs that were actually visited need to be considered;

However, Monte Carlo learning has its own shortcomings:

- It requires that the problem be episodic.
- No learning happens in the middle of an episode.
- Using a sample of the return R_t in place of its expected value can result in an estimator with very high variance (the updates tend to be very noisy).

5.8 | TD Learning

As we have mentioned, although Monte Carlo is a more practical RL algorithm than dynamic programming, its properties do not suit all kinds of tasks equally well. Namely, there are the three issues of episodicity, continual learning and high variance.

Clearly, to solve these issues, we will need to come up with a different estimator of $\mathbb{E}_\pi\{R_t|s_t = s\}$ than the sample return R_t. To do this, we can use the fact that the definition of R_t and of the value function is recursive, i.e. that:

$$V^\pi(s_t) = \mathbb{E}_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1})\}. \tag{5.48}$$

If we now again decide to approximate the expected value using a sample, we end up with:

$$V^\pi(s_t) \approx r_{t+1} + \gamma V^\pi(s_{t+1}). \tag{5.49}$$

This estimate will have much lower variance than the return R_t because it only involves the uncertainty about a single state transition, namely: the action a_t, the immediate reward r_{t+1} and the next state s_{t+1}. The return for the remaining part of the episode is expressed in terms of the value function and is, therefore, still the expected value.

This way of estimating the return will also solve the other two issues: we can now update values at each time step (all we need to know is the new immediate reward and state) and we no longer require that the problem be episodic.

The only problem, of course, is that we do not yet know $V^\pi(s_{t+1})$, which forms part of equation (5.48). However, analogously to dynamic programming, we can bootstrap the value function: start from some arbitrarily initialized values and iteratively reestimate them using the following rule [59]:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]. \tag{5.50}$$

The approach we have just motivated is known as *temporal difference learning* (TD learning). The "temporal difference" in the name refers to the difference between the original estimate of the state-value $V(s_t)$ and the estimate we get once we know the immediate reward and the next state, i.e.:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{5.51}$$

5.8.1 A TD Learning Backup Diagram

To better illustrate what kind of backup TD learning performs, we can again use a backup diagram. This also allows us to make a comparison with both: dynamic programming and Monte Carlo learning. The backup diagram is shown in Fig. 5.20 and it tells us that:

- Just like DP, TD learning uses 1-step backups and for that reason has to resort to bootstrapping (i.e. uses the current estimate $V_k(s_{t+1})$ of value at state s_{t+1}, instead of the true value $V(s_{t+1})$). This reduces variance in comparison to MC learning.
- Just like MC learning, TD learning uses sample backups instead of full backups. This means that (unlike DP) it does not require a model.

5.8.2 Action-Value TD Learning

Having introduced the state-value version of TD learning we will now again move to the action-value function, which can more easily be turned into a policy. This will yield a number of different methods – we are going to discuss the two best-known approaches in the following sections, namely SARSA and Q-learning.

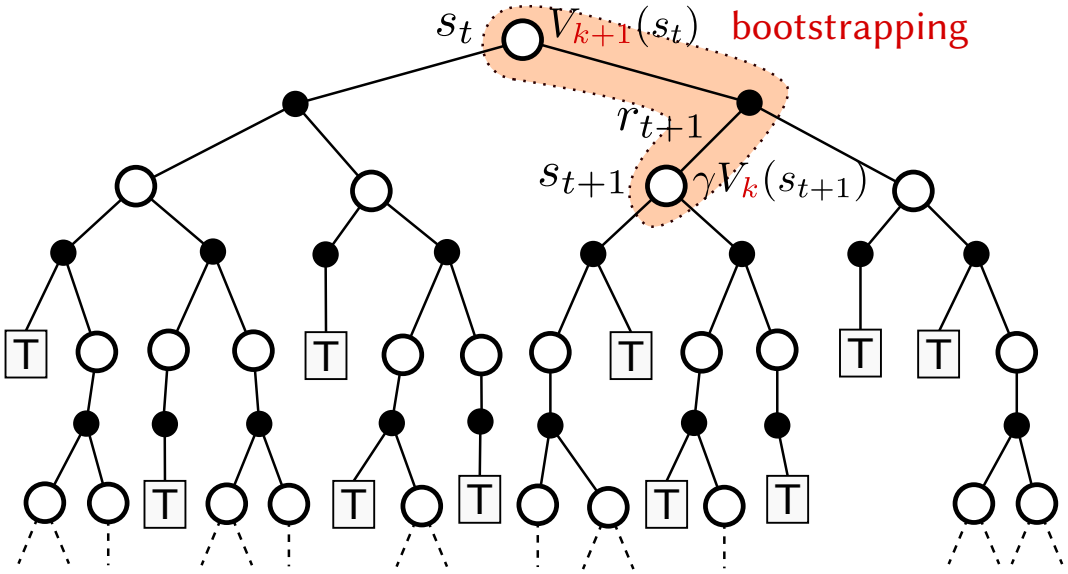


Fig. 5.20: TD learning in an MDP: 1-step sample backups are applied. (Figure from [63] with slight modifications.)

SARSA

When converting the state-value-based TD rule (5.50) into a rule based on the action-value function, we need to make one additional choice. When plugging in the Q-value w.r.t. state s_{t+1} what action do we choose to go with it?

One valid choice is to select the action a_{t+1} that the agent's policy is going to take next. This yields a method called SARSA – so named after the symbols that occur in its re-estimation rule [59]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \tag{5.52}$$

Since we need to know what action a_{t+1} the agent's policy is going to take next, the SARSA rule clearly depends on the policy. Methods of this kind are called *on-policy methods*. On-policy methods only work correctly when the state-action transitions that the policy is being trained on have been produced by that same policy. This means that one cannot use data collected using a different policy (or even an old version of the same policy) when learning using SARSA.

Q-Learning

Q-learning is an example of an action-value-based TD method, which is *off-policy*: its updates do not depend on the current policy and it is able to use samples collected using a different policy. Instead of action a_{t+1}, which the agent's current policy is actually going to take next, Q-learning uses the action with the maximum expected value (according to the

current estimate of the action-value function) [59]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{5.53}$$

5.9 | On-Policy and Off-Policy Methods

We have introduced two distinct TD learning methods based on the action-value function: SARSA and Q-learning. We have seen that they are instances of two fundamentally different types of RL methods:

- *On-policy methods:* Their update rules depend on the policy that is being followed. Training must be done online: experience collected using a different policy cannot be used for training (even older versions of the same policy).
- *Off-policy methods:* Their update rules are policy-independent. It is possible to use experience collected using an arbitrary policy, enabling techniques such as experience replay, or even learning by observing human behaviour.

5.9.1 SARSA vs. Q-Learning: The Cliff-Walking Task

To illustrate the difference between on- and off-policy methods more fully, we will next present an example from [59], which compares the behaviour of SARSA and Q-learning in a simple grid-world setting: the cliff-walking task.

The grid-world and its rules are visualized in Fig. 5.21. The agent is to walk from the start position $\mathcal{S}$ to the goal position G. If the agent steps over the cliff-side, it gets returned back to $\mathcal{S}$ and receives the immediate reward of $r = -100$ as a punishment. In all other cases the agent receives the reward of $r = -1$, to prevent it from loitering. The episode ends as soon as the agent reaches the goal state. As a result, the faster the agent makes its way to the goal state, the greater its total return. For this reason, the shortest path – the one along the cliff – is the optimal one: always provided that the agent does not fall off the cliff.

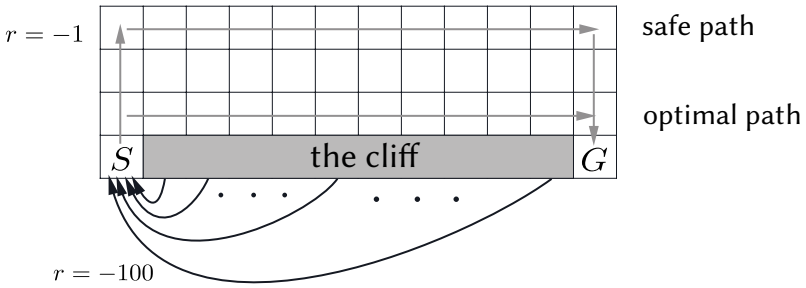


Fig. 5.21: Diagram of the cliff-walking task [59].

When we apply Q-learning to this task, it will tend to follow the optimal path. This is because it computes the action-values under the assumption that max-value actions will

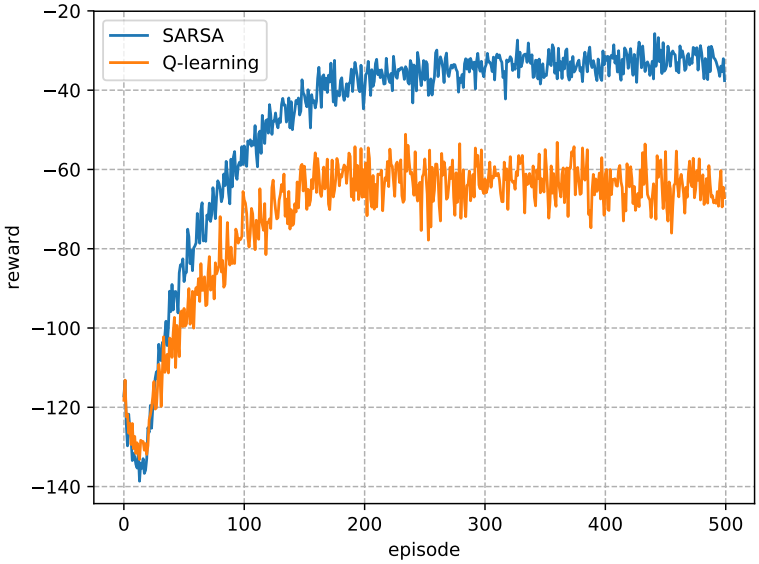


Fig. 5.22: The cliff-walking task: the learning curves.

always be taken.

However, during learning, the agent is not going to use the greedy policy. It will sometimes need to take actions that appear suboptimal in order to explore. It may use the ε-greedy policy, for an instance. If such an agent is walking alongside the cliff and selects a suboptimal action, that may cause it to fall off. Q-learning will be oblivious to this.

SARSA, on the other hand, is an off-policy method. It considers the real actions selected by the current policy. SARSA will gradually become aware of the fact that the agent is sometimes behaving erratically and that it is prone to take actions that make it fall off. It will therefore tend towards safer paths – further from the edge – where the probability of falling off is much smaller.

As a result, SARSA will be able to obtain higher returns during training – the agent will not fall off the cliff so often, as shown in Fig. 5.22. However, if we finish training at some point, turn the exploration off, and start using a greedy policy, Q-learning will easily outperform SARSA, because it has learnt the optimal and not the safe path – this is indicated in Fig. 5.23.

Therefore, in order to make SARSA converge to the optimal policy, one needs to make the policy greedy in the limit. For an ε-greedy policy this might mean decaying the ε over time so that eventually the agent stops making erratic moves and SARSA gradually switches to the optimal path.

5.9.2 Importance Sampling

On-policy methods can sometimes be converted to off-policy methods using a technique known as *importance sampling*. More generally, importance sampling is a technique for

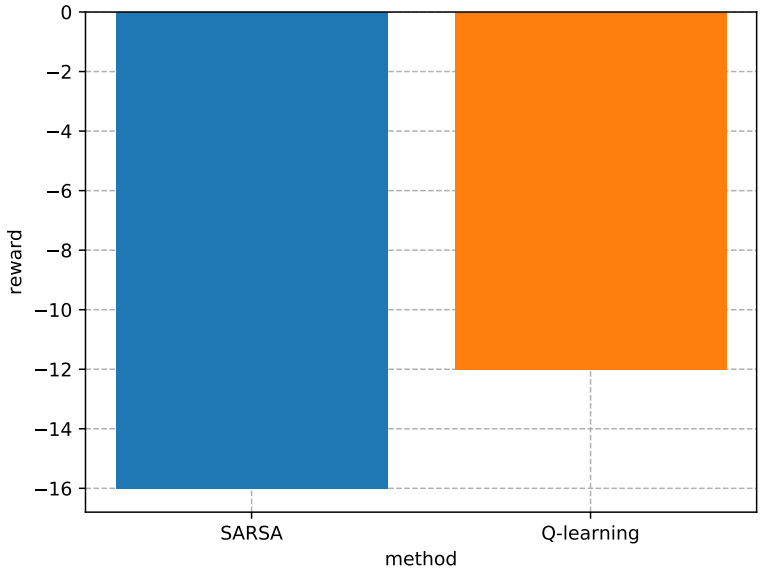


Fig. 5.23: The cliff-walking task: final performance.

estimating the expected value of some quantity $f(x)$, given a random variable $x \sim p(x)$ and using samples drawn from a different distribution $x \sim q(x)$.

Let us start with the expect value w.r.t. $x \sim p(x)$:

$$\mathbb{E}_{x\sim p(x)}\{f(x)\} = \sum_x p(x)f(x). \tag{5.54}$$

We can multiply the sum by number 1 in the form of $q(x)/q(x)$:

$$\mathbb{E}_{x\sim p(x)}\left\{f(x)\right\} = \sum_x \frac{q(x)}{q(x)}p(x)f(x). \tag{5.55}$$

It then follows that

$$\begin{aligned}\mathbb{E}_{x\sim p(x)}\{f(x)\} &= \sum_x q(x)\frac{p(x)}{q(x)}f(x) \\ &= \mathbb{E}_{x\sim q(x)}\left\{\frac{p(x)}{q(x)}f(x)\right\},\end{aligned} \tag{5.56}$$

which gives us a way to estimate the original expectation using samples from a different distribution, provided that we know both $p(x)$ and $q(x)$ so that we can weigh each term of the sum accordingly.

It is also necessary that $p(x)f(x)$ is dominated by $q(x)$, that is to say, the following must hold: $p(x)f(x) > 0 \Rightarrow q(x) > 0\ \forall x$ [64]. Otherwise values for some of the terms, where $p(x)f(x)$ is non-zero would never get sampled and thus the estimate would be biased.

5.10 | Eligibility Traces

The version of TD learning that we have described so far is generally referred to as TD(0) – and we will shortly be able to explain why. However, TD(0) has one significant shortcoming:

After observing transition $\langle s_k, a_k, s_{k+1}, r_{k+1} \rangle$, we only ever update the value $V(s_k)$ of state s_k (or else the action-value $Q(s_k, a_k)$). The values for any of the previous time steps $t < k$ do not get updated.

To see why this would constitute a problem in practice, we will illustrate the behaviour of TD(0) in a simple grid-world example. Afterwards we will introduce several ways to work around this problem.

5.10.1 TD(0) in a Grid-World Maze

To show why TD(0) will generally not be efficient enough for any practical task, we will apply it to the same grid-world example, that we have used earlier to illustrate the concept of value functions. To reiterate: the goal of the agent is to walk through a maze – from the initial position indicated by P in the figures, to the goal state denoted with G. The agent will get the immediate reward of 100 for transitioning into the goal state and the reward of 0 for every other transition.

Let us now suppose that we initialize all state values to zeros, i.e. $V_0(s) = 0 \ \forall s \in \mathcal{S}$ and that we will apply TD(0) according to rule (5.50), that is to say:

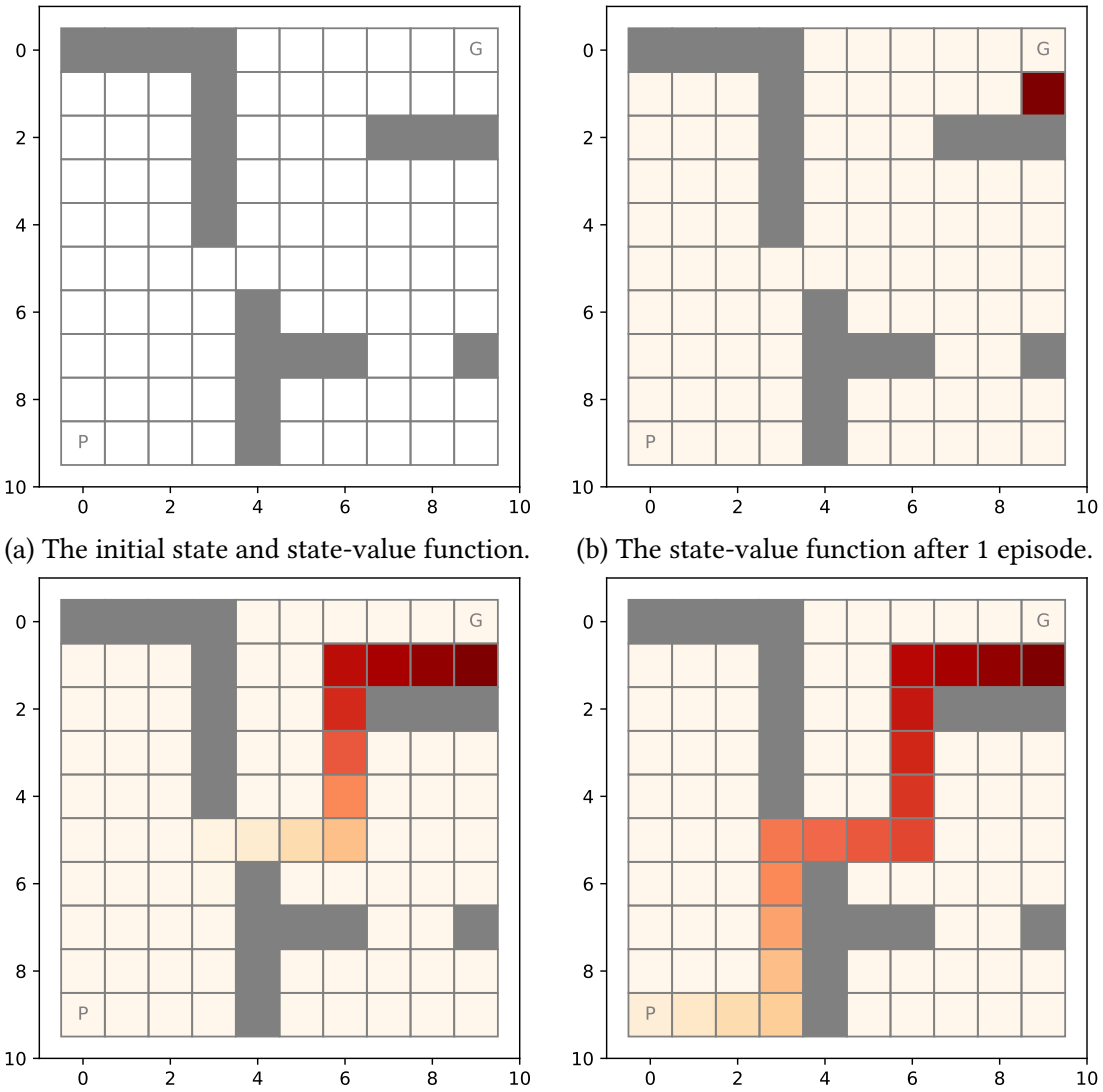
$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right], \tag{5.57}$$

If the agent has not yet visited the goal state G by time $t = k$, we can say for sure that the state-value of all states is zero: $V(s) = 0 \ \forall s \in \mathcal{S}$, because no immediate rewards have been observed at any time.

If we now suppose that the goal state will be visited at time $t = k + 1$ by transitioning from state s_k to s_{k+1} as a result of taking action a_k, we know that the immediate reward will be $r_{k+1} = 100$. Thus the state value for s_k will increase to $V(s_k) = 0 + \alpha[100 + \gamma \cdot 0 - 0] = 100\alpha$.

However, nothing will happen to the values of the states that came before, i.e.: $s_{t<k}$. This means that even after being shown the entire trajectory from the initial state to the goal state, the agent will only be able to remember the very last step of it! Even if we lead the agent by the very same path again, it will take another full episode to propagate part of the value back to state s_{k-1}, yet another episode to s_{k-2} and so on.

The process is further illustrated in Fig. 5.24. Initially, all values are set to zeros. After the first episode, the agent only remembers the last step. After going along the same path for 15 episodes, the agent remembers roughly half of the path. It takes about 30 episodes to remember the entire path. If the agent were to discover the path through exploring by itself, the situation would be much worse yet.



(a) The initial state and state-value function.

(b) The state-value function after 1 episode.

(c) The state-value function after 15 episodes.

(d) The state-value function after 30 episodes.

Fig. 5.24: TD(0) learning in a grid-world maze.

5.10.2 Solving the Problem Using Eligibility Traces

We have seen that with TD(0) a very large number of episodes is needed for the agent to be able to remember any useful action-state trajectories. In order to solve the problem, we need a way to propagate part of the value to previous states. This can be done using the so-called *eligibility traces* (or *e-traces* for a short form).

An eligibility trace $e(s)$ expresses the (presumed) contribution of each state s towards reaching the current state s_t. At each step, eligibility traces of all states $s \in \mathcal{S}$ are updated according to the following rule [5]:

$$e(s) \leftarrow \begin{cases} \gamma\lambda e(s) + 1 & s = s_t \\ \gamma\lambda e(s) & \text{else,} \end{cases} \tag{5.58}$$

where $\lambda \in [0, 1]$ is a constant.

Temporal difference learning with e-traces is denoted with TD(λ), where λ is the constant, which determines how quickly the e-traces decay over time.

The update rule for TD(λ) is, then, as follows [59]:

$$V(s) \leftarrow V(s) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]e(s) \quad \forall s \in \mathcal{S}. \tag{5.59}$$

Note that the update is now done for every state $s \in \mathcal{S}$, not just for state s_t. However, usually $e(s) \approx 0$ for nearly all states, so only a limited number of recently visited states need to be considered [5]. Also, the e-trace is usually reset whenever a new episode starts.

As we have mentioned before, standard TD without e-traces can be denoted with TD(0). Setting $\lambda = 0$ means that the e-trace decays immediately after leaving a state. This reveals standard TD according to equation (5.50) as a special case of (5.59) and explains why it is sometimes referred to as TD(0).

Accumulating E-traces vs. Replacing E-traces

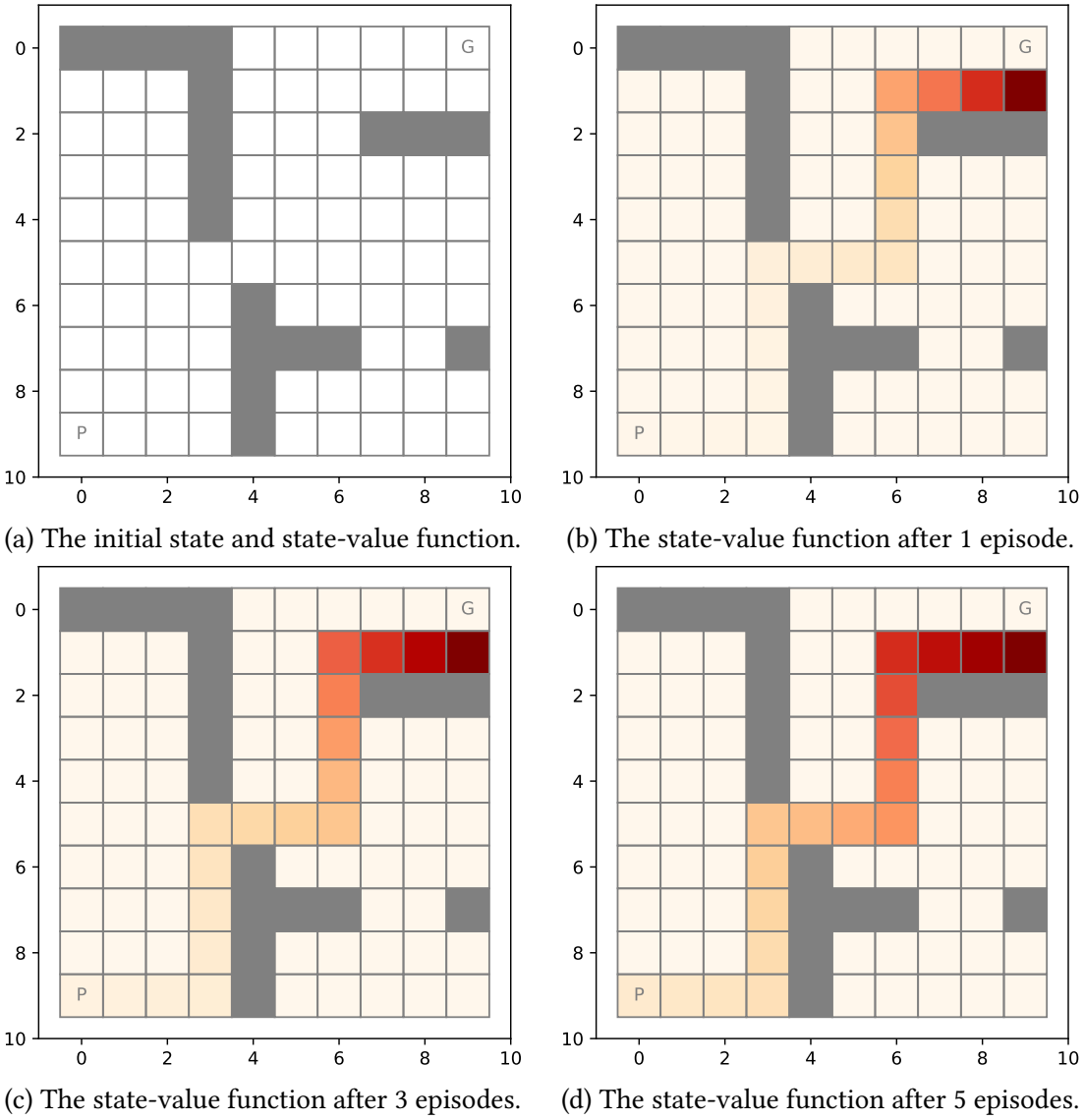
Eligibility trace defined as per (5.58) is called the *accumulating trace* [5], because the values accumulate when the same state s_t is visited again again. There is an alternative definition, which provides us with the so-called *replacing trace*:

$$e(s) \leftarrow \begin{cases} 1 & s = s_t \\ \gamma\lambda e(s) & \text{else.} \end{cases} \tag{5.60}$$

Using replacing traces prevents $e(s)$ from growing too large when a state is visited repeatedly, which can bring about significantly improved performance according to [5].

5.10.3 Eligibility Traces in a Grid-World Maze

Let us now see how TD(λ) with e-traces will fare when applied to our grid-world maze example. We will use accumulative traces with $\lambda = 0.8$. We would usually choose an even



(a) The initial state and state-value function.

(b) The state-value function after 1 episode.

(c) The state-value function after 3 episodes.

(d) The state-value function after 5 episodes.

Fig. 5.25: TD(0.8) learning in a grid-world maze.

larger λ – but with this value the results will be more illustrative. The learning will progress much faster, as shown in Fig. 5.25.

The initial state, shown in Fig. 5.25a is the same as before. However, the rest of the learning process will proceed at a very different pace. After a single episode, the agent has remembered roughly a half of the trajectory, as shown in Fig. 5.25b. After 3 episodes, there is already a faint outline of the entire trajectory (Fig. 5.25c). After 5 episodes, the path is clearly visible (Fig. 5.25d).

With a greater value of λ, the process would be faster yet – values such as $\lambda = 0.95$, or $\lambda = 0.98$ are common choices. With such λs, the agent might be able to remember the entire trajectory even after a single episode. Of course, the setting of λ is (as with any hyper-parameter) task-dependent. For an instance, one does not want to reward actions/states,

which are so far in the future that they can claim almost no credit for the present rewards.

5.10.4 Experience Replay

Experience replay is another technique, which can achieve roughly the same effect as e-traces [65, 66]. The idea is to retain the past experience (in the form of tuples $\langle s_i, a_i, s'_i, r_i \rangle$) in a replay buffer.

One can then apply TD(0) not only with the current transition, but also with some of the past transitions stored in the replay buffer. They may be sampled randomly, or presented in some specific order. If, for an instance, the TD update is applied to all states in the buffer, starting with the newest and ending with the oldest, this propagates the value in a way practically indistinguishable from e-traces. This is because when the value is propagated from state s_t to state s_{t-1}, the value of state s_t will already have been updated.

It is, of course, also possible to combine some kind of experience replay with TD(λ), which will yield a hybrid algorithm. However, the concept of experience replay will be much more important in the context of deep RL, where it represents a crucial part of several methods.

5.11 | Value Function Approximation

The traditional reinforcement learning theory typically considers tabular value functions – that is to say, the value functions are essentially tables, which store the value of each state (for the state-value function) or each state-action combination (for the action-value function) separately. This representation is very simple, which makes tabular algorithms easier to analyze. However, tabular representation has several severe limitations:

- It is only applicable to small, discrete state spaces. It cannot be applied to continuous spaces and will not scale to large discrete spaces either.
- It offers no generalization (either local or global).

Both of these issue are very serious. They are also closely related – tabular representations will not scale to very large discrete state spaces, because (given a finite amount of time) the agent will not observe some states at all, and it will encounter very few of them more than once. Without the ability to generalize, it will therefore not be able to make any progress. Conversely, systems with powerful generalization capabilities will often be able to handle very large and complex spaces – even such as raw-pixel image inputs.

In order to get generalization (as well as a smaller memory footprint) capabilities, one can approximate the value function instead of representing it by a table. The approximator may be able to perceive similarities among states and assign similar values to them – even though it may never have encountered some of them.

The memory requirements will also be lower, because instead of explicitly remembering

all the state values, it is only necessary to store the parameters of the approximator, which are usually much more compact.

There is a number of ways to approximate the value function. Most modern approaches make use of deep learning, which has the ability to generalize globally. These will be discussed in more detail in chapter 7. However, before deep learning, some of the following methods were popular [5, 59]:

- state space coarsening;
- coarse coding;
- radial basis functions (RBFs);
- tile coding;
- linear approximation;
- ...

Other choices have, of course, been explored too – such as polynomial models [67], Fourier series [68], Gaussian Processes [69], decision trees [70, 71], support vector machines [72] and many others.

CHAPTER 6

POLICY-BASED AND ACTOR-CRITIC METHODS

In the previous chapter, we have discussed value-based approaches to reinforcement learning. In the present chapter we will turn to policy-based methods, which represent a policy explicitly, and later to actor-critic methods, which combine both: an explicit value function and an explicit policy.

6.1 | Policy Search

One of the problems with value-based methods is that they do not scale to problems with very large or even continuous action spaces. This is because they represent the policy implicitly and select actions by iterating over the entire action space and comparing action-values.

If we instead represent policies explicitly, we can come up with parametrized policy distributions $\pi_\theta(s)$, from which actions can be sampled directly, i.e.:

$$a \sim \pi_\theta(s), \tag{6.1}$$

where a is an action and θ is the parameter vector, which parametrizes π_θ.

The task of reinforcement learning will then again be to tune the parameters of the policy so as to maximize the expected returns:

$$\theta^* = \arg\max_\theta J(\theta). \tag{6.2}$$

If the policy is represented explicitly, this optimization problem can be solved using policy search methods. The nature of the method will, of course, depend on the policy class – e.g. on the fact whether it is differentiable or not. In consequence, there are two broad classes of policy search methods – based on what kind of optimization method they use:

- **Derivative-based:** They apply approaches such as gradient descent or one of its many variants, or else higher-order optimization methods. Perhaps the best-known

class of derivative-based approaches is that of *policy gradient methods*, which we will consider in more detail hereinafter.

- **Derivative-free:** They make very few assumptions about the policy. Perhaps the most notable class of derivative-free methods is represented by metaheuristic methods, such as evolutionary methods. These are usually based on some relatively general insights, such as that by combining elements of two good solutions, one should be able to get an even better solution.

  We have discussed the merits and disadvantages of metaheuristic methods (and particularly of evolutionary methods) in chapter 3.2 – we will therefore not go over them again at this point. We will just note that RL methods, which apply derivative-free optimization in policy space tend to be much less sample-efficient than gradient-based methods. However, this is not an absolute rule – the authors of [73] have shown, for an instance, that a specific kind of random search can actually beat methods based on policy gradients in some of the common OpenAI Gym benchmarks.

6.2 | Stochastic Policy Representations

To explain how policies can be represented explicitly, we will now go over several well-known classes of policies – for both: discrete and continuous action spaces.

6.2.1 The Softmax Policy

The first type of explicit policy representation that we are going to discuss is the softmax policy, in which action probabilities depend on the output of a differentiable parametric function $f_\theta(s, a)$:

$$\pi_\theta(s, a) = \frac{e^{f_\theta(s,a')/\tau}}{\sum_{a' \in \mathcal{A}} e^{f_\theta(s,a')/\tau}}, \tag{6.3}$$

Function $f_\theta(s, a)$ can be as simple as a linear approximator:

$$f_\theta(s, a) = \phi(s, a)^\top \theta, \tag{6.4}$$

where $\phi(s, a)$ is some feature vector corresponding to state s and θ is the parameter vector of the linear approximator. However, $f_\theta(s, a)$ can also be a very complex function – such as a deep neural network.

Recall that earlier – in chapter 5.4.2 – we have introduced the softmax policy in the context of value-based methods and its formal definition was very similar to our softmax policy:

$$\pi_{softmax(Q)}(s, a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q(s,a')/\tau}}, \tag{6.5}$$

where $Q(s, a)$ would be the current estimate of the action-value function.

The two formulas are very similar. However, there is an important difference: In value-based methods we represent $Q(s, a)$ explicitly and then proceed to learn it – usually using some approach derived from the Bellman equation. In policy-based methods, we represent the *policy* explicitly and then optimize it directly (e.g. using policy gradients) – thus also implicitly learning some (scaled) approximation of $Q(s, a)$.

6.2.2 The Gaussian Policy

For continuous action spaces, Gaussian policies are commonly used. Actions are then drawn from the Gaussian distribution, where the mean and the standard deviation are computed using some parametrized functions $\mu_\theta(s)$ and $\sigma_\theta(s)$ (although sometimes the standard deviation is instead kept constant):

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta^2(s)). \tag{6.6}$$

The probability of action a, given a state s is then:

$$\pi_\theta(s, a) = \frac{1}{\sqrt{2\pi}\sigma_\theta(s)} \exp\left(-\frac{(a - \mu_\theta(s))^2}{2\sigma_\theta^2(s)}\right) \tag{6.7}$$

6.2.3 The Beta Policy

The Gaussian policy has one important property, which can sometimes cause significant problems in practice – it has infinite support – i.e. any action $a \in \mathbb{R}$ has a non-zero probability. However, in the majority of continuous RL problems, actions are, in fact, constrained. These constraints are typically related to physical limitations of actuators and similar.

Since a Gaussian policy will sometimes generate actions, which fall outside the feasible range, they typically need to be clipped. However, it is well-known that clipping can have disastrous effects when it comes to gradient propagation. There is no sensitivity to changes outside of the clipping range, which results in zero gradients. This can be a major problem and it is not the only one (for a more detailed treatment the reader can consult [74]).

These considerations led the authors of [74] to propose a different policy class derived from the Beta distribution. Its advantage is that the Beta distribution has finite support – the range of actions that it is capable of producing can easily be controlled. The Beta distribution's (improper) probability density function is defined as follows [74]:

$$f(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1}, \tag{6.8}$$

where α and β are parameters, which determine the shape and $\Gamma(\cdot)$ is the Gamma function, which extends factorial to positive real numbers (or even complex numbers with a positive real part). It is defined so that for every positive integer n there is the following equivalence:

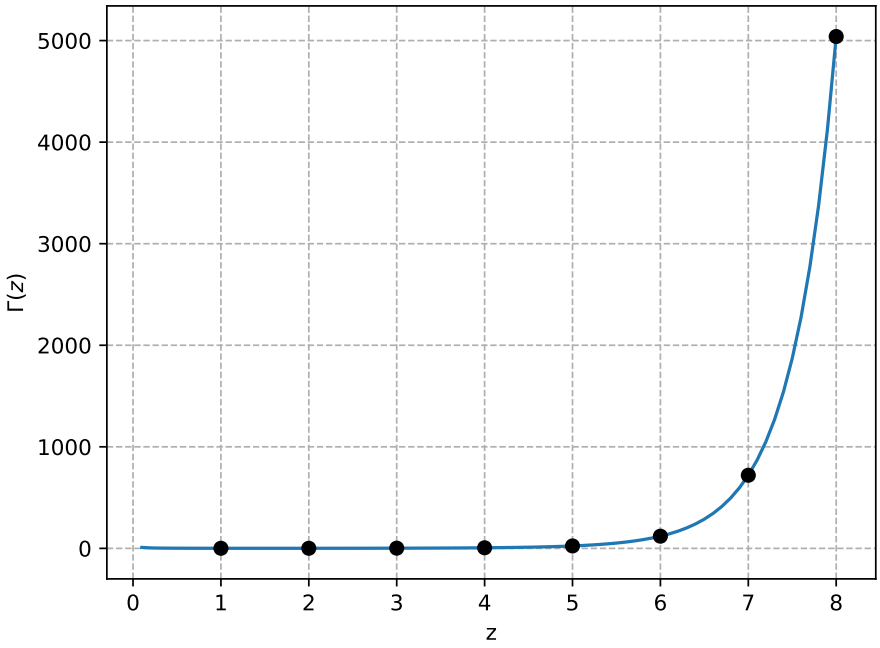


Fig. 6.1: The plot of the Gamma function. The scatter plot on top of the curve represents the corresponding factorial values.

$$\Gamma(n) = (n-1)! \tag{6.9}$$

The actual definition of the Gamma function is as follows:

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} \mathrm{d}x, \tag{6.10}$$

and its plot is shown in Fig. 6.1.

The plots of Beta's (improper) probability density function for different combinations of α and β is shown in Fig. 6.2. Note the finite support – the probability is zero for all values $z \notin [0, 1]$ (or, under an alternate definition, for $z \notin (0, 1)$).

The $[0, 1]$ range can, of course, be rescaled to any other range that may be required by the reinforcement learning problem. This makes the Beta distribution very convenient for problems with constrained actions. The way that the distribution is parametrized in deep RL is, of course, analogical to the other distributions that we have discussed. Given the input observation, the deep neural network computes the two parameters, i.e. it implements two functions: $\alpha(s)$ and $\beta(s)$.

6.3 | Policy Gradient Methods

If the policy is differentiable, it is possible to use gradient-based optimization to learn its parameters. *Policy gradient* (PG) methods represent a class of methods, which take this

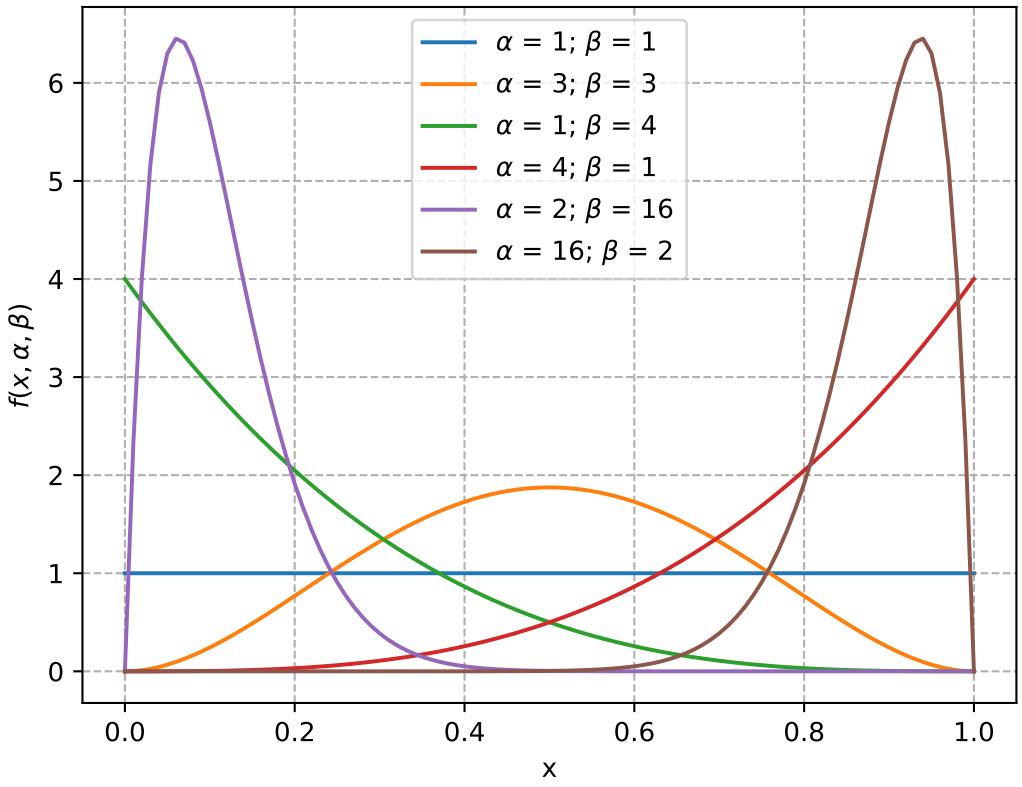


Fig. 6.2: The (improper) probability density function of the Beta distribution for different values of α and β.

approach. In order to apply gradient-based optimization, we need to express the gradient of our criterion $J(\theta)$ w.r.t. the parameter vector θ. This is expressed in the so-called policy gradient theorem, which we are going to discuss next.

For the remainder of this section, we will omit θ when denoting policy π_θ. However, this is merely for the sake of conciseness – we will still understand policy π to be parametrized by θ.

6.3.1 Policy Gradient Theorem

Let us start from the following formulation of the policy gradient [60, 61]:

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a) \,\mathrm{d}a \,\mathrm{d}s, \tag{6.11}$$

which integrates over state space $\mathcal{S}$ and action space $\mathcal{A}$. Symbol $\rho^\pi(s)$ denotes the (improper) discounted state distribution [61]:

$$\rho^\pi(s') =_{\text{def}} \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{I}(s) p(s \to s', t, \pi) \mathrm{d}s, \tag{6.12}$$

where $p(s \to s', t, \pi)$ is the density at state s' after transitioning from state s for t time steps under policy π. Also recall that $\mathcal{I}(s)$ denotes the initial state distribution. The derivation of (6.11) can be found in [60] – we will not reproduce it here (however, it is not too involved).

We would like transform this expression so that the integrals over the state and action spaces are replaced with expected values. If we manage to do that, we can make sample estimates of those expectations, thus obviating the need to actually integrate over the entire spaces, which is generally not feasible.

Note that the integral over $\mathcal{S}$ together with the state distribution $\rho^\pi(s)$ already form the expected value $\mathbb{E}_{s\sim\rho^\pi}\{\cdot\}$. If we were able to transform the integral over $\mathcal{A}$ into an expectation as well, we would be able to form the sample estimate quite easily.

To transform expression (6.11) into the required form, we can make use of the logarithm trick. We will multiply $\nabla_\theta \pi(s,a)$ by 1 in the form of fraction $\frac{\pi(s,a)}{\pi(s,a)}$. We will get

$$\nabla_\theta \pi(s,a) = \pi(s,a) \frac{\nabla_\theta \pi(s,a)}{\pi(s,a)}. \tag{6.13}$$

We can then use the fact that $\nabla \ln f(x) = 1/f(x) \cdot \nabla f(x)$ to obtain the following identity:

$$\nabla_\theta \pi(s,a) = \pi(s,a) \nabla_\theta \ln \pi(s,a). \tag{6.14}$$

This is a useful result, because when we substitute the identity back into (6.11), we get

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi(s,a) Q^\pi(s,a) \nabla_\theta \ln \pi(s,a) \,\mathrm{d}a \,\mathrm{d}s \tag{6.15}$$

where the $\int_{\mathcal{A}} \pi(s,a)\{\cdot\}\mathrm{d}a$ again forms an expected value $\mathbb{E}_{a\sim\pi}\{\cdot\}$.

This finally enables us to transform both integrals into expected values and thus yields

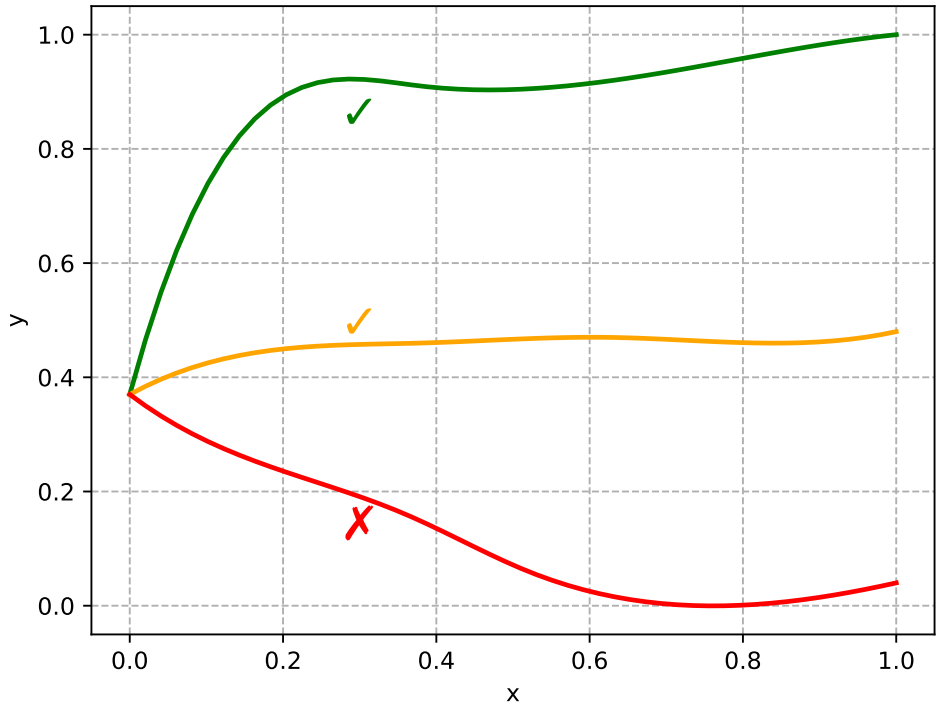


Fig. 6.3: An illustration of 3 different state-space trajectories. The top trajectory is the most rewarding, but the middle trajectory is also acceptable. The bottom trajectory should be avoided, if possible.

the so-called *policy gradient theorem* [60, 61]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}\left[Q^\pi(s,a)\nabla_\theta \ln \pi(s,a)\right]. \tag{6.16}$$

Policy Gradient: The Intuitions

To gain better intuitions as to how policy gradient methods improve the policy, we will now analyze the policy gradient theorem a bit further. Note that (6.16) has two parts – $Q^\pi(s,a)$ and $\nabla_\theta \ln \pi(s,a)$. If we simply followed gradient $\nabla_\theta \ln \pi(s,a)$, that would maximize the log-likelihood of action a in state s under policy π. This would merely make the action more probable in the given state.

By multiplying the expression with $Q^\pi(s,a)$ we are essentially saying, that the probability of each action should be increased in proportion to its expected long-term return. That is to say, better actions should be made more probable.

A graphical illustration of the idea (adapted from [75]) is given in Fig. 6.3 and Fig. 6.4. The three plots in Fig. 6.3 correspond to three different state-space trajectories. The top trajectory, shown in green and marked with a check mark, is the most rewarding one. The middle, orange-coloured trajectory is less desirable, but still acceptable. The bottom trajectory shown in red and marked with a cross mark has very low expected rewards and should be avoided, if possible.

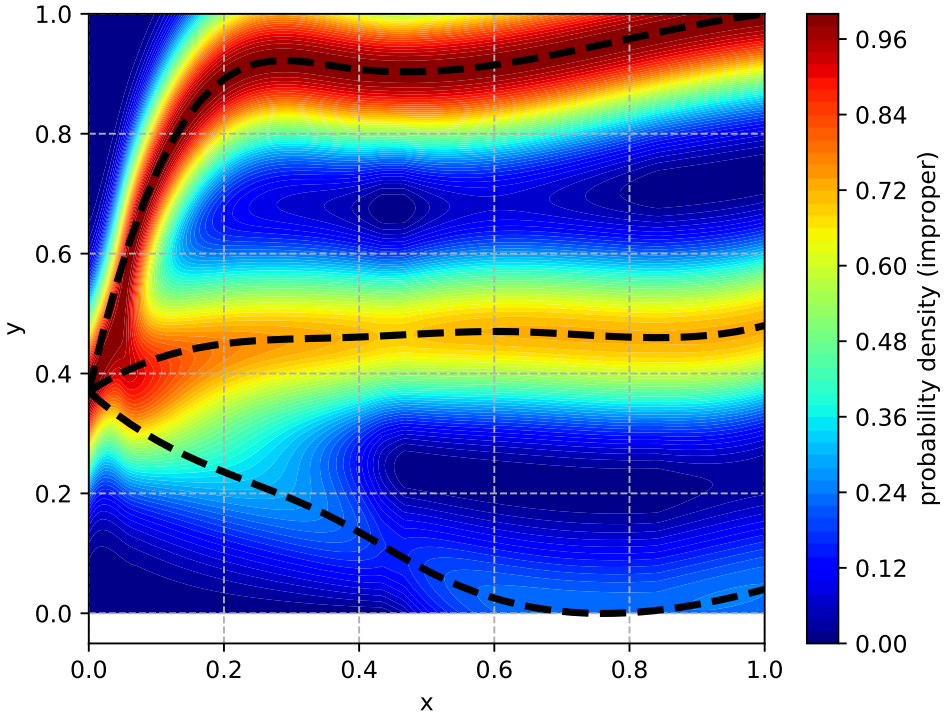


Fig. 6.4: Probability density (improper) over the state space. The most rewarding trajectory should be preferred, therefore it carries most of the density. The middle trajectory is also reasonably probable, but the probability density is noticeably lower than for the top one. The bottom trajectory is very improbable.

Fig. 6.4 shows what the resulting probability distribution might look like. Most probability density should go to the best-performing top trajectory, some to the reasonably good middle trajectory and very little to the bottom trajectory. Note that the probability density function shown in the figure is improper (it does not sum to one): this is merely to make the colours stand out more.

Policy Gradient is On-Policy

Note that the expectation in the policy gradient theorem as described in (6.16) depends on the policy π. This makes the algorithm on-policy. It is only possible to take one gradient step and then new data has to be collected. This, of course, has serious consequences for its sample efficiency. We will see how this is addressed by some methods later on.

6.3.2 REINFORCE

When applying policy gradient methods, one of the principal questions, of course, is what to substitute for $Q^\pi(s, a)$ in the policy gradient equation (6.16) – given that we generally do not know the value action a beforehand.

Perhaps the most straight-forward approach is to replace $Q^\pi(s, a)$ with its sample estimate R_t. That is to say, instead of using the true expected value, we will wait until the end

of the episode and use the actual return R_t in its place. If we approximate $Q^\pi(s,a)$ in this crude way, we get the reinforcement learning algorithm known as *REINFORCE*. The method is sometimes also labelled as Monte Carlo policy gradient, because – just like value-based Monte Carlo learning – it uses the actual return as a sample estimate of the expected return.

The advantage of this approach is its relative simplicity. The disadvantages are several, but the following two are perhaps the most critical ones:

- The algorithm is not able to learn online – we need to wait until the end of the episode to measure the actual return R_t.
- R_t is a very high-variance estimator of $Q^\pi(s,a)$ and this can make learning very difficult in some cases.

The high variance is especially critical and there are several ways to reduce it:

- add a baseline;
- introduce a critic;
- ...

6.3.3 Baselines

One way to reduce variance when using policy gradient methods is to subtract some baseline function $B(s)$ from our estimate of the action-value function $Q^\pi(s,a)$. The baseline can be any function, provided that it does not depend on a. Subtracting it will not change the expectation, because

$$\int_{\mathcal{A}} [Q^\pi(s,a) - B(s)]\, \nabla_\theta \pi(s,a)\, \mathrm{d}a = \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a)\, \mathrm{d}a. \tag{6.17}$$

This is easy to show:

$$\begin{aligned}
\int_{\mathcal{A}} [Q^\pi(s,a) - B(s)] \nabla_\theta \pi(s,a)\, \mathrm{d}a = & \\
&= \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a)\, \mathrm{d}a - \int_{\mathcal{A}} B(s) \nabla_\theta \pi(s,a)\, \mathrm{d}a \\
&= \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a)\, \mathrm{d}a - B(s) \nabla_\theta \int_{\mathcal{A}} \pi(s,a)\, \mathrm{d}a \\
&= \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a)\, \mathrm{d}a - B(s) \nabla_\theta 1 \\
&= \int_{\mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi(s,a)\, \mathrm{d}a.
\end{aligned} \tag{6.18}$$

Thus, subtracting $B(s)$ will not change the expectation, but if we choose it well, it can reduce the variance.

One good choice of a baseline is the state-value function $V^\pi(s)$. If we subtract $V^\pi(s)$ from $Q^\pi(s,a)$, we get the advantage function $A^\pi(s,a)$ [76]:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{6.19}$$

The advantage function expresses how much better action a is than average (recall that $V^\pi(s) = \int_{\mathcal{A}} \pi(s,a) Q^\pi(s,a)$). Intuitively, it actually makes more sense to weight actions by

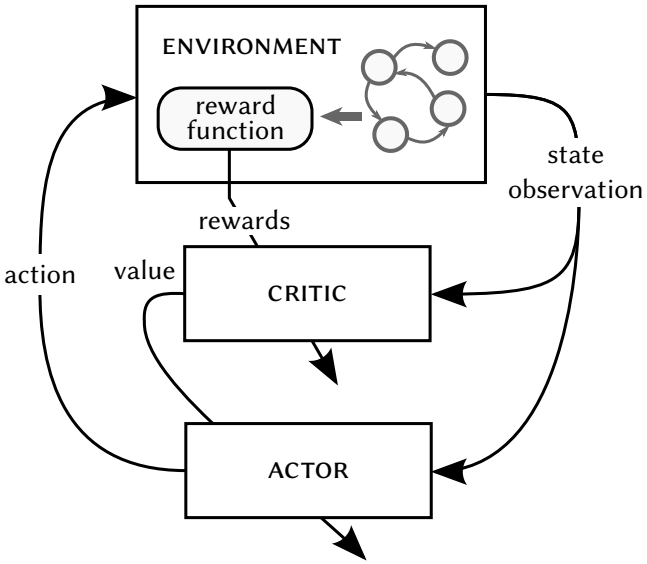


Fig. 6.5: Actor-critic methods: an illustration.

their advantages rather than by their values. That way, probabilities of sub-average actions will be driven down and we will only try to increase the probabilities of actions, which do better than average.

In practice, this baseline can be implemented by running several episodes and forming a sample estimate of $V^\pi(s)$. The sample estimate would be formed by averaging the sample returns R_t over all the episodes, i.e.

$$B(s) = \frac{1}{N} \sum_{i=1}^{N} R_{t,i}, \tag{6.20}$$

where $R_{t,i}$ is the sample return from episode i and the total of N episodes were run.

6.4 | Actor-Critic Methods

When applying policy gradient methods, we need to determine the gradients of the objective $J(\theta)$ using the policy gradient theorem. This requires us to supply the action-value $Q^\pi(s, a)$. We have seen that it is possible to form a sample estimate of $Q^\pi(s, a)$ trivially, by waiting until the end of the episode and measuring the actual return R_t. However, this estimate has very high variance. This can be reduced somewhat by subtracting a baseline, but that may not always be sufficient for practical purposes.

A more sophisticated way to estimate $Q^\pi(s, a)$ would be to represent the action-value function explicitly and to learn it using value-based methods. This can reduce variance substantially. Methods that take this approach are known as *actor-critic methods*, because they combine an explicit policy representation (the actor) with an explicit value function (the critic) and learn both jointly. The principle is illustrated graphically in Fig. 6.5.

Virtually all recent actor-critic approaches are based on deep learning. We will therefore

reserve discussion of practical actor-critic methods for chapter 7 on deep reinforcement learning.

6.5 | Exploration in Continuous Action Spaces

As we know, in order to apply RL to continuous action spaces, we need to represent the policy explicitly. The policy can be:

- *Deterministic:* Left to itself, the policy always selects the same action in each state. With deterministic policies it is therefore usually necessary to add some noise during training to get adequate exploration. Gaussian noise is a popular choice because of its simplicity.
- *Stochastic:* Additional noise is usually not required, because the policy itself is already stochastic. However, it is typically still necessary to encourage exploration, otherwise a policy could easily collapse into a deterministic one. Most methods achieve this by controlling the entropy of the policy. The entropy can, for an instance, be included in the objective that the agent is trying to maximize.

CHAPTER 7

DEEP REINFORCEMENT LEARNING

As we have mentioned in earlier chapters, unless a task has an extremely tiny state space, it will not be feasible to represent the value function using a table – some kind of function approximation will be necessary. For complex tasks, where powerful generalization capabilities are required – especially tasks, which require global generalization – deep neural networks have proven to be a good choice.

In recent years, deep reinforcement learning has been able to solve a number of very challenging problems. To give just a few instances:

- *Atari:* In 2013, Mnih et al. have shown that deep RL is able to learn how to play Atari games from raw pixel representations [77]. This means that the agents are able to learn how to do computer vision in parallel to actually learning how to perform the task. By 2015 such agents were already able to outperform humans in a number of these games [78].

  More recently, Pathak et al. have shown that RL agents imbued with curiosity can learn to play most of these games even without being told what the goal is [19].
- *AlphaGo:* In 2016, an artificial player based on deep RL has defeated Lee Sedol, the grand champion in the game of Go [79]. The system has later been enhanced further and made game-agnostic, achieving success in other games, such as chess [80].
- *AlphaStar:* In early 2019, human players have been defeated in another challenging game – the popular real-time strategy game known as StarCraft II [81]. Games of this kind are very challenging to learn even for human players – they require knowledge of strategy as well as tactics. The player needs to build up and sustain an economy and get well-acquainted with the various kinds of units that are at their disposal, which makes this achievement very impressive.
- *Recommendations:* There have recently been several high-profile applications that use deep RL to give recommendations. YouTube uses deep RL to recommend videos – the

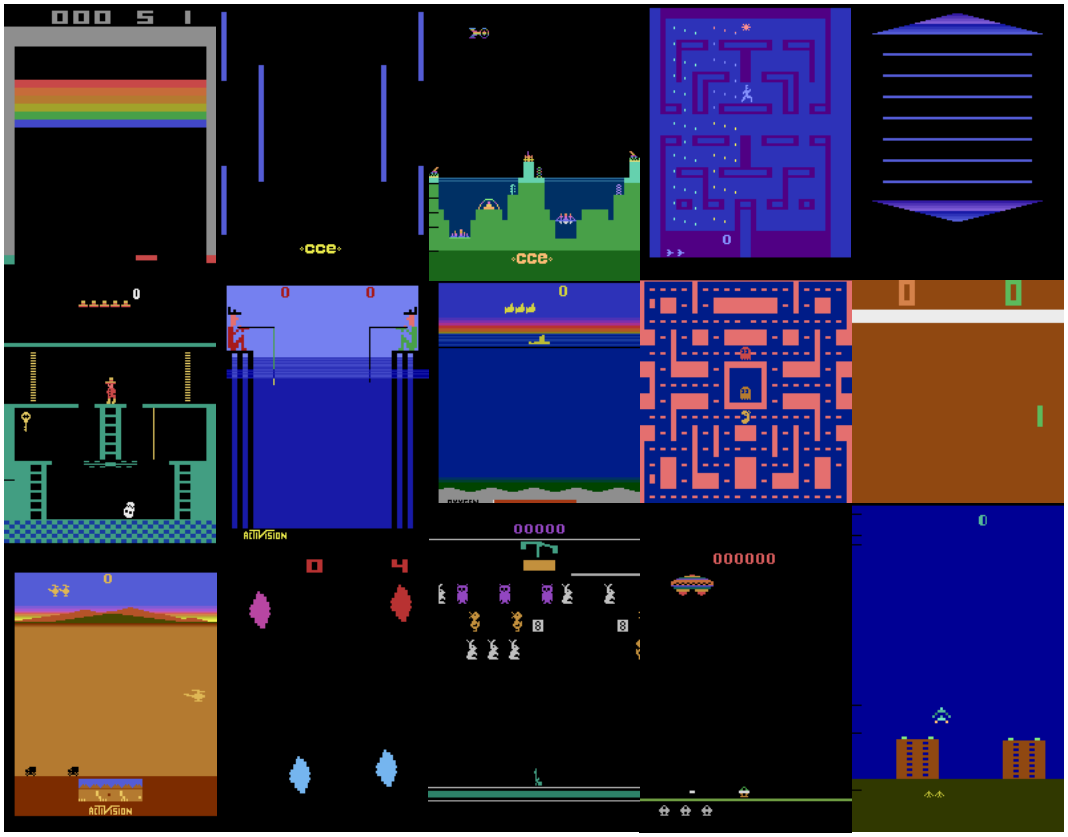


Fig. 7.1: A sample of several different Atari games.

Fig. 7.2: The AlphaStar agent playing StarCraft II [81].

goal is to maximize user engagement [82]. Facebook, on the other hand, is using deep RL to filter notifications [83].

- *Data centre cooling:* Google's DeepMind has developed an approach, which uses deep RL to control cooling in their data centres. The approach has been able to cut cooling-related costs by up to 40% [84, 85].
- *Robotics:* There has been a number of successful applications from the area of robotics such as learning of physical skills from visual demonstrations [86], controlling a human-like hand to turn objects around [87], learning dexterous manipulation of objects, such as turning a door handle or a valve [88], devising more robust deep RL methods suitable for real-world robots [89], and more.

7.1 | The Problem of Catastrophic Forgetting

One of the reasons why neural networks have not been used more extensively for value function approximation in the past, is that neural networks are not good at incremental learning. When a classical RL algorithm decides to update the value of some state s, there is no easy way to implement this update using a neural network. This shortcoming is actually connected with the ability of neural networks to perform global generalization. Whenever we update the value of one state, that update might affect the values of other states – even states, which, on the surface, seem totally unrelated.

This is true about neural networks in general and not specific to their applications in reinforcement learning. Once a neural network has been trained on a dataset, it is not easy to add new knowledge to it. If the network is trained using new data, it is likely to quickly forget all that it has learnt on the original dataset – a phenomenon known as *catastrophic forgetting*.

The problem that is specific to reinforcement learning, though, is that the transitions that come after one another tend to be strongly correlated. Training on transitions as they come in would therefore be roughly analogical to training a supervised neural network with the same class for a large number of steps – the network will forget about what it has seen before.

7.2 | The Deep Q-Network

If neural networks cannot be trained incrementally, then the obvious question to ask is, whether reinforcement learning methods could be modified to work in batch mode. The answer to this question is yes – and the approach is related to the concept of experience replay.

One of the early methods that make use of this concept is the deep Q-network (DQN). The main idea behind it is to record all experienced transitions $\langle s_i, a_i, s'_i, r_i \rangle$ in a so-called

replay buffer. At each learning step, we can then sample random transitions from this buffer and form a supervised mini-batch from them. Due to the random sampling, these transitions should not be too correlated and we should, therefore, be able to successfully train the neural network using them.

When forming the supervised mini-batches, the targets are computed according to:

$$\begin{aligned} x_i &= (s_t, a_t) \\ y_i &= r_i + \gamma \max_{a \in \mathcal{A}} \hat{Q}_T(s'_i, a), \end{aligned} \tag{7.1}$$

where Q_T is the so-called target network.

In the original DQN paper [77], the target network $\hat{Q}_T$ was simply the same as the current Q-network $\hat{Q}$. It was our most recent approximation of the Q-function. However, this meant that the targets were computed using the same network, that was being optimized. Having such moving targets can easily make the training unstable, so in a later version of the approach [78], the authors have therefore introduced the separate target network. The target network can be an old, frozen version of $\hat{Q}$ from several steps back.

There are two ways to synchronize the networks:

- *Hard updates:* The two networks are only synchronized every n steps by doing a hard assignment $\theta^- \leftarrow \theta$, where θ^- are the weights of the target network and θ are the weights of the current network. This means that the targets will remain fixed for the n steps.
- *Soft updates:* The two networks are synchronized at every step, but softly: the target network is only moved towards the current $\hat{Q}$ by some relatively small step (the step size determined by a hyperparameter), i.e.:

  $$\theta^- \leftarrow (1 - \eta)\theta^- + \eta\theta, \tag{7.2}$$

  where η is the hyperparameter, which governs the size of the step. Under this update rule the targets are moving, but we can control how fast.

7.2.1 Double DQN

The DQN algorithm, as outlined above, has one more problem: it tends to overestimate values. As we know, the Q-network is initialized randomly at the beginning. It is then quite likely that at many states, some of the values will be initialized too optimistically. This would not necessarily constitute a problem – optimistic initialization is sometimes even used as an exploration strategy. The problem is that the overestimations will not occur uniformly, and they will also not be reserved to under-explored states and actions. The authors of [90] have shown that this can have adverse effect on the quality of the policy.

The idea behind *double DQN* is that when computing the targets, we can use one network to select the best action and a different network to compute its value. That way the

overestimated values will not be as likely to get selected. Naturally, with the DQN we already have two networks: the current Q-network $\hat{Q}$ and the target Q-network $\hat{Q}_T$. If we then decide to use $\hat{Q}$ to select the best action, but use $\hat{Q}_T$ to estimate its value, we get the double DQN target [90]:

$$y_i = r_i + \gamma \hat{Q}_T(s'_i, \arg\max_{a \in \mathcal{A}} \hat{Q}(s'_i, a)). \tag{7.3}$$

Empirical results indicate that double DQN can have significantly better performance in comparison to standard DQN in some tasks. It is now common practice to use it by default whenever applying the DQN.

7.2.2 Duelling DQN

A further extension to the standard DQN method is the duelling DQN, which modifies the architecture of the network. Instead of predicting the values of the actions, the network splits into two separate paths: one that predicts the value of the state and another, which predicts the advantages of all the actions. Both the paths are then again combined using a special aggregating layer, which finally outputs the estimated Q-values [91].

The advantages of this architecture manifest when the task has states, in which multiple actions have similar values. If, for an instance, there is a state, where the agent is falling off a cliff, that state is inherently bad. At that point, the agent is going to hit the bottom whatever action it takes. It is unnecessary to estimate the value of each action at such states – they will all be similar. By explicitly decomposing the values into state values and advantages, we make it easier for the network to learn that a state is good or bad without forcing it to learn about the effect of each action in each state.

The Architecture

The architecture of the duelling DQN is illustrated in Fig. 7.3. As shown, there is a single path, which handles preprocessing and feature extraction. This then splits into two separate paths: one for the state-values and one for the advantages. Both paths are then combined again using the aggregation layer.

The key idea behind the double DQN approach is that it only modifies the architecture of the network (a well-defined operation, which can even be done automatically in many frameworks). This change is orthogonal to any changes in the RL algorithm itself. The duelling DQN can be used with any variant of the DQN method, such as the double DQN or DQN with prioritized experience replay (see section B.1).

The Aggregation Layer

Ordinarily, each of the two paths in the network would need to be trained separately – each by its dedicated algorithm. The clever thing about duelling DQN, though, is that it does not

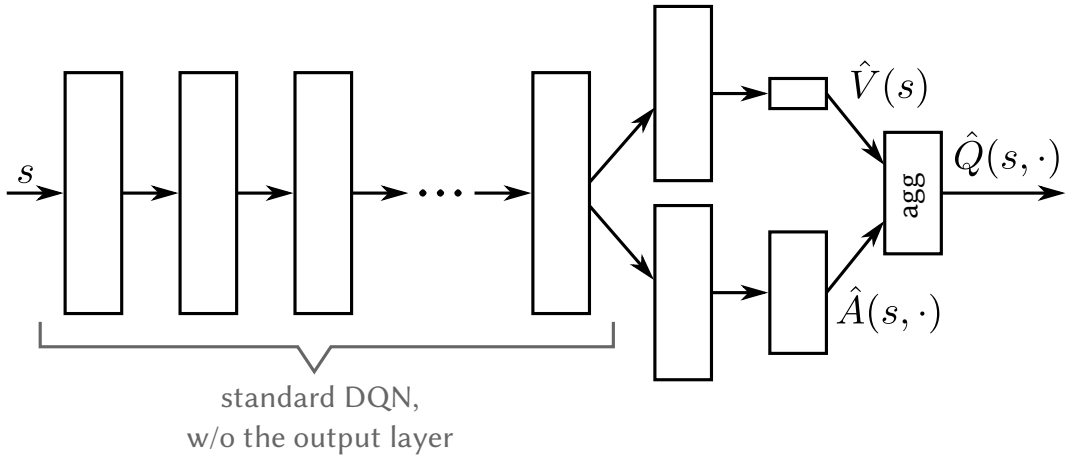


Fig. 7.3: The duelling architecture. The part on the left can be an arbitrary DQN architecture (without the final layer, which outputs the Q-values). After that the network splits into two separate paths: one computes state-values and the other advantages for all the actions. These are then combined using a special aggregation layer to compute the Q-values.

require any special supervision for the state-values and the advantages. It learns how to decompose the Q-values by itself – using the special aggregation layer.

This layer does not merely add the state-values and the advantages to compute the Q-values – if it were implemented in that way, the approach would not work. The state-values and the advantages would not be identifiable from the signal about the Q-values. However, duelling DQN uses some useful identities that hold for deterministic policies.

Let us go through them in more detail. First of all, we need to recall that the following relationship exists between the state-value function and the action-value function:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^{\pi}(s,a), \tag{7.4}$$

and that the advantage is defined as follows:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s). \tag{7.5}$$

Let us now consider the case of a greedy deterministic policy, where the action with the maximum value is always selected, i.e.:

$$\pi(s,a) = \begin{cases} 1 & \text{if } Q^{\pi}(s,a) > Q^{\pi}(s,a') \;\; \forall a' \in \mathcal{A} - \{a\} \\ 0 & \text{otherwise.} \end{cases} \tag{7.6}$$

It is clear that under these conditions, equation (7.4) simplifies to

$$V^{\pi}(s) = Q^{\pi}(s,a^*), \tag{7.7}$$

where $a^* = \arg\max_{a \in \mathcal{A}} Q^{\pi}(s,a)$ is the maximum-value action.

This, in turn, means that the advantage of action a^* is zero:

$$\begin{aligned} A^\pi(s,a^*) &= Q^\pi(s,a^*) - V^\pi(s) \\ A^\pi(s,a^*) &= Q^\pi(s,a^*) - Q^\pi(s,a^*) = 0. \end{aligned} \tag{7.8}$$

This is why the authors of [91] propose to combine the state-values and the advantages in the aggregation layer as follows:

$$\hat{Q}(s,a) = \hat{V}(s) + \left(A(s,a) - \max_{a' \in \mathcal{A}} \hat{A}(s,a') \right). \tag{7.9}$$

This definition makes sure that $\hat{Q}(s,a^*) = \hat{V}(s)$ and that the advantage of a^* is zero.

The authors then go on to present a second version, where the $\max$ operator is replaced by an average. This second version does not have all the properties of the first one, but its behaviour is more stable [91]:

$$\hat{Q}(s,a) = \hat{V}(s) + \left(\hat{A}(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} \hat{A}(s,a') \right). \tag{7.10}$$

This second version is the canonical one, which the authors use in all of their experiments.

7.3 | Advanced Approaches in Deep RL

To keep the chapter concise and to only present the fact that are absolutely necessary to understand the contents of the following chapters, we will defer discussion of more advanced deep reinforcement learning methods unto appendix B.

The following chapters will only make use of the DQN algorithm. This is mainly because our tasks have discrete action spaces and the DQN (being an off-policy method) is more sample-efficient than on-policy actor-critic approaches such as PPO.

However, a more stable off-policy actor-critic method called soft actor-critic has recently been proposed and it is possible that it could achieve better results than the DQN. Unfortunately, it was not yet available at the time when the experiments were run. However, we will at least include its theoretical description in the appendix – to show what is so special about it – and leave its testing for future work.

CHAPTER 8

ABSTRACTION AND ATTENTION IN DEEP RL

For many years, one of the big topics in reinforcement learning has been how to arrive at representations, which would be able to effectively abstract from unimportant details and facilitate quick and efficient learning. In the present chapter, we will discuss the problem of abstraction in the context of deep reinforcement learning. Using the game of Pac-Man as an example, we will show how the concept of visual attention can help to make an RL agent learn faster and also to make it more transferable to different instances of the same problem.

In [5], Frommberger defines abstraction in the following way:

> Abstraction is the process or the result of reducing the information of a given observation in order to achieve a classification that omits information irrelevant for a particular purpose.

The same author then goes on to distinguish among three different facets of abstraction, which he calls [5]:

- *Aspectualization:* abstracting from certain aspects – i.e. not considering the colour of an object or its shape. Aspectualization reduces the dimensionality of observations.

  Formally, it can be defined as a function $\kappa : \mathcal{D}^n \rightarrow \mathcal{D}^m$, where $n, m \in \mathbb{N}$ and $n > m$, and for which the following holds:

  $$\kappa(s_1, s_2, ..., s_n) = (s_{i_1}, s_{i_2}, ..., s_{i_m}), \tag{8.1}$$

  where $i_k \in [1, n],\ i_k < i_{k+1}\ \forall k$.

  In order to apply aspectualization according to this definition, the representation must be such, that the aspect we want to eliminate is represented by one or several distinct dimensions. It must not be mixed with other aspects. This property is referred to as *aspectualizability*.

- *Coarsening:* reduces resolution, making the space of possible observations smaller. It can formally be defined as a function $\kappa : \mathcal{D}^n \to \mathcal{D}^n$:

$$\kappa(s) = (\kappa_1(s_1), \kappa_2(s_2), ..., \kappa_n(s_n)), \tag{8.2}$$

  such that $\kappa_i : \mathcal{D} \to \mathcal{D}$ and at least one κ_i is non-injective (i.e. maps several elements from its domain to the same element).

- *Conceptual abstraction*:* uses existing features to form new abstract concepts (i.e. combining eyes and mouth to form a face). This type of abstraction is the most general and the other two types can be viewed as its special cases.

  It is formally defined as a non-injective function $\kappa : \mathcal{D}^n \to \mathcal{D}^m$, where $m, n \in \mathbb{N}$:

$$\begin{aligned} \kappa(s_1, s_2, ..., s_n) = \quad & (\kappa_1(s_{1,1}, s_{1,2}, ..., s_{1,h_1}), \\ & \kappa_2(s_{2,1}, s_{2,2}, ..., s_{2,h_2}), \\ & ..., \\ & \kappa_m(s_{m,1}, s_{m,2}, ..., s_{m,h_m})), \end{aligned} \tag{8.3}$$

  such that $\kappa_i : \mathcal{D}^{h_i} \to \mathcal{D}$ and $h_i \in \{1, ..., n\}\ \forall i \in \{1, ..., m\}$. Note also that each $s_{j,k}$ refers to some $s_i \in \{s_1, s_2, ..., s_n\}$.

8.1 | Abstraction in Deep Neural Networks

As we discuss in appendix A, deep neural networks are good at preprocessing their inputs into representations, which make learning the task easier. We have also shown that a neural network uses its layers to build up increasingly abstract features.

We could therefore say that deep learning is able to learn how to perform abstraction in the sense of conceptual abstraction – the most general type of abstraction according to Frommberger's scheme [5].

However, neural networks – whether shallow or deep – have several properties, which may constitute a problem in certain cases. These include especially:

- *Fixed-size inputs:* The inputs of the network are of a fixed size. This may become a problem even in relatively simple settings. It is enough that the size of the state representation changes between various instances of the RL task.

  A clear instance of this can be given using the game of Pac-Man, where different levels of the game may have layouts of different sizes. If we design an agent with a fixed-size input and crop the state representation to only include some region just around the agent, this makes the relatively straight-forward task partially observable and more difficult to learn.

*The original author uses the name "conceptual classification", which we found to be a bit misleading. We have therefore decided to use the term conceptual abstraction instead.

- *Uniform attention:* A neural network – especially a convolutional network – tends to pay an equal amount of attention to all parts of its input. Also its computational resources tend to be distributed evenly. At times, such an approach can be wasteful, and we would prefer our system to pay more attention to the most relevant portions of the input and less to the rest of it.
- *Large amounts of data:* Deep learning models require a large amount of data. One could say that this is precisely because they are learning how to preprocess their data.

  If a classifier needs to learn how to do computer vision, in addition to performing classification, it will naturally need to have significantly more parameters and will require much more data to generalize correctly.

8.1.1 Learned Abstractions

In terms of the three different abstraction facets, we could say that these issues are at least partially caused by our learning system's not doing enough aspectualization and coarsening. But naturally, deep neural networks are able to generalize globally, which enables them to perform abstraction in the sense of conceptual abstraction.

Why then would we be concerned about deep neural networks not applying the other two abstraction facets sufficiently? We know, after all, that conceptual abstraction subsumes both aspectualization and coarsening. Surely, then, the deep network will be able to learn how to perform all three?

The problem, of course, is precisely that the ability to perform conceptual abstraction has to be learnt first, and this precludes it – by definition – from providing some of the benefits that we should otherwise expect from both – aspectualization and coarsening. These include, most notably:

- the ability to learn from less data;
- computational savings;
- the ability to allocate most computation to highly relevant portions of the input;
- ...

8.1.2 Abstraction Priors

Let us start from the intuition that our learning system could benefit from applying more coarsening, aspectualization, or from focusing its attention more narrowly. How do we incorporate such insights into the design of our neural model?

The most common way of doing this is to incorporate the insights into the architectural design of the network. If, for an instance, our network works with visual inputs, we would like it to be able to exploit any local 2-dimensional patterns that may be present. We would

also like it to exhibit position invariance and other similar properties that make sense when processing images. To achieve this, we can bias the network by including convolutional layers in it.

Sensible priors can by themselves provide some of the benefits that we seek. They can, for an instance, partially reduce the amount of data required to train a model. This is certainly the case in convolutional networks, for an instance. Convolution can be interpreted as a form of weight sharing (the same neuron is applied to multiple patches). It substantially decreases the number of parameters that the network needs and thus the amount of data required to get good generalization.

Naturally, similar approaches are taken in a number of other cases too. Whenever we use recurrent neural networks, for an instance, it is most likely because we need the network to have memory. Our intuition, therefore, is that it should be able to retain relevant information over a large number of time steps. For this reason we routinely make use of LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit) cells, which are designed expressly to facilitate this.

We can use a similar approach to incorporate intuitions about how our neural network should perform aspectualization and coarsening. In the next section we will look at the concept of *visual attention*, which provides one possible way of implementing the idea.

8.2 | Visual Attention

The concept of visual attention for deep learning has been introduced in [92]. In the paper, visual attention was applied to the supervised task of image classification. The authors motivate the concept by comparing the way in which vision is handled by humans and by convolutional networks. They note that humans are very good at spotting salient parts of an image, and that they are then able to concentrate their attention even on the most minute details, provided that they are relevant.

Standard convolutional networks, on the other hand, attend to all portions of their input equally. Thus, given some limited perceptual budget, determined by the computational resources at its disposal, the network may not be able to attend to all the salient details with sufficient precision, while wasting much computation on irrelevant regions.

To address this, the authors of [92] introduce the concept of visual attention. They equip the convolutional network with additional recurrent layers and a reinforcement learning mechanism. The network is then taught to perform classification by selectively attending to different parts of the input image at different time steps.

Once the network chooses the area that it is going to attend to, the actual focusing of attention is implemented using the concept of *glimpses*. A glimpse is formed by extracting k square patches centred at location l (the location attended to by the agent). The first patch

is of size $g_w \times g_w$ and it retains the original resolution. After that, each successive patch has twice the width of the previous one. All patches, however, are rescaled to the size of $g_w \times g_w$ [92]. This means that each successive patch covers more and more pixels from the original image, but that it also comes at a correspondingly lower resolution.

8.3 | Pac-Man and State Observations

In a later section we will discuss the concept of a visual attention operator for feature maps. The approach will be explored and tested in the context of the well-known game called Pac-Man. We will therefore first discuss the game itself, describe its key mechanics and show what kinds of observations one can present to the RL agent.

8.3.1 The Game of Pac-Man

Pac-Man is a well-known arcade game, in which the player (controlling the "Pac-Man") navigates a 2-dimensional maze, trying to collect all the available pac-dots in the shortest possible time. A secondary objective is to avoid getting eaten by ghosts, which move randomly around the maze.

The game also contains special power pellets. If the Pac-Man collects a power pellet, this makes the ghosts scared for a certain amount of time. While the ghosts are scared, Pac-Man can eat them instead. Eaten ghosts will respawn at their starting locations.

In the original version of the game there were other features. The Pac-Man would, for an instance, also collect various kinds of fruit. However, we are using a simplified implementation, originally implemented for UC Berkeley's CS-188.

An illustration of a scenario from Pac-Man is given in Fig. 8.1. The Pac-Man is shown in yellow. The smaller dots are the pac-dots and the larger ones are the power pellets.

8.3.2 Possible Representations

There are three main kinds of representations that one can use in a game like Pac-Man:

- handcrafted high-level features;
- low-level feature maps;
- raw-pixel representation.

We will now briefly describe each of these options in turn.

Handcrafted High-Level Features

The typical approaches to playing Pac-Man automatically would usually have worked with features designed by hand. These would be constructed so as to provide the agent with the most relevant information a well-structured, compressed form. Such features might include e.g. [93]:

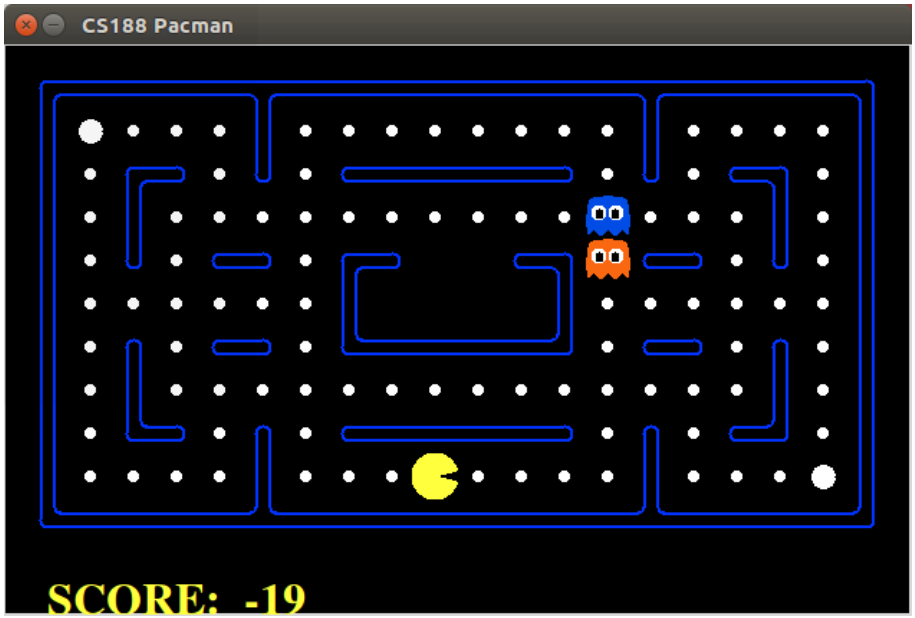


Fig. 8.1: The game of Pac-Man: a sample scenario.

- how far the closest pac-dot is;
- number of ghosts 1 step away;
- whether there is a pac-dot at the position where an action would lead the Pac-Man;
- whether the Pac-Man will collide with a ghost if it takes an action;
- ...

If the agent is provided with high-level features, which are expressive enough, it does not need to perform global generalization. In fact, it can often successfully learn the task using extremely simple approaches such as RL with linear approximation. Such approaches can be very sample efficient, because the agent only needs to learn about the task. It does not have to learn how to uncover the structure of its observations (e.g. by learning to do computer vision). The features it gets are already well-structured.

Naturally, in most applications it is extremely difficult to design a good set of features by hand. It might easily require years, or even decades of problem-specific research. This, of course, is where deep reinforcement learning is so useful: it can learn how to extract useful features automatically. This has enabled the recent considerable advances in reinforcement learning – even though such approaches require a lot of samples.

Low-Level Feature Maps

As we mentioned, the notable advantage of deep reinforcement learning is that we can feed the agent the state observation in a more or less raw format and have it figure out the relevant features by itself.

With a game like Pac-Man, a very natural representation is to use a stack of binary feature maps, where each channel will encode the presence / absence of a certain kind of

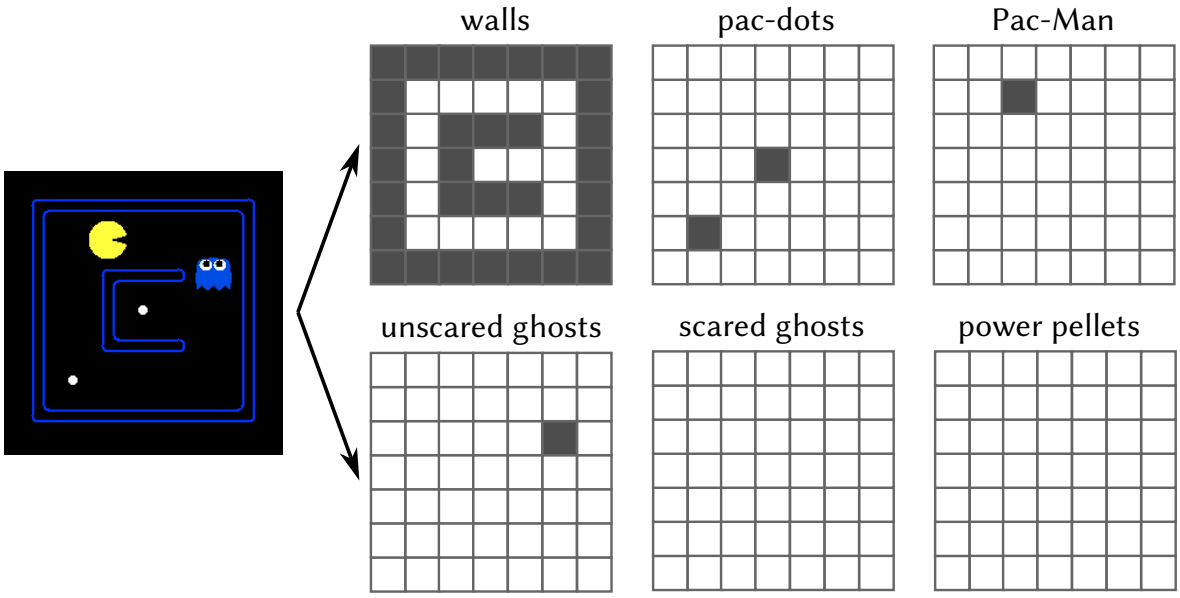


Fig. 8.2: Low-level feature maps: an illustration.

object, e.g.:

- the Pac-Man,
- the walls,
- pac-dots,
- ghosts (regular and frightened),
- power pellets
- ...

This is actually the kind of representation that we have been using in our experiments. The feature maps are further illustrated in Fig. 8.2, which shows a small Pac-Man layout and its corresponding binary feature maps.

Raw-Pixel Representations

If a representation is required, which is even simpler and makes still less assumptions about the task (such as having access to the ground-truth representation of the simulator), a raw-pixel representation can be used. In that case the raw pixels from the video game (or even from a video camera) are fed directly into the RL agent.

Deep RL agents are capable of handling such representations. They can learn how to preprocess the video so as to get good results. In recent years this has been evidenced by the results that RL algorithms exhibit when playing Atari games from raw pixels [77, 78, 94]. Naturally, there has also been a number of other, even more impressive results since.

Even though RL methods are now sophisticated enough to handle raw-pixel representations for many tasks, agents that work with such representations are necessarily much more computationally expensive than agents that make use of more structured inputs. As a rule, they are also significantly less sample efficient, because they have to learn to do at least some limited amount of computer vision before they can start to learn the actual task. This is why in our experiments we will be using binary feature maps rather than raw-pixel

inputs.

8.4 | An Attention Operator for Feature Maps

The concept of visual attention, as described in the previous section, is used to perform classification by the authors of [92]. We propose to apply a similar operator to reinforcement learning feature maps in order to perform abstraction in the sense of aspectualization and coarsening. In the following subsections we will show how the proposed approach differs from the concept of visual attention described hereinbefore. We will also provide motivations and intuitions that underlie our design in the following subsections.

The approach has been verified using the game of Pac-Man. The first set of results, which will also be discussed hereinafter, has already been published in [20] and will be reproduced here from that source with minor additions.

8.4.1 The Main Objectives of Our Design

We have considered several objectives, when designing the approach. These include especially:

- applicability to differently sized layouts;
- coarsened full observability;
- focused attention.

Differently Sized Layouts

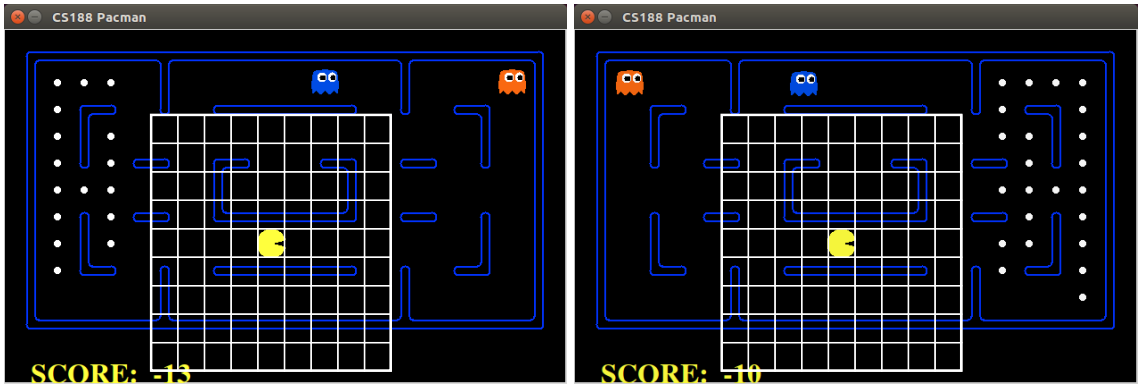
The first objective that we are going to consider is the agent's ability to perform tasks, where the dimensionality of the state observation matrix is subject to change. The game of Pac-Man is a good example of this phenomenon, because the size of the game layouts varies.

Naturally, if we are using a neural network, this has a fixed input shape. If we just feed in the feature maps as they are, we will therefore need a separate network for every single layout shape.

A simple solution to this problem would be to take the feature maps and cut out a window of some predefined size around the Pac-Man. That way, no matter what layout we use, the shape of the input will remain the same (for layouts smaller than the window, we can always apply zero padding).

Coarsened Full Observability

The problem with simply cutting out a window centred at the Pac-Man is that the Pac-Man will not be able to observe anything outside of that limited window. This is an issue, because we would ideally want to give the agent the ability to perceive the entire layout at



(a) Pac-dots to the left. (b) Pac-dots to the right.

Fig. 8.3: Pac-Man with simple cropping: 2 scenarios.

every time step. It would be okay to reduce the size of the input by applying abstraction – but probably abstraction in the sense of coarsening rather than aspectualization.

To show why discarding anything outside of a fixed-size window is a problem, we may refer to Fig. 8.3, which displays two possible game scenarios. In both scenarios we cut out a 9×9 window centred at the Pac-Man. In both cases, the observation that the Pac-Man receives is exactly the same: the entire window is clear of pac-dots. The optimal action, however, is entirely different in each case. In the scenario shown in Fig. 8.3a, the Pac-Man should obviously go left, while in Fig. 8.3b it should go right.

The problem can be solved either by providing the Pac-Man with memory (which would allow it to explore one side first and then proceed to the other), or by making sure that the state is fully observable at all times – even if only in a coarser form.

Focused Attention

It should be clear that our first two goals are somewhat mutually opposed. One is to provide a fixed-size representation no matter what the size of the layout is and the other is to maintain observability. However, both these objectives can be reconciled by applying the concept of visual attention as described in an earlier section.

Conveniently, in Pac-Man we can work under the assumption that the immediate surroundings of the Pac-Man are invariably the most relevant region, to which the agent should focus most of its attention. The intuition behind this is that keeping good track of its closest surroundings enables the Pac-Man to identify the most immediate threats (such as approaching ghosts) and opportunities (e.g. collecting nearby pac-dots). Other, more distant perceptions, may also be relevant, but they will not be more important in general.

We can use this fact to dispense with the additional actions, which we would otherwise need for the agent to control where it is going to pay attention at the next time step. It also means that we do not require the agent to use a recurrent neural architecture. The

attended-to location will instead remain coupled with the location of the Pac-Man.

The focusing of attention will itself be implemented in a way similar (though not completely identical) to what the authors of [92] suggest. The immediate surroundings of the Pac-Man will be made available to the agent at full resolution, but it will also perceive increasingly larger portions of the space at correspondingly lower resolutions.

8.4.2 The Full-Resolution Window

As already mentioned, the immediate surroundings of the Pac-Man will be cut out and fed into the neural network as a full-resolution window. As in the approach presented in [92], we will be using a $g_w \times g_w$ cut-out, which (in our case) will be centred at the Pac-Man's current position. The window will remain centred at the Pac-Man even when it approaches the edge of the layout. The overflowing cells will be filled with zeros.

8.4.3 The Lower-Resolution Windows

As per the visual attention concept, in addition to the full-resolution window, we will also feed several lower-resolution windows into the agent. Since we want to make sure, that the agent is able to perceive the entire layout (in at least some coarse form), we depart from the approach used in [92]. The size is not going to be doubled for each subsequent cut-out. Instead we are going to specify the ratio that relates the size of each cut-out to the size of the entire layout.

Unless explicitly stated otherwise, we will be using two low-resolution windows in all our experiments. The first window will contain the downscaled version of a cut-out, which is half the size of the layout. The second window will use the downscaled version of the entire layout.

The lower-resolution windows are still centred at the Pac-Man, but only weakly. That is to say, they are only centred, unless that would lead to discarding part of the layout. If the Pac-Man finds itself on the edge of the layout, the centre of the cut-out is shifted so that the cut-out lies inside the layout instead of being padded. The approach is illustrated in Fig. 8.4.

8.4.4 Downscaling of the Windows

In the previous sections, we have described how a glimpse if formed. An essential component of that process is the downscaling operator, which is applied to the cut-outs. On raw-pixel representations, it may be sensible to use subsampling procedures from standard image processing. In our case, the operator will be applied to feature maps. We will therefore use the max operator to pool rows and columns together, where necessary. In this way, we will always preserve the information about the presence of a feature in the

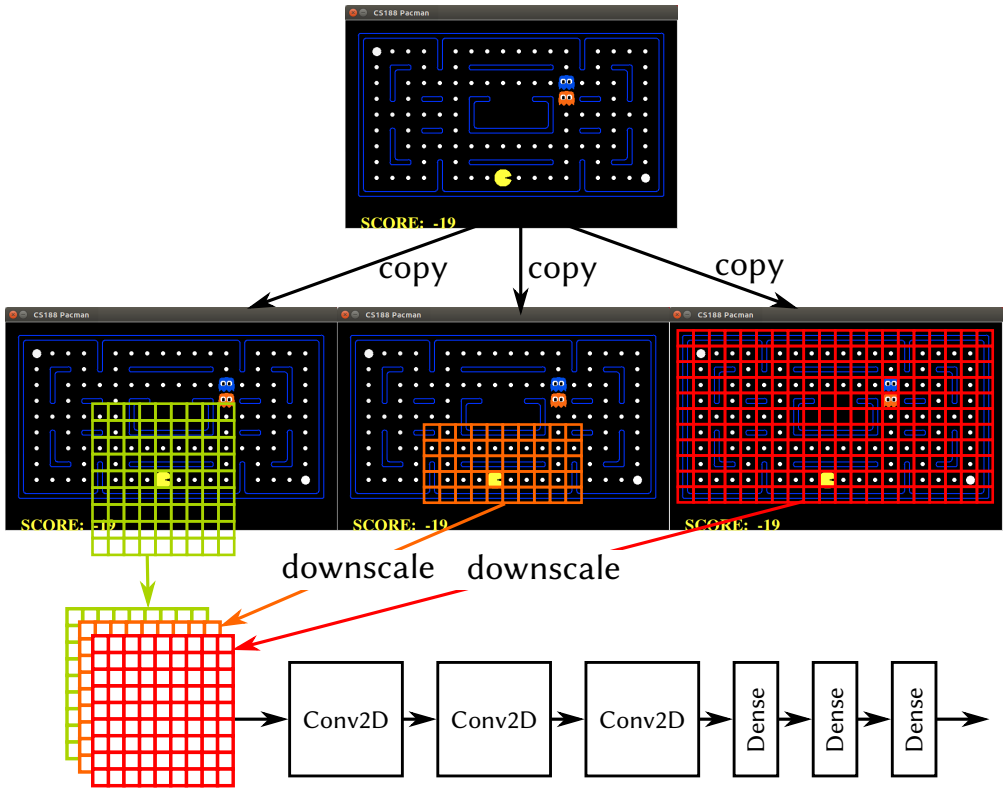


Fig. 8.4: The proposed method of forming the glimpses. The full-resolution window is centred at the Pac-Man. The lower-resolution windows are weakly centred – they always lie inside the layout and no padding is necessary. Two lower-resolution windows are used – one which covers half of the layout and the other the entire layout.

corresponding area of the original feature map.

Naturally, this is not the only option and in future work it would be a good idea to make a comparison with several other simple baselines and also with an approach that would learn how to perform the downsampling by itself.

8.5 | The Experimental Setup

In this section, we will describe the experimental setup used to test and verify the proposed attention operator. Let us start with the methods, tools and parameters that we have employed and then proceed to describe the way in which the actual experiments are structured.

8.5.1 Tools and Methods

We use the UC Berkeley Pac-Man environment for Python [95]. We have implemented bindings, which provide it with an OpenAI Gym-like [96] interface. The RL algorithm that we use it the DQN as implemented in the keras-rl repository [97]. The DQN has some known failure modes and it is not the most stable algorithm. However, as an off-policy method, it generally has better sample-efficiency than on-policy methods such as PPO.

In the light of more recent advancements – soft actor-critic might be an even better choice. However, at the time that this was implemented, it had not been developed yet. Also, the current implementations of SAC that we have inspected only work well with continuous action spaces and would need to be partially rewritten before they could be applied to our problem. Consequently, we leave experimentation with SAC for future work.

The neural network architecture that we use for the DQN is based on that found in repository [98]. Its details are as follows:

- A 2D convolutional layer with 16 filters and a ReLU (rectified linear unit).
- A 2D convolutional layer with 32 filters and a ReLU.
- A flattening operation.
- A dense layer with 256 units and a ReLU.
- A dense layer with as many units as there are actions.

The kernels of all the convolutional filters are 3×3.

In one of the tests we have also used a slightly larger version of the same network. This larger version has the following parameters:

- A 2D convolutional layer with 48 filters and a ReLU.
- Two 2D convolutional layers with 64 filters and ReLUs.
- A flattening operation.
- Two dense layers with 256 units and ReLUs.

- A dense layer with as many units as there are actions.

In keeping with [98], we also shape the rewards according to the following rules:

$$r_s = \begin{cases} 50 & r > 20 \\ 10 & r \in (0, 20] \\ -500 & r < -10 \\ -1 & r \in [-10, 0) \\ 0 & r = 0 \end{cases} \tag{8.4}$$

where r_s is the shaped reward and r is the raw reward. If the reward goes below -10, we also consider the episode failed. In the charts, however, the original, raw and unshaped rewards are shown.

We use the mean squared error as a loss function and Adam [99] as the optimizer. The number of RL training steps is set to $50\,000$. We use an annealed ϵ-greedy policy, which goes from the value of $\epsilon = 1.0$ to $\epsilon = 0.1$ in $10\,000$ steps and uses $\epsilon = 0.05$ for the testing phase. The size of the replay memory used by the learning algorithm is $50\,000$ steps. For discounting, we use $\gamma = 0.95$.

Since the Pac-Man cannot walk through walls, not all actions are legal in every state. To take legality into account, we have developed a policy, which queries the environment to determine, which actions are illegal and decreases their Q-values to $-\infty$. It would also have been possible to merely replace any illegal action the Pac-Man chooses with the stop action. An approach that is not very useful, however, is to replace the illegal action with a randomly selected one. We have attempted this at first with very discouraging results. This is likely due to the fact that introducing a strongly stochastic element of this kind into the environment's dynamics makes the task considerably more difficult to learn.

8.5.2 The Structure of the Experiments

To account for the stochasticity present in the game, once we train an agent, we evaluate its performance on 100 testing episodes and average the results. Since the training procedure itself is stochastic too, we also repeat the entire training 50 times for each method. The results are again averaged.

We experimented with the following methods:

1. The attention operator (as described hereinbefore).
2. The result of simple cropping (only the full-resolution window centred at the Pacman is used, the low-resolution windows are not).
3. A full-observation version, where the entire feature maps are presented to the network without any cropping.
4. A full-observation version with a larger network (the architecture described in the

previous section).

The performance metrics, which we evaluate, include:

- The *learning curves*, which show how much reward the agent is able to accumulate in the course of a single episode as training progresses.

  These results are naturally very stochastic and since every episode may take a different number of steps, they are not well aligned in time. In order to average the results and produce interpretable curves, we combine data from all the runs and compute a moving average.
- *Testing-phase episode rewards*, which show how much reward the agent is able to accumulate in the course of a single episode on average in the testing phase (these results are averaged first over the evaluation runs and then also over the learning runs, so that we get 1 value for each method).
- *Testing-phase victory rates*, which indicate what portion of testing-phase episodes each agent has won.

8.6 | The Results

In the previous section, we have described some specifics concerning our experimental setup. In the present section, we will present the results achieved by the four aforementioned methods. These will be inspected through the prism of the three performance metrics and briefly commented.

8.6.1 The Learning Curves

Let us start with the learning curves, which are shown in Fig. 8.5. As mentioned above, the results from all the runs were combined into a single series, which was then smoothed using the moving average operator. The shaded areas around the curves represent t-based 95% confidence intervals, computed over the moving windows.

In order to get smoother curves, we have applied a rolling window of size 300. This means (roughly) that in addition to averaging over all the 50 runs, we also apply some amount of smoothing over adjacent time steps. However, a rolling window of 50 was used to compute the confidence intervals, which would otherwise have been too optimistic (we did not apply this approach in [20] – however, it was still possible to make judgements about the confidence using the rest of the plots).

It is clear that the version with the attention operator performs best. The version observing the entire layout (full obs., larger net) – in addition to not being usable for layouts of different dimensions – also has the worst performance.

The remaining two methods (the one, which does simple cropping and the one, which take in the entire layout, but uses the larger neural network – full obs., larger net) have

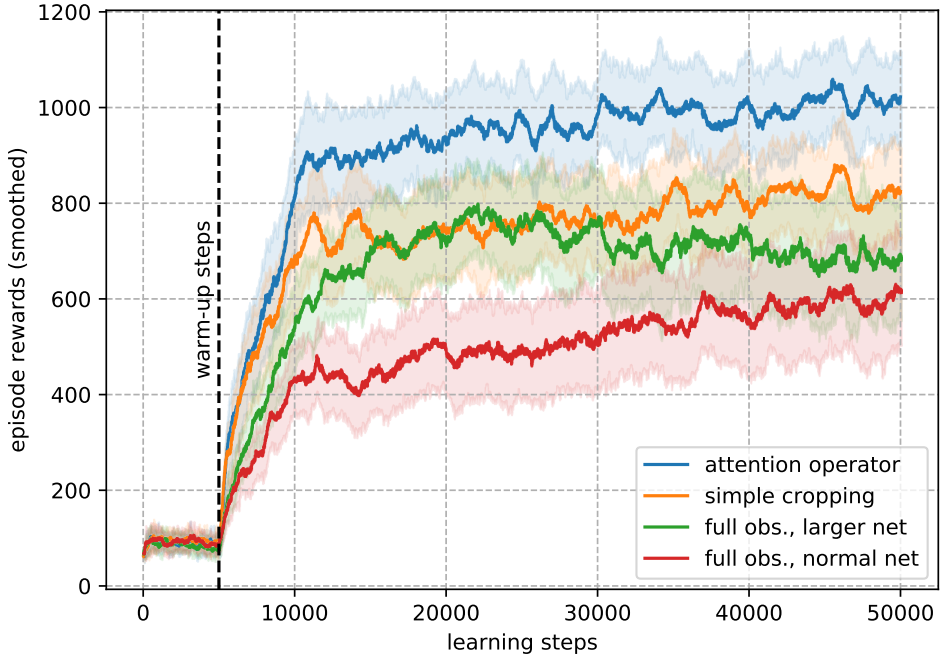


Fig. 8.5: The learning curves (episode rewards over time).

very similar performance

Note also that no learning happens for the duration of the first 5000 steps – these are the warm-up steps, during which we collect the data necessary to fill the replay buffer with some initial samples.

8.6.2 Testing-Phase Episode Rewards

During learning, the agent needs to perform exploration, which is why its policy will have stochastic elements, even though DQN is fundamentally about deterministic policies. It will therefore be useful to also compare the agents in terms of their testing-phase performance, i.e. to test the final policy with exploration turned off. We do this comparison by looking at the mean episode rewards that each agent has achieved. The results are shown in Fig. 8.6.

As we can see, the results correspond very well with the conclusions that we have drawn from the learning curves. The attention operator still dominates, simple cropping and the full-layout version with the larger net are very close to each other and the full-layout version with the standard net is the weakest.

8.6.3 Testing-Phase Victory Rates

To complement the testing-phase results, we may also want to compare the methods in terms of their victory rates. The victory rate is defined as the ratio of wins to the total number of testing-phase episodes. The results in Fig. 8.7 indicate that the attention operator method is successful in slightly less than half of the cases and simple cropping in approximately 22% of cases. The full-layout version with a standard net again achieves the

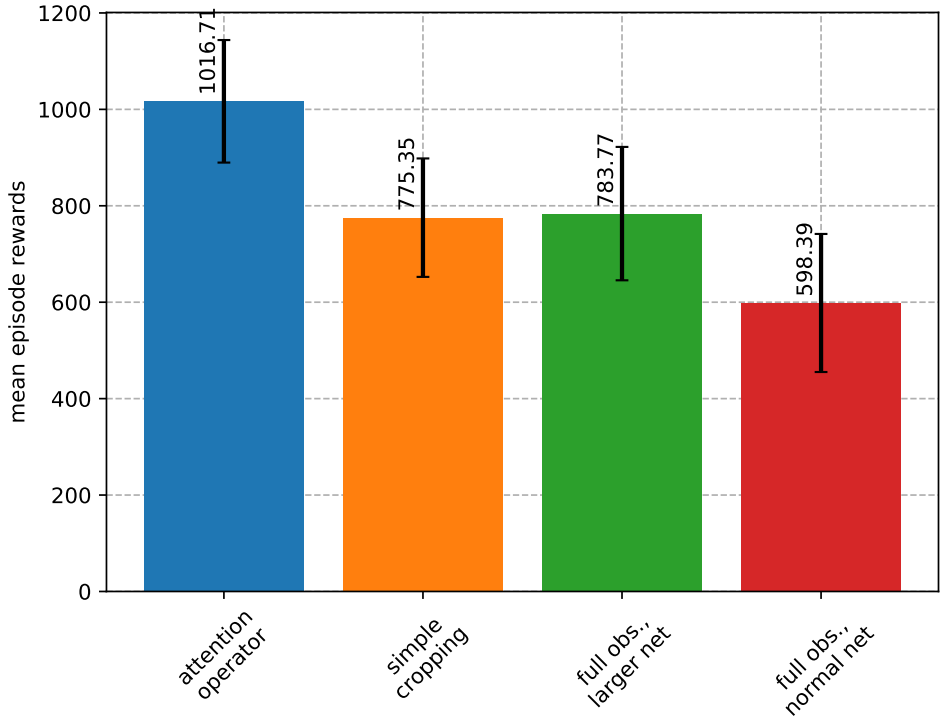


Fig. 8.6: Testing-phase mean episode rewards with t-based 95% confidence intervals.

worst results, winning just about 5% of the games.

The surprising thing, however, is that while the average rewards of simple cropping and the full layout version with the larger net were roughly the same, the victory rate of the latter is much worse – not even 8%. We may conclude from these observations that the attention operator clearly outperforms the other approaches, but also that there is still a lot of room for improvement. A victory rate of just about 50% is still not a very impressive achievement.

8.7 | Ways to Extend The Approach

While the approach presented hereinbefore brings about improved performance, there is a number of ways in which it could still be improved and extended. The most obvious ways include especially:

- Doing proper architecture design and hyperparameter tuning.
- Testing other RL methods, such as DQN with prioritized experience replay, or – better still – SAC.
- Using attention inside the neural architecture, not just at the input.
- Testing whether and to what extent the proposed approach actually generalizes to layout sizes with which the agent was not trained and how far it is possible to scale the layout without significantly damaging the performance of the agent.
- Make the approach more general by:

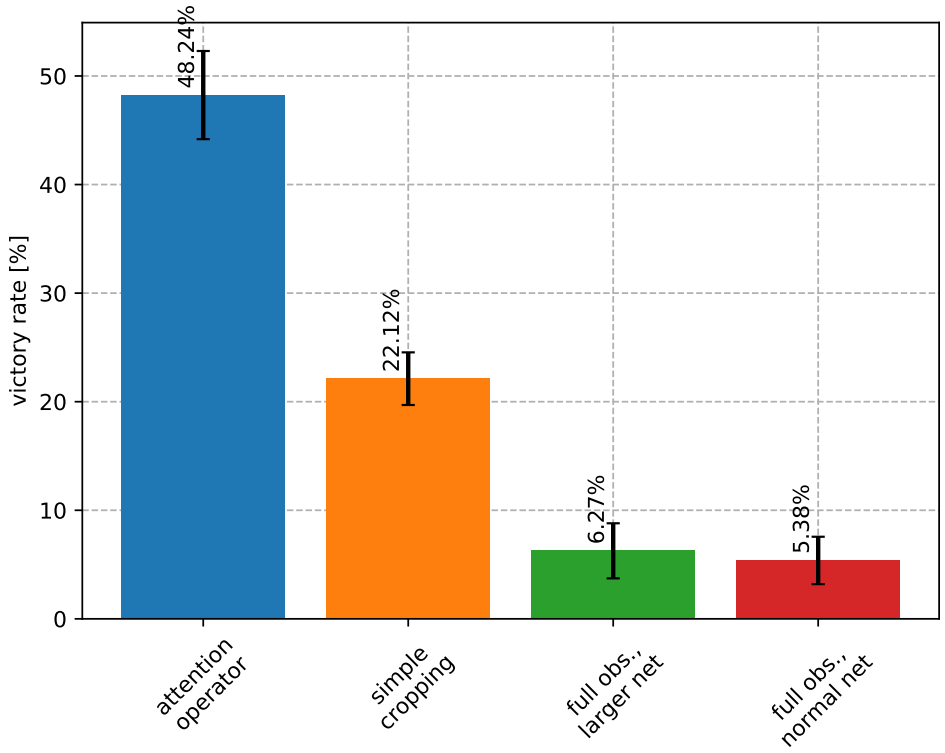


Fig. 8.7: Testing-phase victory rates with t-based 95% confidence intervals.

- Providing the agent with the ability to learn its own task-specific way to effectively downscale the input.
- Providing a version, where attention is not coupled to the position of the agent – which might be useful in some cases.

It is possible that some of these steps – architecture and hyperparameter tuning and different RL methods in particular – would yield significant performance improvements.

8.8 | Learned Visual Summarization

This section will document some preliminary work that we did concerning learned visual summarization: i.e. automatically learning an effective domain-specific way to downscale the inputs. Our results have been negative so far, we will therefore also outline some suggestions as to why this might be and how it might be possible to work around the issues in the future.

How is it possible for the agent to learn the downscaling? When producing glimpses before, we downscaled the input using the max operator. This made some intuitive sense in the context of feature maps, but for many other representations it might not be a reasonable thing to do. Even for binary feature maps, it is not necessarily true that this is the best approach. Ideally, we would therefore prefer the network to find its own way to summarize the relevant information contained in the input and to downscale it to the required size.

8.8.1 Downsampling by Stacking Transformation Cores

For this reason we have decided to design an approach for learned visual summarization, which would be able to learn how to pick out the most relevant content from an input of an arbitrary size and to compress it into an output of a smaller predefined size.

We have designed an automatic procedure that will stack some predefined transformations so that the desired compression ratio is achieved. The procedure is based on a transformation core $T(\mathbf{x}, W, \theta)$, where $\mathbf{x}$ is the input tensor, W is a weight tensor and θ is a parameter vector. When the transformation cores are stacked, the same weights are shared among all instances of the core.

The core can consist of as little as a single convolutional layer, but it can also be implemented using a deeper architecture. Vector θ contains parameters that control how aggressively the transformation core downsamples its input. For a single-layer convolutional core, this may involve parameters such as stride or dilation. Note that the transformation core needs to be implemented using a deep learning framework, so that it forms a part of the overall neural architecture and all the weights can be learned end-to-end.

Under this approach, the downsampling architecture is assembled as follows (we will first consider the 1-dimensional case):

1. *The fast downsampling phase:* Transformation cores T are stacked on top of each other, with dilation rates increasing according to 2^{i-1}, where $i = 1, 2, 3, ...$ is the number of the core in the succession of cores. This process stops just before the size of the output gets below the target value.
2. *The slow downsampling phase:* In this phase, more transformation cores are stacked, but the dilation rates are decreasing – to progressively get as close to the target size as possible.

For multi-dimensional inputs, a separate downsampling plan is first formed for each dimension. Once all the plans have been formed, they are all padded to the length of the longest one. When a point is reached, where the input tensor does not need to be downsampled along a certain dimension any more, the dilation rate for that dimension is fixed to 1 and the convolutional kernel is sliced so that it has the size of 1 in that dimension. That way, we stop downsampling along that dimension and continue only with the remaining ones.

Naturally, this is not the only way to setup the architecture. It would make sense to explore other procedures in the future. It is possible, for an instance, that the fast downsampling phase is too aggressive. The slicing of the convolutional kernel might also need further thought. It could also be interesting to experiment with ideas from architectures such as ResNet [100] and DenseNet [101].

8.8.2 Experimental Setup with Learned Visual Summarization

The experimental setup for our experiment with learned visual summarization will, for the most part, be identical to the one that we have used in our original experiments. The main difference is in the algorithms – in addition to our original attention operator, we will also be experimenting with several versions of our new learned visual summarization – namely:

- A single-layer transformation core (denoted with LVS, 1L, sharing in the plots);
- A 2-layer transformation core (denoted with LVS, 2L, sharing in the plots).

Normally, when the transformation cores are stacked, the same weights are shared among all instances of the core. However, we have also carried out an experiment, where the weights were not shared – we include it as a further baseline:

- A single-layer transformation core without weight sharing (denoted with LVS, 1L, no sharing in the plots).

To add further information: Our transformation cores use convolutional layers with 3×3 kernels. The dilation rate is determined by the procedure outlined in the previous section. Another parameter that can change is the number of convolutional filters in the transformation core's layers. We will report this for each configuration as we present the results.

8.8.3 The Results with Learned Visual Summarization

The results will again be compared in terms of the same plots that we have used before. Fig. 8.8 shows a comparison of the learning curves for the new methods. We also include the curve for the standard attention operator and for simple cropping. This is for the sake of comparison.

As we can see, with learned visual summarization, the learning proceeds more slowly at the beginning. This effect is less pronounced in the case of a single-layer transformation core with weight sharing, and much more severe in the case of the other two variants: the multi-layer core and the single-layer core without weight sharing. The reason for this is fairly obvious: these versions have a lot more parameters.

The testing-phase mean episode rewards are shown in Fig. 8.9. What we can see is that the mean rewards are surprisingly similar for all methods other than the attention operator, which performs significantly better.

As far as the testing-phase victory rates are concerned, simple cropping performs the worst, while the original attention operator clearly dominates all the other approaches.

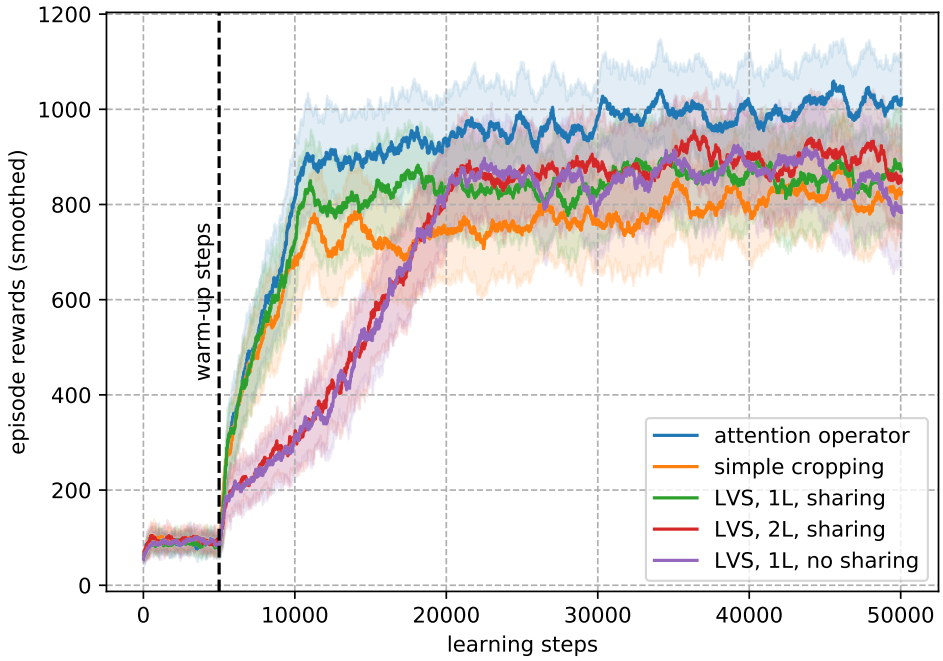


Fig. 8.8: The learning curves (episode rewards over time) with and without learned visual summarization.

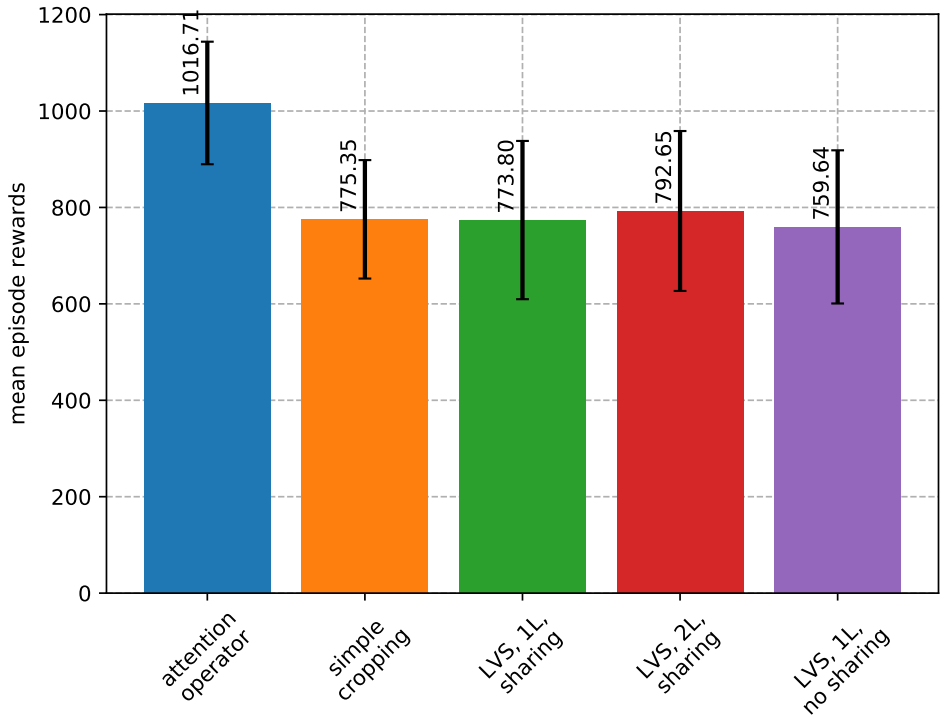


Fig. 8.9: Testing-phase mean episode rewards with t-based 95% confidence intervals – with and without learned visual summarization.

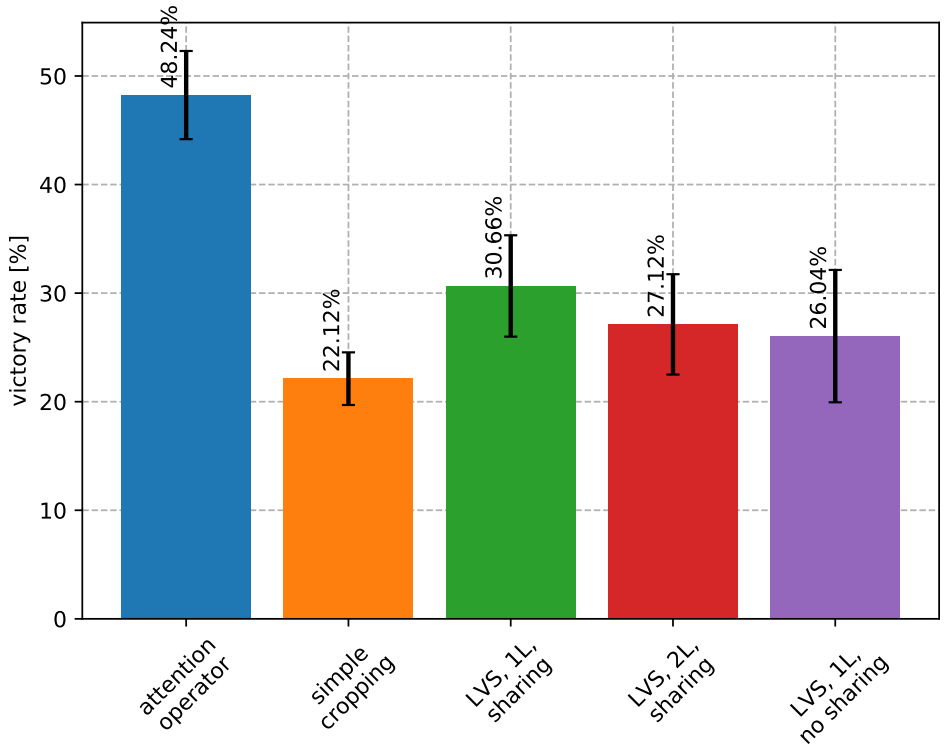


Fig. 8.10: Testing-phase victory rates with t-based 95% confidence intervals – with and without learned visual summarization.

8.8.4 Discussion

The results presented in the previous section are preliminary – the findings should not be regarded as conclusive. It is not clear what causes this suboptimal performance. One factor that must clearly contribute is the need to learn additional parameters. The predefined downsampling procedure based on max, which we have used in the original visual attention operator does not have any tunable parameters; it does a sensible thing from the beginning – even though its behaviour might not be quite optimal.

In contrast to this, with learned visual summarization, the network has to learn how to effectively downscale by itself – its initial behaviour will be random. However, it remains unclear, why the DQN has not been able to learn a good way to downscale the input even given a relatively large number of time steps.

This could, perhaps, be related to some property of the DQN algorithm. It is possible, for an instance, that the replay buffer becomes dominated by samples that do not contribute to performance improvements and that drawing random mini-batches from it is no longer effective.

Naturally, this is to be taken as pure conjecture at this point. However, the hypothesis could be tested by replacing standard experience replay with prioritized experience replay, or by using a different RL method altogether (this latter approach would also rule out that

the problem is caused by DQN's weak convergence properties or by some quirk in the particular implementation that we are using). This remains to be tested, of course.

Even if this problem can be alleviated, learned visual summarization can still be expected to delay significant increases in performance until the network has learnt a reasonable way to downscale. This would then tend to make the approach less sample-efficient. This effect could, perhaps, be eliminated, if the weights for this part of the network were initialized so that it does something sensible from the beginning.

The learned downscaling could, for an instance, be initialized to downscale the input by computing noisy averages (the weighting of the average would not be quite uniform so as to prevent weight symmetries, which cannot easily be broken by subsequent learning). Whether this would help significantly and whether it might perhaps constrain the network too much in terms of solutions that it is likely to discover, also remains to be tested.

CHAPTER 9

MULTISENSORY DEEP REINFORCEMENT LEARNING

One of the strong advancements brought about by the introduction of deep neural networks into reinforcement learning is the ability to easily use and combine various multisensory inputs. Before deep RL, it was very unclear how to effectively preprocess and fuse several heterogeneous streams of data.

With deep learning, data preprocessing no longer needs to be designed by hand. However, we are still able to use our prior knowledge about how the preprocessing should be structured – we can consider these aspects in our architecture design. This creates excellent conditions for creating RL agents, which fuse and then utilize several distinct streams of data.

We will first discuss how this can be done effectively in general terms. We will then also present a case study, where a reinforcement learning agent will be supposed to reduce the psychoacoustic annoyance in a simulated vehicle. The agent will be provided with 2 different types of data: structured tabular data and an audio stream.

The case study is from a project that I have participated on during my visiting scholar appointment at UC Berkeley. Several other researchers have worked on the same project, including especially *Erickson R. Nascimento* and *Edson Roteia* from Universidade Federal de Minas Gerais, Brazil, *Ismael Villegas* from the University of North Carolina, *Isabella Huang*, *Cole Rose* and *Gregorij Kurillo* from UC Berkeley and the head of the group: *Ruzena Bajcsy* from UC Berkeley.

My role in the project was to design and test a new deep reinforcement learning controller, but I have also participated in data collection, analysis and visualization of the collected data and in other tasks. The results of our research are presented more fully in [102, 103]. Here I will only show a small portion of the material that is most directly relevant to this chapter's topic.

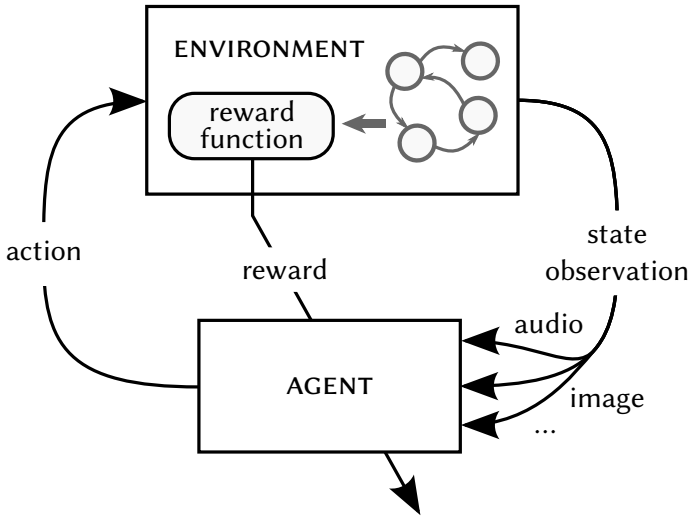


Fig. 9.1: Reinforcement learning with multisensory observations.

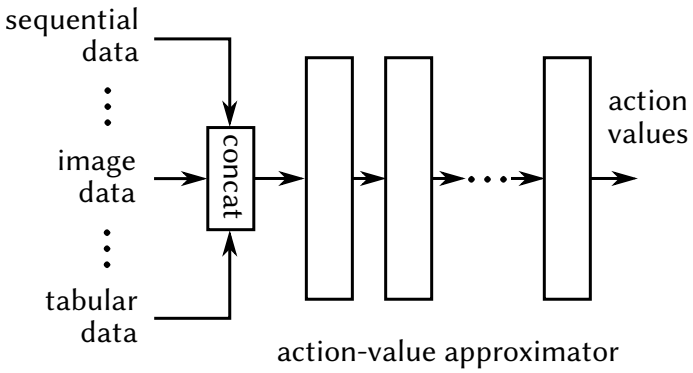


Fig. 9.2: A naïve architecture for multisensory deep RL.

9.1 | RL with Multi-Sensory Observations

In many practical tasks, the observations of the agent will be composed of several data streams, as illustrated in Fig. 9.1. The data may be of different kinds – such as images, audio, tabular data, natural language and others. Additionally, there may be more than 1 stream for each kind of data.

One of the best-known advantages of deep learning is that it can learn to preprocess data automatically. However, the architecture design needs to be sensible, because it predetermines the way it which the data is going to be approached. We could say that the architecture of the network introduces a certain kind of cognitive bias, which can have both – positive and negative effects.

As an example of a negative kind of architecture-related cognitive bias from the multisensory domain, we may present Fig. 9.2. At the first sight, the figure might seem correct – all the input data is first concatenated and then thrown at a deep neural network. If neural networks can learn their own preprocessing, why would this be a problem?

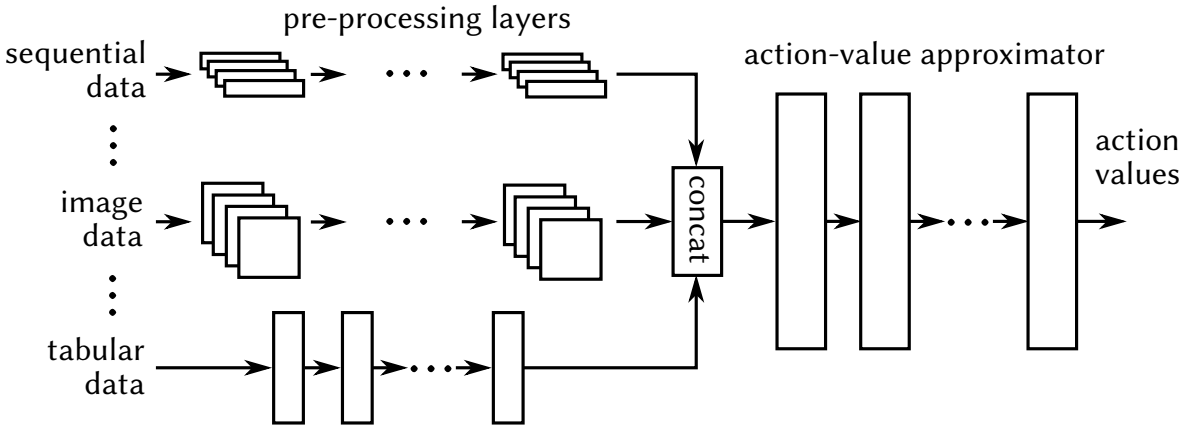


Fig. 9.3: An architecture with a separate preprocessing sub-network for each data stream.

The problem, of course, is that different kind of data need different types of preprocessing and the architecture should reflect this. For images, for an instance, we routinely use convolutional layers, which incorporate our prior beliefs on how visual information needs to be processed. While it would in principle be possible for a sufficiently large neural network without convolutional layers to learn the same behaviour, it would take much more parameters and a vast amount of data.

There is a second problem as well. The sizes of the different data streams are very different. If we take even a small image of, say 64×64 pixels, we will already end up with a 4096-dimensional input vector. If we naïvely concatenate this vector with a 5-dimensional vector that contains tabular data, the additional columns can easily get drowned out by the visual inputs. It would take a vast amount of data for the network to learn that these additional dimensions contain valuable information.

9.1.1 Content-Specific Preprocessing Sub-Networks

All these considerations lead us to the conclusion that a sensible architecture should first apply some suitable preprocessing to each input stream – such as 2D convolutional layers for images, 1D convolutions or attention layers for sequential data, recurrent layers, if memory is required and so on. Only once the inputs have been sufficiently preprocessed and their dimensionality appropriately reduced, should they be combined. This principle is illustrated in Fig. 9.3. If there are multiple streams with the same kind of data, it can also be advantageous to share parameters among their respective preprocessing sub-networks.

9.2 | Reducing Psychoacoustic Annoyance: A Deep-RL-based Feasibility Study

Next we are going to present results form the feasibility study that I participated in at UC Berkeley. These are divided into two parts. The first part will present the experimental setup and the results from our simulation experiment. The second part will present a portion of the results from our analysis of real driving data collected during our real-world experiments. These will aid us in interpreting the results of the first part and provide additional justification for our conclusions.

Our study considers the various psychoacoustic influences that affect drivers. Nowadays a lot of commendable effort is being invested into developing fully autonomous systems and this trend has long since taken root in the area of autonomous driving. With the recent advancements in artificial intelligence and machine learning, the field has reached promising results and several brands of autonomous cars are now on the roads for testing.

Nevertheless, sufficient safety guarantees cannot always easily be provided for these new approaches and even if they were, it seems quite clear that for the foreseeable future, artificial systems will remain in interaction with humans when it comes to driving. However, this in itself opens the door to another field of research, which too involves artificial intelligence and machine learning: that of semi-autonomous systems designed to aid human drivers. Such systems can, for an instance:

- provide feedback and warnings;
- detect lack of concentration, drowsiness, ...
- make the environment in the vehicle more pleasant and less distracting;
- help reduce stress;
- ...

A number of such systems have been explored in the past, but most of these did not consider acoustic influences on drivers at all. Even if some studies did take sound into account, it was to measure how various sounds affected the driver, but not how one could actively handle the acoustic influences on the driver in order to improve pleasantness, decrease annoyance and so on (a review of some prior works is given in [102] – we will not reproduce it again here, because it is not really relevant to multisensory RL).

Learning to Reduce Psychoacoustic Annoyance

The idea that we intended to test in our feasibility study was whether it might be possible to reduce the psychoacoustic annoyance experienced by the driver (as evaluated using an already established indicator) by taking simple actions such as turning the air conditioning on and off, opening and closing the windows, or changing the speed of the car.

The problem of reducing psychoacoustic annoyance is not trivial. In practice, the human driver perceives a mixture of sounds coming from a wide variety of sources. In order to effectively counteract annoying acoustic events, an automatic system will typically need to be able to tell them apart and to form some idea of what their source is, if that is required in order to suppress them. It is also neither necessary nor useful to suppress all sounds without distinction, because that would make the environment in the car monotonous, which may lead to sleepiness and a decrease in concentration. Furthermore, some sounds are highly informative and they should definitely not be suppressed. However, filtering sounds based on their information value is a concern that our work does not currently address.

Let us now address in turn all the respective points regarding the necessary background work, our experimental setup and the results themselves.

9.2.1 Our Simulation Environment

The core of the simulation environment for our feasibility study is formed by the well-known video game Grand Theft Auto V (GTA V) by Rockstar Games. Since this piece of software has originally been designed as a video game, it seems – at the first glance – that this calls for some justification. However, in the recent years, GTA V has been used as a simulation environment for driving, traffic modelling and other related problems quite extensively by other researchers. There is, of course, a number of other traffic and driving simulators and some of them even come with significant advantages.

The main reason why some projects use GTA V in particular, is that – being built as video game – it has a level of complexity and realism, which other simulators lack. This is not only true of the graphics, but especially of other aspects, such as:

- realistic pedestrian modelling;
- realistic traffic rules;
- the ability to simulate both day and night scenarios;
- different kinds of weather;
- the ability to render convincing audio;

the last item being the most critical one for our research task. As it turns out, dedicated simulators tend to abstract from all of these aspects to various extents.

To interface with GTA V, we used a customized version of the DeepGTAV interface for Python [104]. Naturally, it was necessary to add several additional features to the protocol, such as the ability to open and close windows, enable or disable the air conditioning and such. Some acoustic effects are even achieved by mixing external sounds with the game sound – half of the time, for an instance, we introduce the sounds of bells and beeps to ensure that there is more noise in the environment. These external sounds are muffled using a low-pass filter when the windows of the car are closed.

Fig. 9.4: A screenshot from the simulator.

Finally, we have wrapped the entire environment in and OpenAI gym interface so as to make its interaction with existing RL libraries easier.

9.2.2 The Reinforcement Learning Task

The structure of our reinforcement learning task is shown in Fig. 9.5. Let us now describe the standard components of the task – most notably the reward function, the action space and the state space.

Observations

In our task the actual state space involves all the game variables that give rise to what the agent experiences. However, this state is not fully observable to the agent, which is only informed about a small subset of these variables. At every time step it receives the following:

1. the normalized car speed (the current car speed divided by the maximum speed – set to 35 mph in our experiments);
2. the state of the window (open/closed);
3. the state of the air conditioning (on/off);
4. a recording that captures the last 4 seconds of the sound (191 488 samples; captured after the current action has been successfully applied).

This makes the dimensionality of the observation vector to be 191 491 in total.

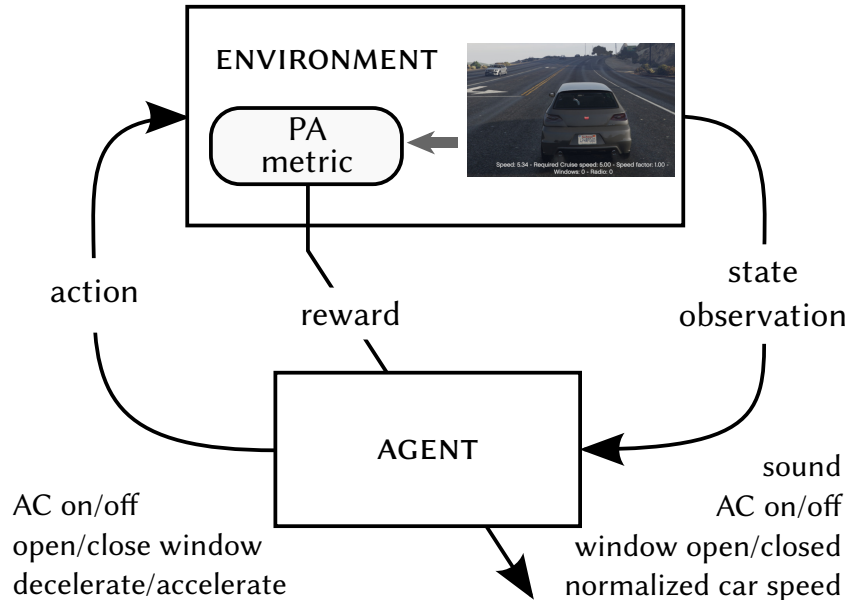


Fig. 9.5: The structure of the RL task posed in our feasibility study.

Due to the partial observability, we should actually not formulate our problem as an Markov decision problem (MDP), but rather as a partially observable Markov decision problem (POMDP). However, we choose not to address the partial observability expressly and we use the same framework that we would use for a regular MDP. By doing this, we make the assumption that the observations made available to the agent will still be sufficient for it to select reasonable actions. This is common practice and still works reasonably well for many tasks. For some tasks it might be advantageous to choose a more complex approach instead – using a recurrent neural network architecture is another common choice and it can sometimes help.

The Action Space

The action space has three dimensions and there are three possible actions for each:

- window state: no change, open, close;
- radio state: no change, on, off;
- speed: no change, accelerate, decelerate.

The Cartesian product along these three dimensions results in a discrete action space with 27 distinct actions. The possible choices for speed are $\{5, 10, 15, 20, 25, 30, 35\}$ – actions accelerate and decelerate make a step to the left or to the right in the list respectively.

Once an action is chosen, the simulation is run until the action has been properly applied, i.e. the window state, the radio state, and the speed are all as prescribed. Afterwards the agent is presented with a new observation and decided upon the new action to take.

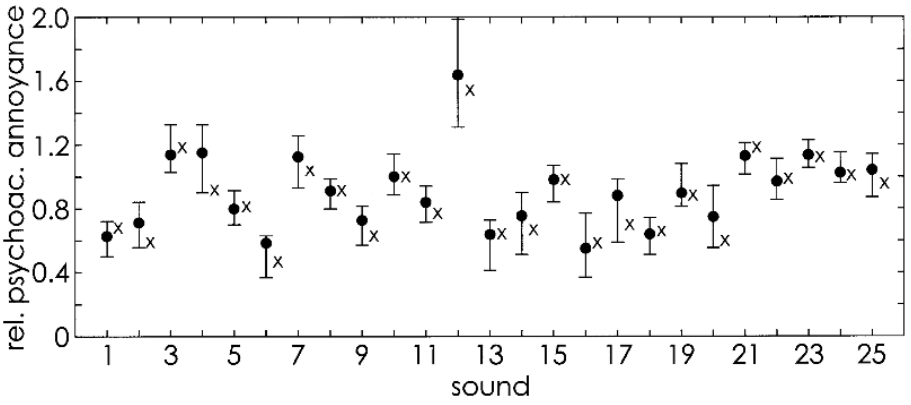


Fig. 9.6: The relative psychoacoustic annoyance of different car sounds [105]

.

The Reward Function

We define our immediate reward as being a function of the psychoacoustic annoyance metric, which will be discussed briefly in the next section (9.2.3), i.e.:

$$r_t = r(s_{t-1}, a_t, s_t) = f(PA_t), \tag{9.1}$$

where PA_t is the psychoacoustic annoyance metric at time step t and $f(\cdot)$ is the shape function

$$f(x) = 1 - (x/MAX_{PA})^{0.4}. \tag{9.2}$$

MAX_{PA} is the maximum acceptable value for PA. In our experiments we used $MAX_{PA} = 27$.

9.2.3 Psychoacoustic Annoyance

Psychoacoustic annoyance (PA) is a metric devised by Zwicker and Fastl and described in [105]. It can be used to estimate how any particular sound is going to affect a human listener: namely, how annoying it is going to be. The metric is based on empirical evidence, gathered in a psychological experiment with human subjects. Table 9.1 and Fig. 9.6 display a sample of the data used to calibrate the metric. Table 9.1 shows the different kinds of car sounds considered and Fig. 9.6 displays the different values of relative psychoacoustic annoyance as perceived by the human subjects. The PA metric is not personalized – we could say that it averages over different human preferences.

PA is defined in terms of several different psychoacoustic indicators, namely:

- Fluctuation strength: An indicator that considers changes in the acoustic signal with modulation frequencies of 20 Hz or less. In that range, the listener can clearly hear the volume of the modulated sound coming up and down.

Table 9.1: The various types of car sounds used to calibrate the PA metric [105].

Sound	Motor	Speed in km/h	Gear	Distance in m
1	Diesel	60	3	7.5
2	Otto	30	1	7.5
3	Diesel	70	2	7.5
4	Otto	0	idle	0.9
5	Otto	80	3	7.5
6	Otto	80	3	15
7	Diesel	35	1	7.5
8	Diesel	70	4	7.5
9	Otto	50	2	7.5
10	Diesel	110	4	7.5
11	Diesel	acceleration	1	7.5
12	Diesel	racing start	1	7.5
13	Otto	60	3	7.5
14	Diesel	30	1	7.5
15	Otto	50	2	3.5
16	/	60	coast	7.5
17	Diesel	0	idle	7.5
18	Diesel	60	4	7.5
19	Otto	ISO 362	2	7.5
20	Diesel	0	idle	7.5
21	Diesel	80	3	3.75
22	Diesel	50	2	3.75
23	Diesel	80	3	3.75
24	Diesel	90	4	7.5
25	Otto	80	3	3.75

According to [105], the following holds for fluctuation strength:

$$F \approx \frac{\Delta L}{4\,\mathrm{Hz}/f_{mod} + f_{mod}/4\,\mathrm{Hz}}, \tag{9.3}$$

where ΔL is the masking depth (which is, roughly speaking, equivalent to the perceived modulation depth) and f_{mod} is the modulation frequency.

Even though this definition does not seem to be too complex, to estimate the fluctuation strength for an arbitrary signal is quite challenging.

- Roughness: With modulation frequencies much higher than $20\,\mathrm{Hz}$, fluctuation is no longer perceived. What human listeners perceive instead is a certain roughness of the acoustic signal. Roughness is perceived at modulation frequencies of approximately up to $300\,\mathrm{Hz}$. The following holds for roughness [105]:

$$R \sim f_{mod}\Delta L. \tag{9.4}$$

  This seems to indicate that roughness is linear in the modulation frequency. However, ΔL also has a (non-trivial) relationship with the modulation frequency, because the perception of modulation depth depends on frequency.
- Sharpness: Indicates how much high-frequency content a sound has.
- Loudness: Models the human perception of how loud a sound is. It is well-known that human perception of sound volume is also non-linear.

With these indicators, the psychoacoustic annoyance can be computed using the following equation [105]:

$$PA = N_5 \left(1 + \sqrt{\omega_S^2 + \omega_{FS}^2}\right), \tag{9.5}$$

where N_5 is the 95th percentile of loudness and

$$\begin{aligned} \omega_S &= \begin{cases} -(S - 1.75)\log(N_5 + 10), & \text{if } S > 0, \\ 0, & \text{otherwise}, \end{cases} \\ \omega_{FS} &= \frac{2.78}{N_5^{0.4}}(0.4F + 0.6R). \end{aligned} \tag{9.6}$$

Symbols R, F, and S correspond to roughness, fluctuation strength and sharpness respectively.

Further details of how psychoacoustic annoyance is computed are actually quite complex. For an instance, some of the indicators are computed using the so-called Bark scale, which is based on a band of filters much like the Mel scale used to compute the Mel-frequency cepstral coefficients used in speech processing. We will therefore not discuss the implementation details in any further depth and we will instead refer interested readers to [102, 105].

9.2.4 Methods and Hyperparameters

In all subsequent experiments we have used double DQN as the reinforcement learning method. Our code is based on the implementation in the Keras-RL repository [106].

We have experimented with several architectures for the deep Q-network – their configurations are illustrated in Fig. 9.7. As we can see, the following baselines have been considered:

- A random agent (Fig. 9.7a);
- A multi-layer perceptron (MLP) that only takes in the tabular data, but not the audio (MLP w/o audio; Fig. 9.7b);
- A multi-layer perceptron that takes in both: the audio signal and the tabular data, but does not preprocess the audio data specially (MLP w/ audio; Fig. 9.7c).

These have been compared to the following two architectures, which take in both the tabular data and the audio, but apply different types of preprocessing:

- The audio is preprocessed by a 1-D convolutional neural network (1D CNN w/ audio; Fig. 9.7d);
- The audio is transformed into a spectrogram and processed using a 2-D convolutional neural network (2D CNN w/ spectrogram; Fig. 9.7e).

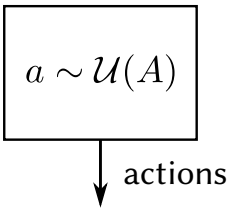
The architectures of the individual components are displayed in Fig. 9.8. The figure shows the number and type of layers as well as their parameters.

As far as learning was concerned, in all cases presented hereinafter we have used the Adam optimizer with the learning rate of $1 \cdot 10^{-3}$. For the DQN we have also used a replay buffer with the maximum size of $128\,000$ samples, from which batches of size 32 are sampled at every training step. The discount factor was set to $\gamma = 0.99$. The target network was updated softly at the rate of $1 \cdot 10^{-2}$. We have used the ε-greedy policy with a linear schedule for ε, which starts from the value of 1 and then gradually decreases to 0.01 over 20000 steps.

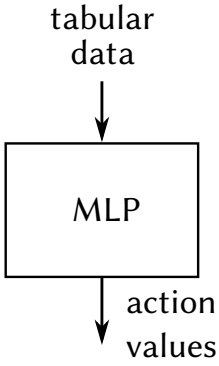
9.2.5 Results

To see how the various agents did and whether any of them was able to reduce the PA, we can now turn to the results of our experiments. The first figure that we are going to examine is Fig. 9.9. It shows the PA profiles of the various agents displayed as violin plots. The vertical axis shows all the various PA values and the width of the violin plot indicates the frequency of each particular value – in a similar way that a histogram would.

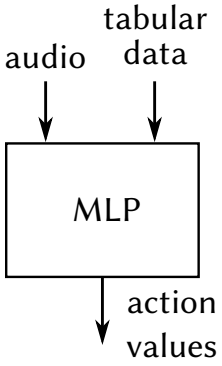
We can see that with the random agent, the most common PA value (indicated by the mode of the violin plot) is just below 30; PA values lower than 25 are very uncommon. In comparison to this, the 1D CNN and 2D CNN architectures are doing much better; it is clear that the mode has shifted down slightly, but – more importantly – also that a significant



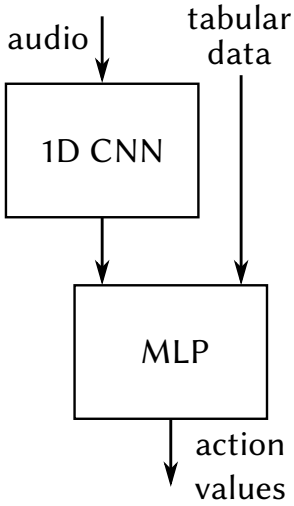
(a) A random agent: the actions are sampled from a uniform distribution.



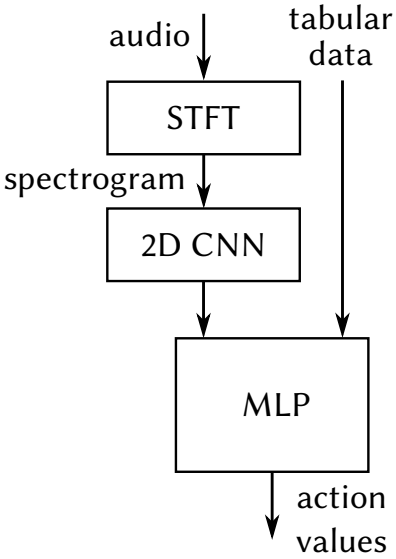
(b) A multi-layer perceptron that only takes in the tabular data.



(c) A multi-layer perceptron that takes in both: the audio signal and the tabular data.

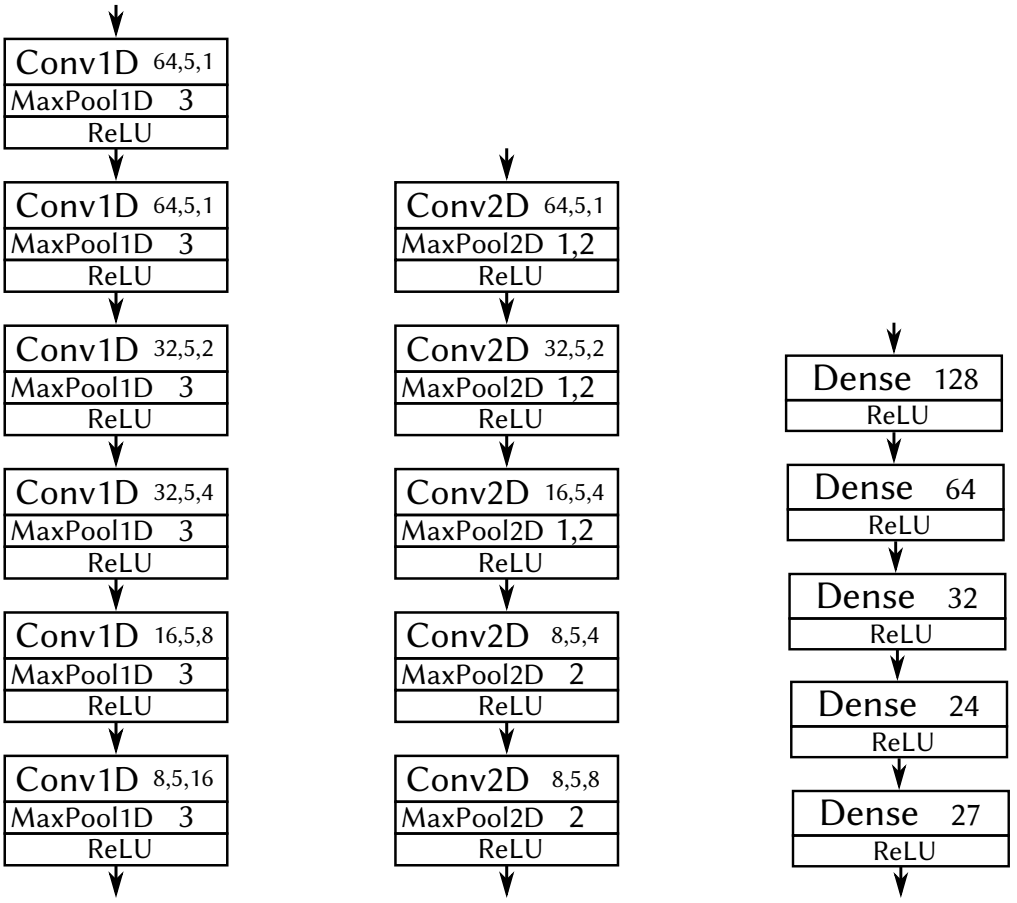


(d) Audio is preprocessed by a 1-D convolutional neural network.



(e) Audio's spectrogram is preprocessed by a 2-D convolutional neural network.

Fig. 9.7: The various architectural configurations that we experimented with.



(a) The 1D CNN architecture. (b) The 2D CNN architecture. (c) The MLP architecture.

Fig. 9.8: Components of our architectural configurations. Parameter notation – Conv(1D/2D): (number of filters, kernel dimensions, dilation rate); MaxPool(1D/2D): (pooling size); Dense: (number of units).

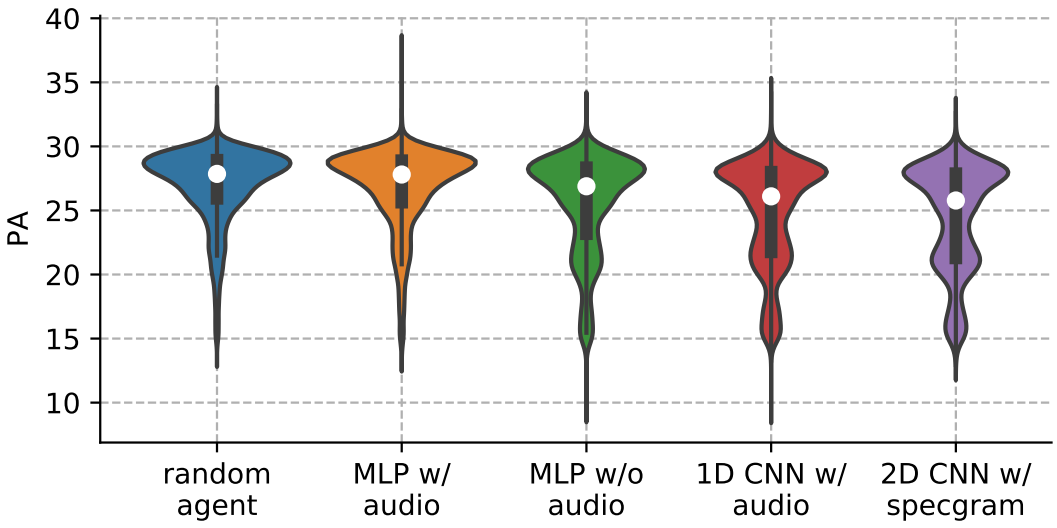


Fig. 9.9: Violin plots of the PA profiles taken across all training steps.

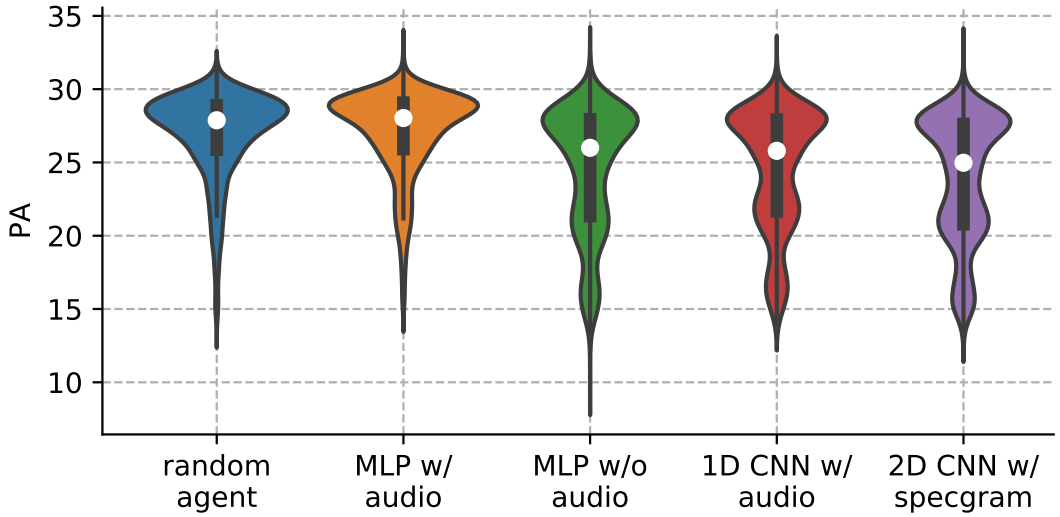


Fig. 9.10: Violin plots of the PA profiles taken after the first 11 000 steps of training.

portion of the frequency has been redistributed to two new modes: a mid-PA mode at around 20–22 and a low-PA mode at around 16–17.

The same trend – only more pronounced – is apparent in Fig. 9.10, which also shows PA profiles, but only considers data collected after the first 11 000 steps of training. The mid- and low-PA modes are larger and the high-PA mode has lost even more mass.

To get some idea of how the learning proceeds we can examine Fig. 9.11, which shows the cumulative reward curves for all the different agents: i.e. we visualize how much total reward the agent has accumulated until each time step. This plot indicates that the 1D and 2D CNN agents learn faster than the other agents and achieve better performance overall.

Most importantly though, the results indicate that it is indeed feasible to take actions that lower the psychoacoustic annoyance – which is what we wanted to know.

9.2.6 Multisensory Architectures

Let us now also discuss the performance differences across our several multisensory architectures. As we have mentioned in section 9.1, we cannot expect good results if we combine multiple sensor streams by simply concatenating them and feeding them into the input layer of an MLP. If we mix tabular data with audio/video data in this way, not only will the added audio data/video data not be likely to increase performance. The performance can actually be expected to degrade, because the MLP will have a hard time telling the useful correlations with tabular data apart from any spurious correlation from the very high-dimensional audio/video data.

This idea is, indeed, supported by our empirical results. Perhaps the clearest illustration is given in Fig. 9.11, which shows the cumulative rewards. We can see that the architecture, which naïvely concatenates the audio with the tabular inputs and feeds that into the MLP

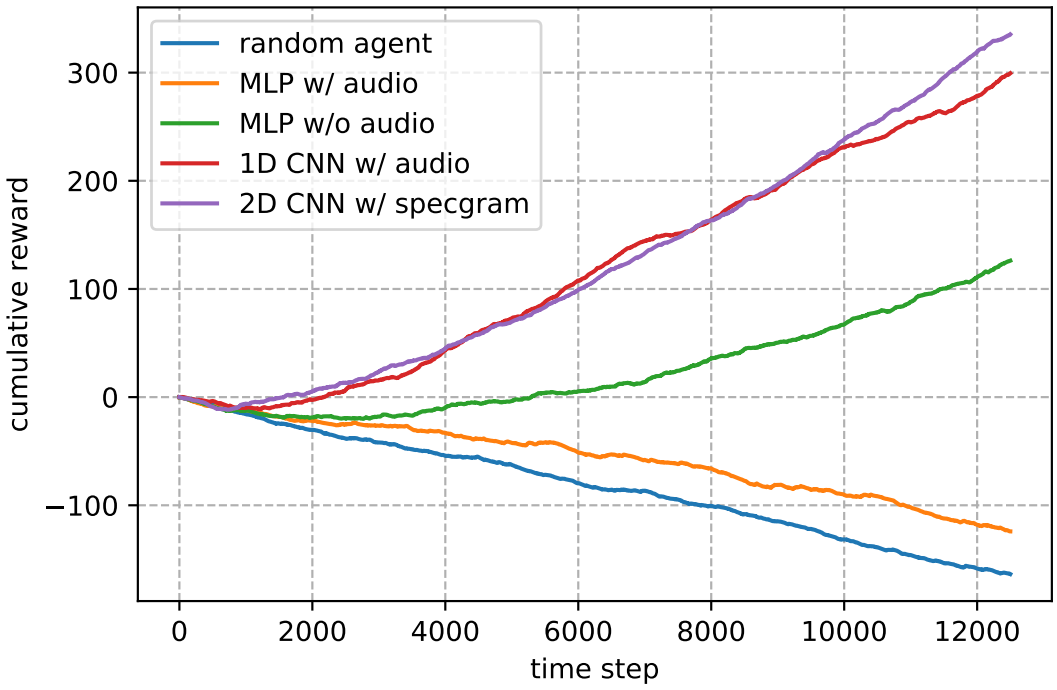


Fig. 9.11: The cumulative rewards for the different agents.

(denoted MLP w/ audio) performs much worse than the other agents – in fact it is only marginally better than the random agent.

In contrast to this, the MLP that only uses the tabular data without any audio (denoted MLP w/o audio) is still able to reduce the PA. This not only shows that the naïve approach to combining multisensory inputs actually makes things worse, but it also demonstrates that there are strategies that can reduce the PA in general – even without considering the current soundscape. This last finding will be further supported by the real data that we have collected.

Naturally, agents that are able to properly utilize the audio input have the best performance. This further illustrates that deep reinforcement learning can effectively combine data from different sensory streams, always provided that the architecture is properly designed – so that it biases the deep network towards preprocessing the inputs and only combining them once their dimensionality has been sufficiently reduced.

9.3 | Real Data

To complement our simulation experiments we have also collected real data, using an instrumented Lincoln MKZ. We have been recording the following data:

- 3D point cloud map (LiDAR);
- pressure data;
- IMU data;
- acceleration pedal;
- brake pedal/torque;
- gear;

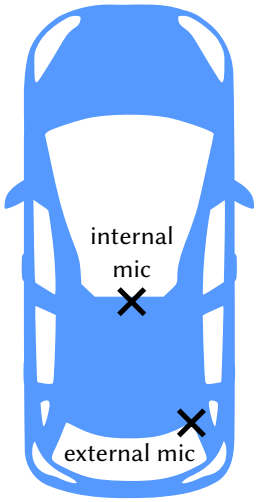


Fig. 9.12: Microphone positions.

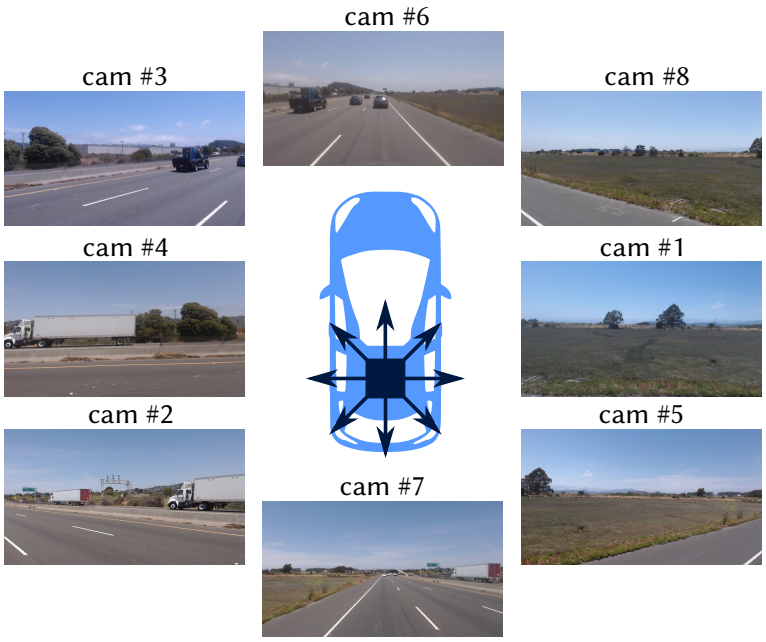


Fig. 9.13: The 8 cameras for video collection.

- wheel joint states (position & velocity);
- steering wheel angle/torque;
- suspension;
- tire pressures;
- wheel speeds;
- turn signals;
- internal/external audio;
- external video.

The audio was recorded using 2 Olympus ME-51S microphones with wind muffs. The positions of the two microphones are indicated in Fig. 9.12. The external video comes from 8 directional cameras – the layout is illustrated in Fig. 9.13.

For a more complete version of our analysis of this dataset, the reader can refer to [102]. Here we will only focus on two figures, which illustrate the point made in the feasibility study: that it is possible to reduce the PA to a certain extent even without perceiving the current soundscape.

9.3.1 Speed, Gears and Braking

Fig. 9.14 shows a scatter plot of speed vs. the PA. It displays the original data points as well as the moving average over 200 samples (for added readability). It is clear that the PA is high at low speeds – when the car is accelerating. It can get even higher at especially high speeds. However, around $20\,\mathrm{mph}$ there is a minimum. It is general relationships such as this one, that the agent without audio input can use to achieve lower PA.

A further instance of the same principle can be seen in Fig. 9.15. The plots illustrate that certain gears tend to be associated with lower PA values than others, and that braking tends to correlate with higher PA.

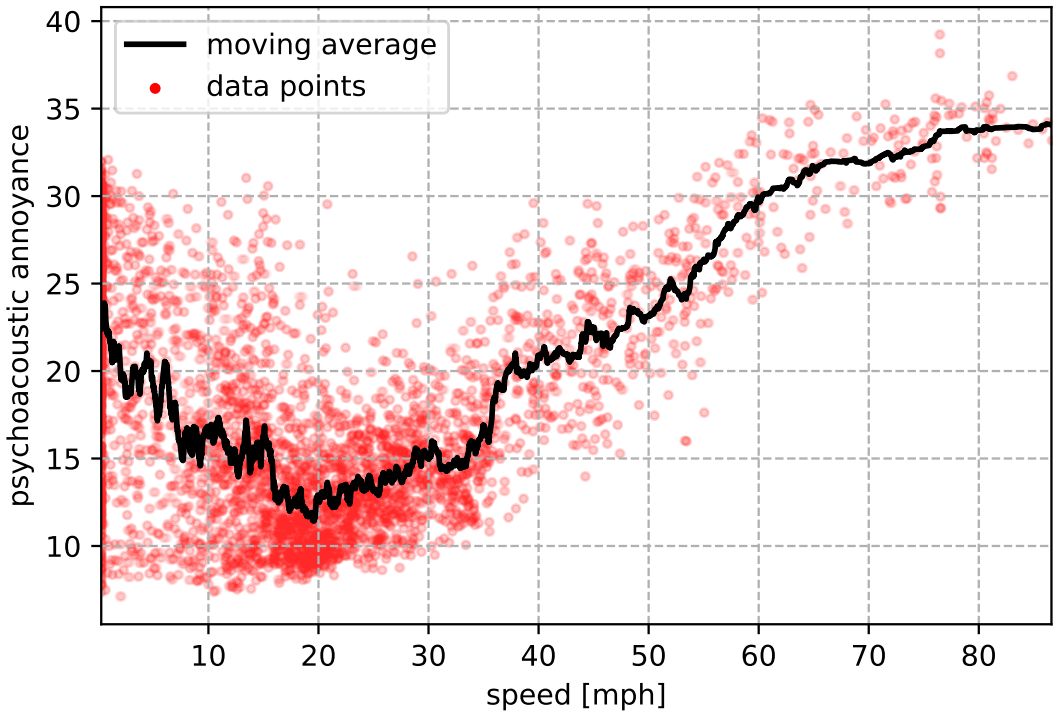


Fig. 9.14: Speed vs. PA: points correspond to measurements; the curve shows the moving average over 200 samples.

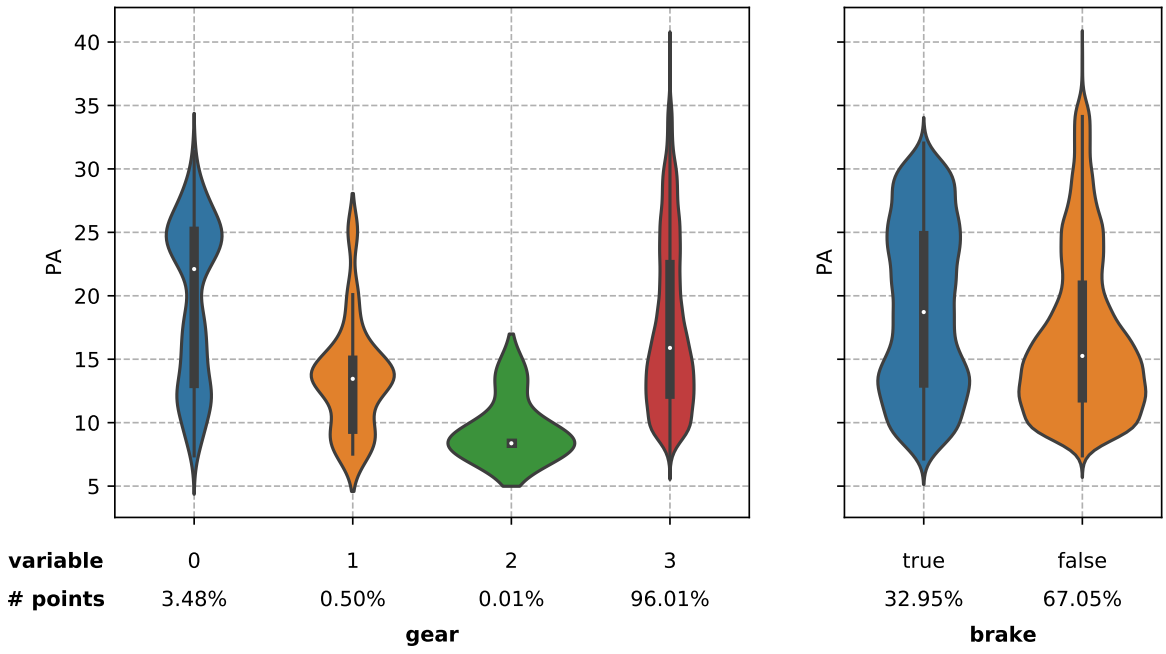


Fig. 9.15: Psychoacoustic annoyance under various gears (left) and with braking or not braking (right).

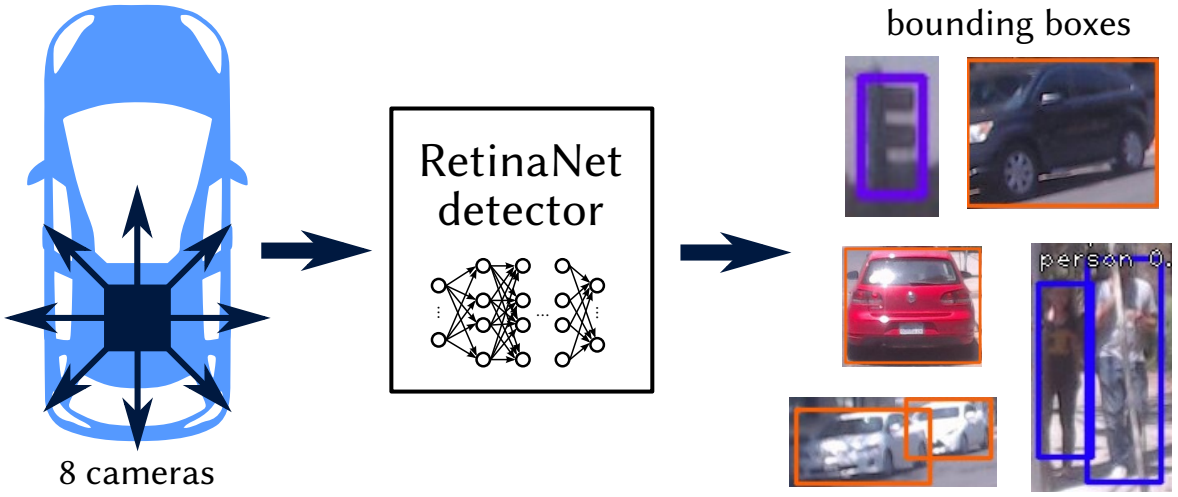


Fig. 9.16: Counting nearby objects using a pre-trained RetinaNet detector.

9.3.2 Number of Nearby Vehicles

One further thing that we wanted to verify was whether there might be a relationship between the number of other vehicles surrounding the car and the PA. The intuition behind this idea is, of course, that cars make noise.

In order to test this hypothesis, we have used images from the 8 directional cameras and passed them through a pre-trained RetinaNet visual object detector (from [107]), which can output the bounding boxes for several different kinds of objects, including different types of cars and motorbikes, but also bicycles, pedestrians, traffic lights and such, as illustrated in Fig. 9.16.

A sample image with the bounding boxes predicted by RetinaNet is shown in Fig. 9.17. The robustness of the detector is unlikely to be perfect, given that it was trained using different data, but it should be able to provide a reasonable estimate.

The actual relationship between the number of nearby vehicles and the PA that we have observed in our dataset is displayed in Fig. 9.18. Although the PA is not distributed uniformly, there is also no clear trend.

The trend could be missing because the number of nearby cars really is not a good indicator of the PA. However, it could also be because (as illustrated in Fig. 9.19) our RetinaNet detector picks out both: moving cars and parked cars, but only moving cars contribute to the noise. Since it is not immediately obvious how we could easily differentiate between moving and parked cars, we did not pursue this idea further.

Since the instrumented vehicle was equipped with a GPS unit, we were also able to plot measurements against their GPS coordinates. An example of this is shown in Fig. 9.20. However, visualizations of this kind are perhaps more useful for exploratory analyses and sanity checks than for extracting generalizable knowledge – at least unless the map regions

Fig. 9.17: A sample image with bounding boxes by RetinaNet.

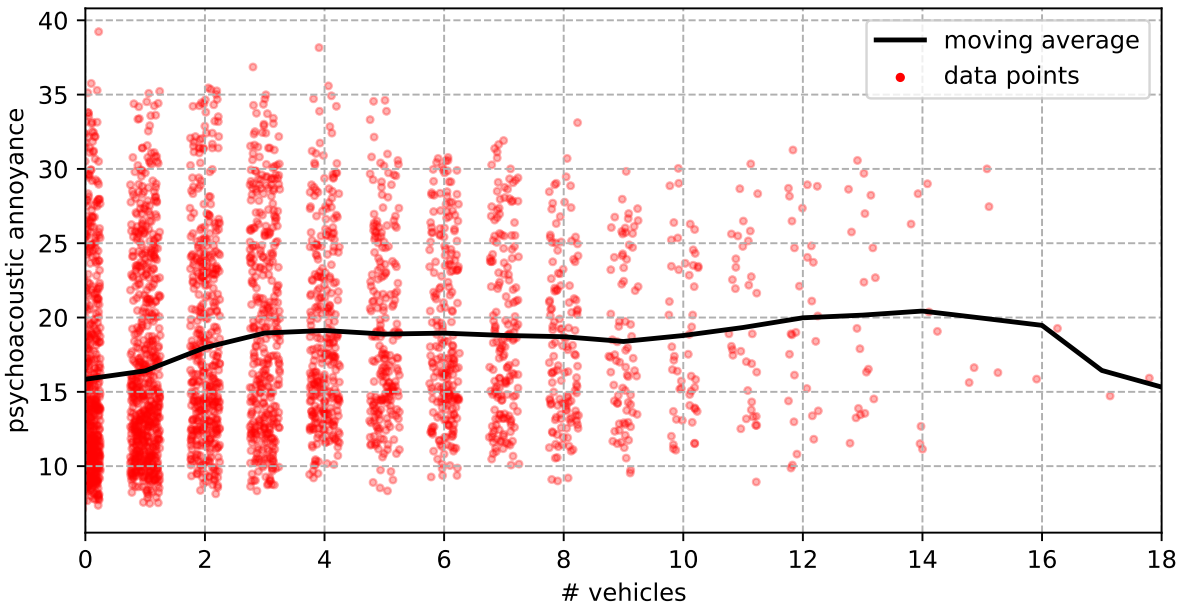


Fig. 9.18: Number of nearby cars vs the PA.

Fig. 9.19: RetinaNet detects both moving and parked vehicles.

onto which the measurements are overlaid are annotated with further information that would make the observed regularities transferable to other similar locations.

9.4 | Discussion and Future Work

The results of our feasibility study have shown several things:

- It is indeed feasible to reduce the PA by taking actions such as modifying the speed of the car, controlling the windows, the air conditioning and such.
- The PA can be reduced to some extent even without having information about the current soundscape.
- The data collected from a real car shows that there is a relationship between the PA on one hand and the speed of the car, braking, and the active gear on the other hand.

More relevantly to our inquiry into multisensory RL, the results clearly indicate that

- It is possible to effectively combine several multisensory streams using deep RL.
- For this to work, it is necessary to provide the appropriate cognitive bias through sensible architecture design. A good strategy is to have the network preprocess all the inputs and only combine them once their dimensionality has been reduced.
- If the different sensory streams are not preprocessed properly and we just naïvely concatenate them, this can have a significant negative impact on the performance.

  We have shown that in our case adding the high-dimensional audio stream in this naïve way strongly degrades the performance in comparison to the agent, which does not use the audio stream at all.

The research – considered as work in the psychoacoustic domain – could be further ex-

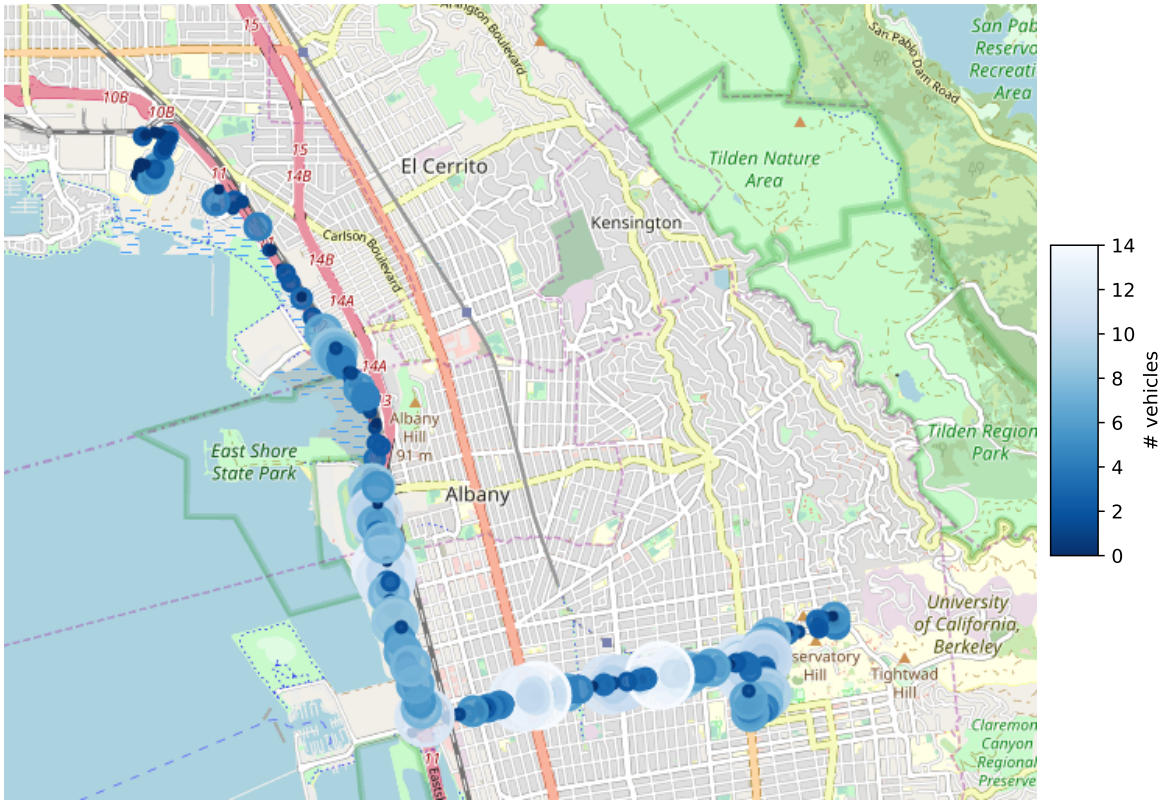


Fig. 9.20: The number of vehicles overlaid on a map by the GPS coordinates of the measurements.

tended in several directions. Most notably, perhaps, some research should be invested into designing a proper human-machine interface for the system – otherwise its actions could introduce more distractions than they prevent. Several other suggestions are discussed in [102], but we are not going to address them again here.

From the point of view of multisensory RL the most interesting possibility for further research would probably be to add a further sensory stream: visual data. Such data is already available through the DeepGTAV interface – but we are currently not using it. It would be interesting to see whether it might be possible to extract some information from it that the audio and the tabular inputs do not already carry.

That the visual input could, in principle, carry cues about incoming acoustic events seems obvious – as an instance of this, one can think of a sign warning that there is a construction site down the road. However, it remains unclear how prevalent such events are and to what extent the agent would be able to take advantage of them.

Furthermore, one could even help the agent by designing some of its feature space in a semi-automatic way. For an instance, we could use a pre-trained visual detector to pick out any vehicles surrounding the agent – like we did in the section about real data. Even though the data itself showed no clear correlation with the PA, it is possible that to the RL agent the information would be useful.

CHAPTER 10

CONCLUSION

As we have seen in the thesis, designing intelligent agents is a very complex and multi-disciplinary problem. However, the deep reinforcement learning framework seems – thanks to the power of the deep learning models that it is based on – to be flexible enough to bring a lot of the necessary components together in an unprecedented way.

The applications of deep neural networks are very wide and their ability to automatically learn good features has brought about huge strides in machine learning. However, as we have seen, even these powerful models only work well when their architecture design provides them with useful cognitive bias, which reflects prior knowledge about how the task is structured and how it should be approached. It is this that helps them to generalize well.

The fact is evident in all areas, where deep learning has achieved success. In computer vision applications, convolutional architectures incorporate the knowledge that image is built up from local patterns, which hierarchically form more and more complex features. When processing sequential data, where memory is required, special architectures such as the long short-term memory (LSTM) or the gated recurrent unit (GRU) are required to achieve good results. Recently though, some of these architectures are being replaced with sequential attention mechanisms.

We have shown, that the same is true in reinforcement learning, and that the concept of visual attention can help the agent to apply certain forms of abstraction more aggressively, which again helps the agent to generalize. Furthermore, it has been shown that the approach can also help to increase sample efficiency to some extent – and high sample complexity is one of the main barriers to wider application of deep reinforcement learning systems at the present. Our visual attention operator will also likely help to transfer a trained agent to other variants of the same task more easily – however, this is, in fact, yet to be verified.

To make the approach more easily applicable to other problems, which may require more sophisticated forms of visual summarization than the one we have used in our initial experiments, we have tried to design an operator for automatic learned visual summarization. The approach, in its current form, has proven not to be effective. However, we have outlined several possible problems and hope to be able to resolve the issues in our future work.

A further point of interest is that in realistic applications, intelligent agents need to be able to make use of rich, multisensory inputs. Being able to learn effective ways to fuse data of multiple types and from multiple sources automatically, is a vital necessity. We have presented a feasibility study, which illustrates that this can be done using deep reinforcement learning, but that a proper architecture design is again of paramount importance.

The feasibility study reports on experiments, where tabular data and acoustic signal are used together as inputs to an agent, which is supposed to reduce psychoacoustic annoyance to a human driver in a vehicle. The study is carried out in a rich simulation environment, where psychoacoustic annoyance is evaluated using an existing metric from the literature. As we have shown, it is indeed possible – at least in principle – to take actions, which reduce the metric. The results are also in part already supported by real data that we have collected.

We have used this experiment to show that improperly designed architectures are not only unable to benefit from the information contained in both the data streams – introducing the acoustic data can actually harm performance when both signals are just naïvely concatenated and fed into the input layer of the network.

All in all, then, the present work shows that architecture design is a powerful way to incorporate prior knowledge. It also reinforces the idea that deep reinforcement learning is a promising and flexible framework for intelligent agent design.

Naturally, a lot of open problems still remain. Many vital features such as automatic concept discovery, effective unsupervised learning, or language grounding and the consequent ability to provide rich natural-language instructions and feedback to the agent, continue to elude us. Nevertheless, there are good grounds for cautious optimism. We still do not have all the answers, but in the future we may yet be able to find the holy grail of artificial intelligence: the ability to design intelligent agents.

BIBLIOGRAPHY

[1] Rojas, R. *Neural Networks: A Systematic Introduction.* Berlin: Springer Verlag, 1996. ISBN 9783540605058. URL: <http://page.mi.fu-berlin.de/rojas/neural/index.html.html> (cit. 2014-05-30).

[2] Chapelle, O. – Schölkopf, B. – Zien, A. *Semi-Supervised Learning.* Cambridge, Massachusetts: The MIT Press, 2006. ISBN 978-0-262-03358-9.

[3] Kröse, B. – Smagt, P. v. d. *An Introduction to Neural Networks.* Amsterdam, The Netherlands: University of Amsterdam, 1996.

[4] McInnes, L. – Healy, J. *Umap: Uniform manifold approximation and projection for dimension reduction.* arXiv preprint arXiv:1802.03426, 2018.

[5] Frommberger, L. *Qualitative Spatial Abstraction in Reinforcement Learning.* Berlin, Heidelberg: Springer Verlag, 2010. ISBN 978-3-642-16589-4. URL: <http://www.springerlink.com/index/10.1007/978-3-642-16590-0> (cit. 2014-05-30).

[6] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, Massachusetts: MIT Press, 1998. ISBN 0-262-11170-5.

[7] Gregor, M. *Applications of Artificial Intelligence Methods to Design and Control of Robotic Systems.* Master's thesis, Žilinská univerzita v Žiline, Elektrotechnická fakulta, Katedra riadiacich a informačných systémov, 2011. Supervised by Juraj Spalek.

[8] Spalek, J. – Gregor, M. *Adaptive Approaches to Parameter Control in Genetic Algorithms and Genetic Programming.* Applied Computer Science: Improvements Methods in Manufacturing Design, Scheduling and Control, 7(1):38–56, 2011. ISSN 1895-3735. URL: <http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-f8a07e0a-7da5-43dd-afc3-5e0455daf098>.

[9] Gregor, M. – Spalek, J. *On Use of Node-attached Modules with Ancestry Tracking in Genetic Programming.* In 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems. 2012. ISBN 978-3-902823-21-2. URL: <https://doi.org/10.3182/20120523-3-CZ-3015.00027>.

[10] Gregor, M. – Spalek, J. *Using Context Blocks in Genetic Programming with JIT Compilation.* ATP Journal PLUS, (2), 2013. ISSN 1336-5010.

[11] Gregor, M. – Spalek, J. *Using LLVM-based JIT compilation in genetic programming.* In 2016 ELEKTRO, pp. 406–411. IEEE, 2016. ISBN 978-1-4673-8698-2. URL: <https://doi.org/10.1109/ELEKTRO.2016.7512108>.

[12] Salimans, T. – Ho, J. – Chen, X. et al. *Evolution strategies as a scalable alternative to reinforcement learning.* arXiv preprint arXiv:1703.03864, 2017.

[13] Such, F. P. – Madhavan, V. – Conti, E. et al. *Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning.* arXiv preprint arXiv:1712.06567, 2017.

[14] Gregor, M. – Spalek, J. *Novelty Detector for Reinforcement Learning Based on Forecasting.* In IEEE 12th International Symposium on Applied Machine Intelligence and Informatics: Proceedings, pp. 73–78. Herľany, Slovak Republic, 2014. ISBN 978-1-4799-3441-6. URL: <https://doi.org/10.1109/SAMI.2014.6822379>.

[15] Gregor, M. – Spalek, J. *Curiosity-driven Exploration in Reinforcement Learning.* In Proceedings of 10th International Conference, ELEKTRO 2014, pp. 435–439. Rajecké Teplice, Slovak Republic, 2014. ISBN 978-1-4799-3720-2. URL: <https://doi.org/10.1109/ELEKTRO.2014.6848933>.

[16] Gregor, M. – Spalek, J. *The Optimistic Exploration Value Function.* In INES 2015: IEEE 19 th International Conference on Intelligent Engineering Systems, Proceedings, 19, pp. 119–123. Bratislava: IEEE, 2015. ISBN 978-1-4673-7938-0. URL: <https://doi.org/10.1109/INES.2015.7329650>.

[17] Gregor, M. *Control System of an Autonomous Robot for Solving Multi-objective Tasks.* PhD thesis, University of Žilina, 2014.

[18] Pathak, D. – Agrawal, P. – Efros, A. A. – Darrell, T. *Curiosity-driven Exploration by Self-supervised Prediction.* In ICML. 2017.

[19] Burda, Y. – Edwards, H. – Pathak, D. et al. *Large-Scale Study of Curiosity-Driven Learning.* In ICLR. 2019.

[20] Gregor, M. – Nemec, D. – Janota, A. – Pirník, R. *A Visual Attention Operator for Playing Pac-Man.* In Proceedings of 12th International Conference, ELEKTRO 2018. 2018. ISBN 978-1-5386-4759-2. URL: <https://doi.org/10.1109/ELEKTRO.2018.8398308>.

[21] Hanes, J. *Učenie s odmenou* [Reinforcement Learning]. Master's thesis, Žilinská univerzita, 2017. Supervised by Michal Gregor.

[22] Cardon, D. – Cointet, J.-P. – Mazières, A. *The Revenge of Neurons.* 2018. URL: <https://neurovenge.antonomase.fr/> (cit. 2018-11-18).

[23] Groumpos, P. P. – Stylios, C. D. *Modelling supervisory control systems using fuzzy cognitive maps.* Chaos, Solitons & Fractals, 11(1):329–336, 2000.

[24] Groumpos, P. P. *Fuzzy Cognitive Maps: Basic Theories and Their Application to Complex Systems.* In M. Glykas, editor, Fuzzy Cognitive Maps, p. 1–22. Berlin: Springer-Verlag, 2010.

[25] Dickerson, J. A. – Kosko, B. *Virtual worlds as fuzzy cognitive maps.* In Virtual Reality Annual International Symposium, pp. 471–477. IEEE, 1993.

[26] Gregor, M. – Groumpos, P. P. *Tuning the Position of a Fuzzy Cognitive Map Attractor using Backpropagation through Time.* In Proceedings of The 7th International Conference on Integrated Modeling and Analysis in Applied Control and Automation (IMAACA 2013), pp. 78–86. Athens, 2013. ISBN 978-1-62993-488-4.

[27] Gregor, M. – Groumpos, P. P. *Training Fuzzy Cognitive Maps using Gradient-based Supervised Learning.* In Artificial Intelligence Applications and Innovations (9th IFIP WG 12.5 International Conference, AIAI 2013, Paphos, Cyprus, September 30 – October 2, 2013, Proceedings), IFIP Advances in Information and Communication Technology, pp. 547–556. Cyprus: Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41142-7. URL: <http://dx.doi.org/10.1007/978-3-642-41142-7_55>.

[28] Gregor, M. – Groumpos, P. P. – Gregor, M. *Using Weight Constraints and Masking to Improve Fuzzy Cognitive Map Models.* In Conference on Creativity in Intelligent Technologies and Data Science, pp. 91–106. Springer, 2017. ISBN 978-3-319-65551-2. ISSN 1865-0937. URL: <http://dx.doi.org/10.1007/978-3-319-65551-2_7>.

[29] Gregor, M. – Miklošík, I. – Spalek, J. *Automatic tuning of a fuzzy meta-model for evacuation speed estimation.* In 2016 Cybernetics & Informatics (K&I), pp. 1–6. IEEE, 2016. ISBN 978-1-5090-1834-5. URL: <https://doi.org/10.1109/CYBERI.2016.7438594>.

[30] Kello, P. – Gregor, M. – Miklošík, I. – Spalek, J. *Learning a Fuzzy Model for Evacuation Speed Estimation using Fuzzy Decision Trees and Evolutionary Methods.* Zeszyty Naukowe Wyższej Szkoły Technicznej w Katowicach, (9):49–62, 2017. ISSN 2450-5552. URL: <http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-6fdc439d-1a22-4321-aaef-46237387713d>.

[31] Kindernay, J. *Prestavba grafického užívateľského rozhrania pre knižnicu Fuzzylite* [Rebuilding of the graphical user interface for the Fuzzylite library]. Master's thesis, Žilinská univerzita, 2017. Supervised by Michal Gregor.

[32] Gregor, M. – Zábovská, K. – Smataník, V. *The Zebra Puzzle and Getting to Know Your Tools.* In INES 2015: IEEE 19th International Conference on Intelligent Engineering Systems, Proceedings, 19, pp. 159–164. Bratislava: IEEE, 2015. ISBN 978-1-4673-7938-0. URL: <https://doi.org/10.1109/INES.2015.7329698>.

[33] Bučko, B. – Hanušniak, V. – Joštiak, M. – Zábovská, K. *Inferring over functionally equivalent ontologies.* In Digital Information Processing and Communications (ICDIPC), 2015 Fifth International Conference on, pp. 201–206. IEEE, 2015. URL: <https://doi.org/10.1109/ICDIPC.2015.7323029>.

[34] Sun, R. – Zhang, X. *Top-down versus Bottom-up Learning in Cognitive Skill Acquisition.* Cognitive Systems Research, 5(1):63–89, 2004.

[35] Gregor, M. – Spalek, J. *Log-learning: A Preliminary Study.* In 2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), pp. 235–238. IEEE, 2015. ISBN 978-1-4673-9282-2. URL: <https://doi.org/10.1109/ICUMT.2015.7382434>.

[36] Andre, D. – Russell, S. J. *State abstraction for programmable reinforcement learning agents.* In AAAI/IAAI, pp. 119–125. 2002.

[37] Marthi, B. – Russell, S. – Latham, D. *Writing Stratagus-playing agents in concurrent ALisp.* Reasoning, Representation, and Learning in Computer Games, pp. 67–71, 2005.

[38] Marthi, B. – Russell, S. J. – Latham, D. – Guestrin, C. *Concurrent hierarchical reinforcement learning.* In IJCAI, pp. 779–785. 2005.

[39] Lewis, M. – Yarats, D. – Dauphin, Y. N. et al. *Deal or no deal? end-to-end learning for negotiation dialogues.* arXiv preprint arXiv:1706.05125, 2017.

[40] Co-Reyes, J. D. – Gupta, A. – Sanjeev, S. et al. *Guiding Policies with Language via Meta-Learning.* arXiv preprint arXiv:1811.07882, 2018.

[41] Gregor, M. – Gregor, M. *Chatbot systémy v inteligentných rečových užívateľských rozhraniach* [Chatbot Systems in Intelligent Voice User Interfaces]. ProIN, 16(4):50–55, 2015. ISSN 1339-2271.

[42] Jombík, M. *Chatbot systémy* [Chatbot Systems]. Master's thesis, Žilinská univerzita, 2014. Supervised by Michal Gregor.

[43] Slováček, L. *Chatbot systémy* [Chatbot Systems]. Bachelor's thesis, Žilinská univerzita, 2016. Supervised by Michal Gregor.

[44] Lokaj, M. *Kontrola pravopisu na báze rekurentných neurónových sietí* [Grammar Checking based on Recurrent Neural Networks]. Master's thesis, Žilinská univerzita, 2018. Supervised by Michal Gregor.

[45] Gregor, M. *Systém rozpoznávania reči* [A Speech Recognition System]. Bachelor's thesis, Žilinská univerzita, 2009. Supervised by Tomáš Michulek.

[46] Michulek, T. – Gregor, M. *Remote Control of an Autonomous Mobile 3DLS System using ANN-based Automatic Speech Recognition.* In Metody i techniki zarządzania w inżynierii produkcji. Bielsko-Biała: Wydawnictwo akademii techniczno-humanistycznej w Bielsku-Białej, 2009. ISBN 978-83-60714-64-5.

[47] Gregor, M. – Michulek, T. *Intelligent Manufacturing Systems – Automatic Speech Recognition System.* In Applied Computer Science, vol. 5. Bielsko-Biała: Wydawnictwo akademii techniczno-humanistycznej w Bielsku-Białej, 2009. ISBN 978-83-60714-95-9. URL: <http://www.acs.pollub.pl/index.php?option=com_content&view=article&id=143:intelligent-manufacturing-systems--automatic-speech-recognition-system&catid=47:vol-5-no-12009&Itemid=102>.

[48] Gregor, M. – Gregor, M. *Komponenty inteligentných rečových užívateľských rozhraní* [Components of Intelligent Voice User Interfaces]. ProIN, 16(3):48–52, 2015. ISSN 1339-2271.

[49] Gregor, M. – Gregor, M. *Aplikácie inteligentných rečových užívateľských rozhraní* [Applications of Intelligent Voice User Interfaces]. ProIN, 16(5-6):51–56, 2015. ISSN 1339-2271.

[50] Vít, M. *Implementácia umelých hráčov pre hru 2048* [Implementation of Artificial Players for 2048]. Bachelor's thesis, Žilinská univerzita, 2018. Supervised by Michal Gregor.

[51] Gregor, M. – Kovalský, M. – Gregor, T. *Rozvrhovanie I: Programovanie ohraničení* [Scheduling I: Constraint Programming]. ProIN, 18(1):26–30, 2017. ISSN 1339-2271.

[52] Gregor, M. – Kovalský, M. – Gregor, T. *Rozvrhovanie II: Nástroj MiniZinc* [Scheduling II: MiniZinc]. ProIN, 18(2):55–61, 2017. ISSN 1339-2271.

[53] Bujňák, D. *Implementácia a riešenie plánovacieho problému* [Implementation and Solution of a Planning Problem]. Master's thesis, Žilinská univerzita, 2016. Supervised by Michal Gregor.

[54] Nemec, D. *Riadenie komplexných robotických mobilných systémov* [Control of Complex Robotic Mobile Systems]. Ph.D. thesis, Žilinská univerzita, 2018. Supervised by Aleš Janota.

[55] Gregor, M. *Umelá inteligencia 1* [Artificial Intelligence 1]. Žilina: CEIT, a.s., 2014. ISBN 978-80-971684-1-4.

[56] Gregor, M. – Nemec, D. – Hruboš, M. – Spalek, J. *Umelá inteligencia 2: Hlboké učenie* [Artificial Intelligence 2]. CEIT, a.s., 2017. ISBN 978-80-89865-03-1.

[57] Gregor, M. – Hruboš, M. – Nemec, D. *Umelá inteligencia, skriptá I: Návody na vybrané cvičenia* [Artificial Intelligence, Lecture Notes I]. CEIT, a.s., 2017. ISBN 978-80-89865-02-4.

[58] Gregor, M. – Janota, A. – Hruboš, M. *Kompendium vybraných metód umelej inteligencie.* EDIS: Vydavateľstvo Žilinskej univerzity, 2018. ISBN 978-80-554-1539-0.

[59] Sutton, R. – Barto, A. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning. The MIT Press, 1998. ISBN 0262193981.

[60] Sutton, R. S. – McAllester, D. A. – Singh, S. P. – Mansour, Y. *Policy gradient methods for reinforcement learning with function approximation.* In NIPS'99 Proceedings of the 12th International Conference on Neural Information Processing Systems, pp. 1057–1063. 1999.

[61] Silver, D. – Lever, G. – Heess, N. et al. *Deterministic policy gradient algorithms.* In ICML. 2014.

[62] Sutton, R. S. – Barto, A. G. *Reinforcement Learning: An Introduction.* Cambridge, Massachusetts: The MIT Press, 2018, second edition. ISBN 9780262039246.

[63] Silver, D. *Lecture 4: Model-Free Prediction.* University College London. 2015. URL: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf> (cit. 2019-05-11).

[64] Kroese, D. P. – Taimre, T. – Botev, Z. I. *Handbook of monte carlo methods*, vol. 706. John Wiley & Sons, 2013.

[65] Cichosz, P. *An analysis of experience replay in temporal difference learning.* Cybernetics & Systems, 30(5):341–363, 1999.

[66] Cichosz, P. *TD (λ) learning without eligibility traces: a theoretical analysis.* Journal of Experimental & Theoretical Artificial Intelligence, 11(2):239–263, 1999.

[67] Busoniu, L. – Babuska, R. – De Schutter, B. – Ernst, D. *Reinforcement Learning and Dynamic Programming using Function Approximators.* CRC press, 2010. ISBN 9781439821084.

[68] Konidaris, G. – Osentoski, S. *Value Function Approximation in Reinforcement Learning using the Fourier Basis.* Computer Science Department Faculty Publication Series, (101), 2008. URL: <http://scholarworks.umass.edu/cs_faculty_pubs/101>.

[69] Engel, Y. – Mannor, S. – Meir, R. *Bayes meets Bellman: The Gaussian process approach to temporal difference learning.* In Proceedings of the 20th International Conference on Machine Learning (ICML-03), pp. 154–161. 2003.

[70] Chapman, D. – Kaelbling, L. P. *Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons.* In IJCAI, vol. 91, pp. 726–731. 1991.

[71] Pyeatt, L. D. – Howe, A. E. et al. *Decision tree function approximation in reinforcement learning.* In Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models, vol. 2, pp. 70–77. Cuba, 1998.

[72] Dietterich, T. G. – Wang, X. *Batch value function approximation via support vectors.* In Advances in neural information processing systems, pp. 1491–1498. 2002.

[73] Mania, H. – Guy, A. – Recht, B. *Simple random search provides a competitive approach to reinforcement learning.* arXiv preprint arXiv:1803.07055, 2018.

[74] Chou, P.-W. – Maturana, D. – Scherer, S. *Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution.* In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 834–843. JMLR. org, 2017.

[75] Levine, S. *Lecture 5: Policy Gradients.* UC Berkeley. 2018. URL: <http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-5.pdf> (cit. 2019-06-10).

[76] Silver, D. *Lecture 7: Policy Gradient.* University College London. 2015. URL: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf> (cit. 2019-05-11).

[77] Mnih, V. – Kavukcuoglu, K. – Silver, D. et al. *Playing atari with deep reinforcement learning.* arXiv preprint arXiv:1312.5602, 2013.

[78] Mnih, V. – Kavukcuoglu, K. – Silver, D. et al. *Human-level control through deep reinforcement learning.* Nature, 518(7540):529–533, 2015.

[79] Silver, D. – Huang, A. – Maddison, C. J. et al. *Mastering the game of Go with deep neural networks and tree search.* Nature, 529(7587):484–489, 2016.

[80] Silver, D. – Schrittwieser, J. – Simonyan, K. et al. *Mastering the game of go without human knowledge.* Nature, 550(7676):354, 2017.

[81] Vinyals, O. – Babuschkin, I. – Chung, J. et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II.* [online], 2019. URL: <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/> (cit. 2019-26-04).

[82] Boutilier, C. *Reinforcement Learning for Recommender Systems: Some Foundational and Practical Issues.* [online], 2018. URL: <https://youtu.be/hzcaVWkyOzk> (cit. 2019-04-04).

[83] Gauci, J. – Conti, E. – Virochsiri, K. *Horizon: The first open source reinforcement learning platform for large-scale products and services.* [online], 2018. URL: <https://code.fb.com/ml-applications/horizon/> (cit. 2019-04-04).

[84] *DeepMind AI Reduces Google Data Centre Cooling Bill by 40%.* DeepMind. [online]. URL: <https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/> (cit. 2017-05-19).

[85] Vanian, J. *Google Artificial Intelligence Whiz Describes Our Sci-Fi Future.* [online], 2016. URL: <http://fortune.com/2016/11/26/google-artificial-intelligence-jeff-dean/> (cit. 2017-05-19).

[86] Peng, X. B. – Kanazawa, A. – Malik, J. et al. *SFV: Reinforcement learning of physical skills from videos.* In SIGGRAPH Asia 2018 Technical Papers, p. 178. ACM, 2018.

[87] Andrychowicz, M. – Baker, B. – Chociej, M. et al. *Learning dexterous in-hand manipulation.* arXiv preprint arXiv:1808.00177, 2018.

[88] Zhu, H. – Gupta, A. – Rajeswaran, A. et al. *Dexterous manipulation with deep reinforcement learning: Efficient, general, and low-cost.* arXiv preprint arXiv:1810.06045, 2018.

[89] Haarnoja, T. – Zhou, A. – Hartikainen, K. et al. *Soft actor-critic algorithms and applications.* arXiv preprint arXiv:1812.05905, 2018.

[90] Van Hasselt, H. – Guez, A. – Silver, D. *Deep Reinforcement Learning with Double Q-Learning.* In AAAI, vol. 2, p. 5. Phoenix, AZ, 2016.

[91] Wang, Z. – Schaul, T. – Hessel, M. et al. *Dueling network architectures for deep reinforcement learning.* arXiv preprint arXiv:1511.06581, 2015.

[92] Mnih, V. – Heess, N. – Graves, A. et al. *Recurrent models of visual attention.* In Advances in Neural Information Processing Systems, pp. 2204–2212. 2014.

[93] *Project 3: Reinforcement Learning.* UC Berkeley. [online], 2018. URL: <http://ai.berkeley.edu/reinforcement.html> (cit. 2018-01-18).

[94] Mnih, V. – Badia, A. P. – Mirza, M. et al. *Asynchronous methods for deep reinforcement learning.* In International Conference on Machine Learning, pp. 1928–1937. 2016.

[95] DeNero, J. – Klein, D. *Teaching introductory artificial intelligence with pac-man.* In Proceedings of the Symposium on Educational Advances in Artificial Intelligence. 2010.

[96] Brockman, G. – Cheung, V. – Pettersson, L. et al. *OpenAI gym.* arXiv preprint arXiv:1606.01540, 2016.

[97] et al., M. P. *Deep Reinforcement Learning for Keras.* [online]. URL: <https://github.com/matthiasplappert/keras-rl> (cit. 2018-01-19).

[98] *Deep Reinforcement Learning in Pac-man.* [online]. URL: <https://github.com/tychovdo/PacmanDQN> (cit. 2018-01-19).

[99] Kingma, D. – Ba, J. *Adam: A method for stochastic optimization.* arXiv preprint arXiv:1412.6980, 2014.

[100] He, K. – Zhang, X. – Ren, S. – Sun, J. *Deep residual learning for image recognition.* arXiv preprint arXiv:1512.03385, 2015.

[101] Huang, G. – Liu, Z. – Van Der Maaten, L. – Weinberger, K. Q. *Densely connected convolutional networks.* In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 4700–4708. 2017.

[102] Nascimento, E. – Bajcsy, R. – Gregor, M. et al. *Acoustic-driven Interior Vehicle Adaptation based on Deep Reinforcement Learning to Improve Driver's Comfort.* 2019. Unpublished.

[103] Araujo, E. – Gregor, M. – Huang, I. et al. *On Modeling of Effects of Auditory Annoyance on Driving Style and Passenger Comfort.* In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (iROS). 2019. In press.

[104] Ruano, A. *DeepGTAV: A plugin for GTAV that transforms it into a vision-based self-driving car research environment.* URL: <https://github.com/aitorzip/DeepGTAV> (cit. 2018-11-23).

[105] Zwicker, E. – Fastl, H. *Psychoacoustics: Facts and models*, vol. 22. Springer Science & Business Media, 2013.

[106] Plappert, M. *keras-rl.* [https://github.com/keras-rl/keras-rl], 2016.

[107] *Keras RetinaNet.* URL: <https://github.com/fizyr/keras-retinanet> (cit. 2018-08-02).

[108] Goodfellow, I. – Bengio, Y. – Courville, A. *Deep Learning.* MIT Press, 2016. URL: <http://www.deeplearningbook.org>.

[109] Marček, D. – Marček, M. *Neurónové siete a ich aplikácie* [Neural Networks and Their Applications]. Žilina: Žilinská univerzita v Žiline/EDIS, 2006. ISBN 80-8070-497-X.

[110] Xu, B. – Wang, N. – Chen, T. – Li, M. *Empirical evaluation of rectified activations in convolutional network.* arXiv preprint arXiv:1505.00853, 2015.

[111] Clevert, D.-A. – Unterthiner, T. – Hochreiter, S. *Fast and accurate deep network learning by exponential linear units (elus).* arXiv preprint arXiv:1511.07289, 2015.

[112] Ramachandran, P. – Zoph, B. – Le, Q. V. *Searching for activation functions.* 2018.

[113] Bengio, Y. *Learning deep architectures for AI.* Foundations and trends® in Machine Learning, 2(1):1–127, 2009.

[114] Bengio, Y. *Practical recommendations for gradient-based training of deep architectures.* In Neural Networks: Tricks of the Trade, pp. 437–478. Springer, 2012.

[115] He, K. – Zhang, X. – Ren, S. – Sun, J. *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.* In Proceedings of the IEEE International Conference on Computer Vision, pp. 1026–1034. 2015.

[116] Ioffe, S. – Szegedy, C. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* arXiv preprint arXiv:1502.03167, 2015.

[117] Olah, C. – Mordvintsev, A. – Schubert, L. *Feature Visualization.* Distill, 2017. URL: <https://distill.pub/2017/feature-visualization>.

[118] Schaul, T. – Quan, J. – Antonoglou, I. – Silver, D. *Prioritized experience replay.* arXiv preprint arXiv:1511.05952, 2015.

[119] Schulman, J. – Levine, S. – Abbeel, P. et al. *Trust region policy optimization.* In International Conference on Machine Learning, pp. 1889–1897. 2015.

[120] Doersch, C. *Tutorial on Variational Autoencoders.* arXiv preprint arXiv:1606.05908, 2016.

[121] Wright, S. – Nocedal, J. *Numerical optimization.* Springer Science, 35:67–68, 1999.

[122] Martens, J. *Deep learning via Hessian-free optimization.* In Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 735–742. 2010.

[123] Martens, J. – Sutskever, I. *Learning recurrent neural networks with Hessian-free optimization.* In Proceedings of the 28th International Conference on Machine Learning (ICML-11), pp. 1033–1040. 2011.

[124] Martens, J. – Sutskever, I. *Training deep and recurrent networks with hessian-free optimization.* In Neural networks: Tricks of the trade, pp. 479–535. Springer, 2012.

[125] Pascanu, R. – Bengio, Y. *Revisiting natural gradient for deep networks.* arXiv preprint arXiv:1301.3584, 2013.

[126] Schulman, J. – Wolski, F. – Dhariwal, P. et al. *Proximal policy optimization algorithms.* arXiv preprint arXiv:1707.06347, 2017.

[127] Lillicrap, T. P. – Hunt, J. J. – Pritzel, A. et al. *Continuous control with deep reinforcement learning.* arXiv preprint arXiv:1509.02971, 2015.

NOTES

[1] Figure made available by the authors under the CC-BY licence.

[2] There has been a lot of controversy with respect to the question of what precisely constitutes agenthood. Just as with the definition of the term artificial intelligence, no general consensus has been reached by the community. Since both of these definitions have by now become a point of philosophical debate rather than a purely terminological problem, lying within the safe confines of science, we will not attempt to address the issue here. We will merely ask the reader to accept the term agent as a part of the common parlance of this scientific field.

UNIVERSITY OF ŽILINA

FACULTY OF ELECTRICAL ENGINEERING
AND INFORMATION TECHNOLOGY

APPENDICES

ING. MICHAL GREGOR, PHD.

Towards Intelligent Agents using Deep Reinforcement Learning

Filing Number: 28260220195002

Žilina, 2019

APPENDIX A

DEEP LEARNING

Virtually all state state-of-the-art approaches to reinforcement learning currently use deep neural networks as function approximators. Before we proceed to the next chapter, which discusses some of these methods, it is therefore necessary to give a concise introduction to the area of deep learning as well. In the interest of brevity, we will try to keep this chapter to a bare minimum. For more detailed information, the reader can always refer to sources such as [108], or even to our own [56], in case a source in Slovak is preferable.

A.1 | Artificial Neuron

An artificial neuron is a mathematical structure that serves as a very simplified model of a biological neuron. Its structure is illustrated in Fig. A.1. It is characterized by the following properties [109]:

- $\mathbf{x} = (x_1, x_2, ..., x_n)$ – a vector of inputs,
- $\mathbf{w} = (w_1, w_2, ..., w_n)$ – a vector of weights corresponding to the individual inputs,
- Θ – the threshold potential,
- $f(x)$ – the activation function of the neuron,
- $y = f(x)$ – the output of the neuron,

where $x_i, w_i \in \mathbb{R} \ \ \forall i = 1, 2, ..., n$.

Neuron's inner potential u is computed in the following way:

$$u = \sum_{i=1}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x}, \tag{A.1}$$

and the following holds for the output y:

$$y = f(u - \Theta) = f\left(\sum_{i=1}^{n} w_i x_i - \Theta\right). \tag{A.2}$$

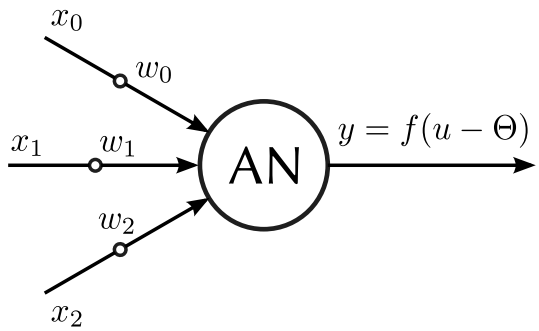


Fig. A.1: An artificial neuron.

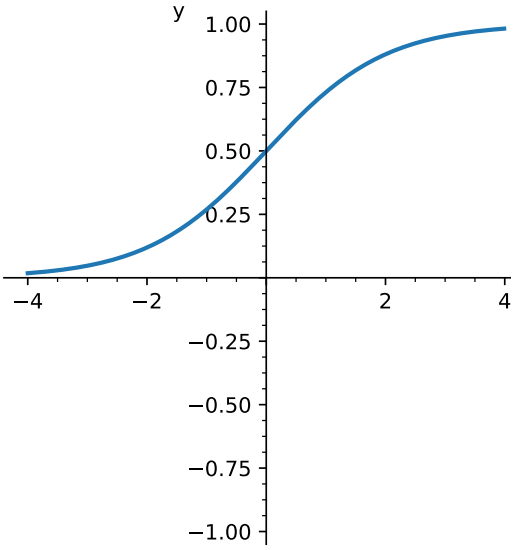


Fig. A.2: The sigmoid function.

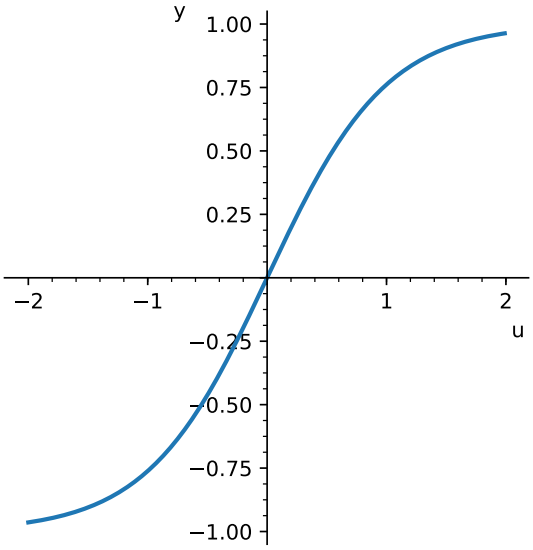


Fig. A.3: Activation function $\tanh(x)$.

A.1.1 The Activation Function

The activation function $f(x)$ can take one of many different forms. Historically, the sigmoid function (Fig. A.2) and the hyperbolic tangent (Fig. A.3) were by far the most popular choices. However, they have now been largely abandoned. The problem is that they both saturate very easily, which makes learning extremely slow, when using learning methods based on the gradient (i.e. all state-of-the-art methods).

Currently, the rectified linear unit (ReLU; see Fig. A.4) is a popular activation function choice. It is piecewise linear, which makes it very efficient computationally. More importantly, large inner potentials do not make it saturate.

Several versions of it have been proposed and used, including a leaky and a parametric version [110]. There is also a number of different alternatives, such as the exponential linear unit (ELU) [111], or the swish function [112] (which was actually discovered by an automatic search mechanism based on reinforcement learning).

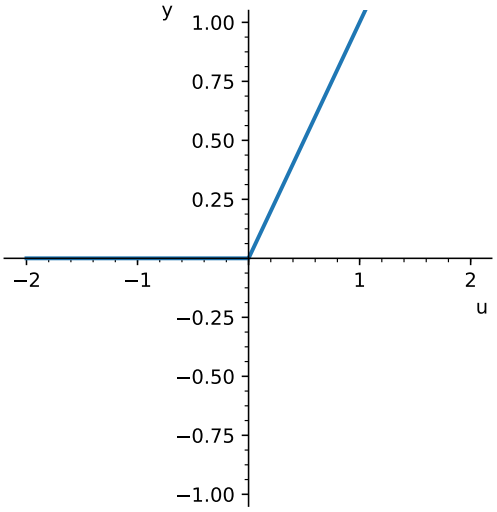


Fig. A.4: The rectified linear unit.

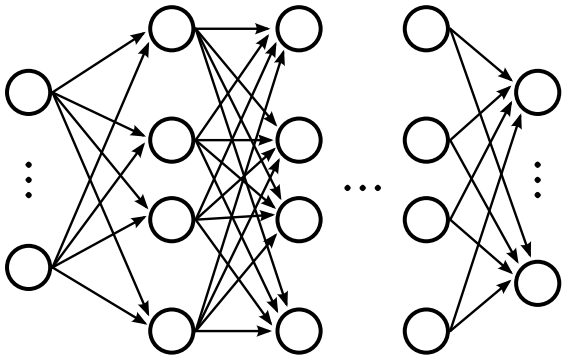


Fig. A.5: A feed-forward, layered neural network.

A.2 | Neural Networks and Universal Approximation

An artificial neuron is not very powerful by itself. However, if we connect a larger number of artificial neurons, we get a neural network. Fig. A.5 shows an illustration of a neural network, in which neurons are arranged into several distinguishable layers. We refer to such networks as layered.

The simple kind of a layered feed-forward neural network shown in Fig. A.5 is sometimes referred to as a multi-layer perceptron (MLP). The MLP architecture is distinguishable by several characteristics: it uses dense layers (the kind that we have talked about so far, as opposed to special layers, such as convolutional layers, that will be discussed later), it is feed-forward (all signals propagate in the same direction – from the inputs to the outputs), neurons in adjacent layers are fully connected, and there are no skip connections (i.e. layer i is directly connected to layers $i-1$ and $i+1$ and to no other layers).

The behaviour of the network depends on the strength of the connections between the individual neurons – that is to say on the weights $\mathbf{w}$ that each neuron assigns to its individual inputs and on the threshold potentials θ. These are the parameters that are tuned when a neural network is learning.

The first layer of a layered network, into which the inputs are fed, is called the input layer. The last layer, from which the outputs are collected, is called the output layer. The layers in between are referred to as hidden layers.

Neural networks are universal approximators, provided that they have at least one hidden layer and non-linear activation functions. This means that they can approximate any

function with a finite number of discontinuities to arbitrary precision [3]. Naturally, this only means that there exists a network, which can approximate the function: it does not mean that we know how many hidden neurons there must be, or what the weights and thresholds should be.

A.3 | Learning in Neural Networks

Artificial neural networks have one useful property: they are differentiable. Given some differentiable loss function, which measures how well the neural network's real output is matching the desired output, this loss function's gradient w.r.t. the networks output can be computed. This gradient can then be backpropagated using the chain rule through all the layers, all the way back to the network's input. Along the way, partial derivatives w.r.t. the weights and threshold potentials of all the artificial neurons can be computed.

Once we are able to compute the gradients w.r.t. all the weights and thresholds in this way, we are able to perform gradient descent, i.e. to iteratively slide down the error surface to a local minimum. There is a number of variations on this process (at present, a method known as Adam is one of the most popular alternatives [99]), but they all ultimately result in a set of weights and thresholds that (are trying to) minimize the errors that the network is making.

A.4 | Deep Neural Networks

The recent success of artificial neural networks in many challenging tasks, can to a large part be attributed to deep neural networks, i.e. to networks with many hidden layers. In order to gain intuition concerning why neural networks with many layers are more expressive, we can consider the following simple example. Let us say that we would like to build a network representation of the following arithmetic expression:

$$[(a+b)+(c+d)]\cdot[(e+f)+(g+h)], \tag{A.3}$$

where a, b, c, d, e, f, g, h are variables.

Fig. A.6 shows a deeper network consisting of function blocks "$+$" and "$\times$", which expresses (A.3). In total, 7 block are necessary to represent the expression.

If we expand all the brackets in (A.3), we obtain an expression, which can be represented using a two-layer network of function blocks:

$$\begin{aligned} &a\cdot e + a\cdot f + a\cdot g + a\cdot h + b\cdot e + b\cdot f + b\cdot g + b\cdot h + \\ &c\cdot e + c\cdot f + c\cdot g + c\cdot h + d\cdot e + d\cdot f + d\cdot g + d\cdot h. \end{aligned} \tag{A.4}$$

As we can see, although it is possible to represent the expression using just two layers in the case of (A.4), we need significantly more function blocks to do so – a visual depiction

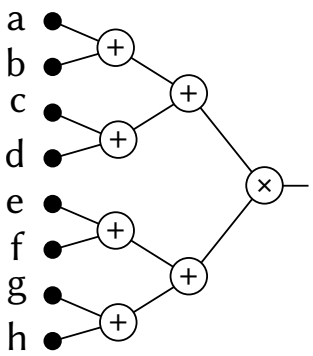


Fig. A.6: Expression (A.3) represented using a deep network. 7 blocks are used.

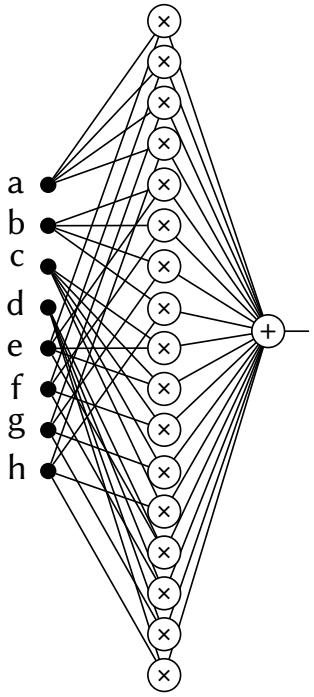


Fig. A.7: Expression (A.3) represented using a shallow network. 18 blocks are used.

of the resulting network is shown in Fig. A.7.

If we apply the same intuition to artificial neural networks, we have a reason to expect that although a neural network with a single hidden layer is, in principle, a universal approximator, a shallow network may need a much bigger number of neurons to represent a function than a deep network. This number can often be unacceptably large and can even increase exponentially [113].

Naturally, with large increases in the number of neurons, the number of parameters that need to be learned also increases dramatically, which impacts the ability of the network to generalize.

A.5 | Deep Learning

Even though it is clear that deep representations are advantageous in that they can create effective factorized representations, until a few years ago there were no effective ways to train them. This was because of vanishing/exploding gradients: Whenever the gradients were backpropagated through a larger number of layers, they would either become extremely small (vanishing), or extremely large (explosion). This caused problems with numerical stability, and ultimately prevented learning.

However, in recent years effective ways to work around the problem have been discovered. These include using:

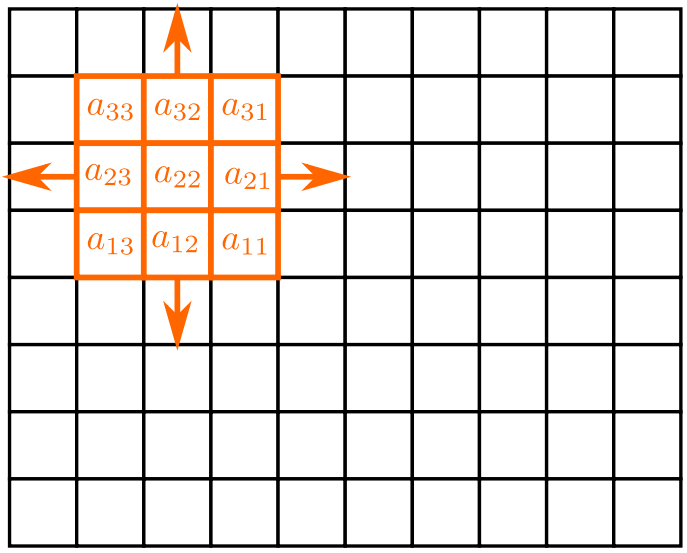


Fig. A.8: A 3×3 convolutional kernel sliding over an image.

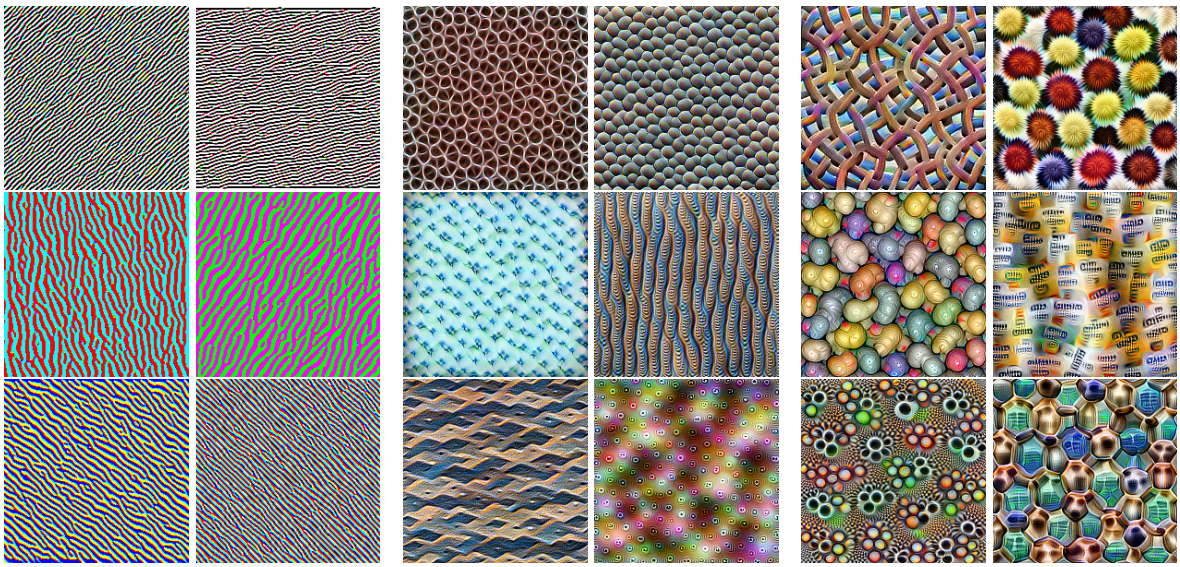
- Activation functions that do not saturate;
- Cleverer ways to initialize the weights at the beginning of training [114, 115];
- Large amounts of data;
- Powerful regularization techniques;
- Special architectural elements, such as convolutional layers and residual blocks [100];
- Other specialized techniques, such as batch normalization [116].

A.6 | Convolutional Neural Networks

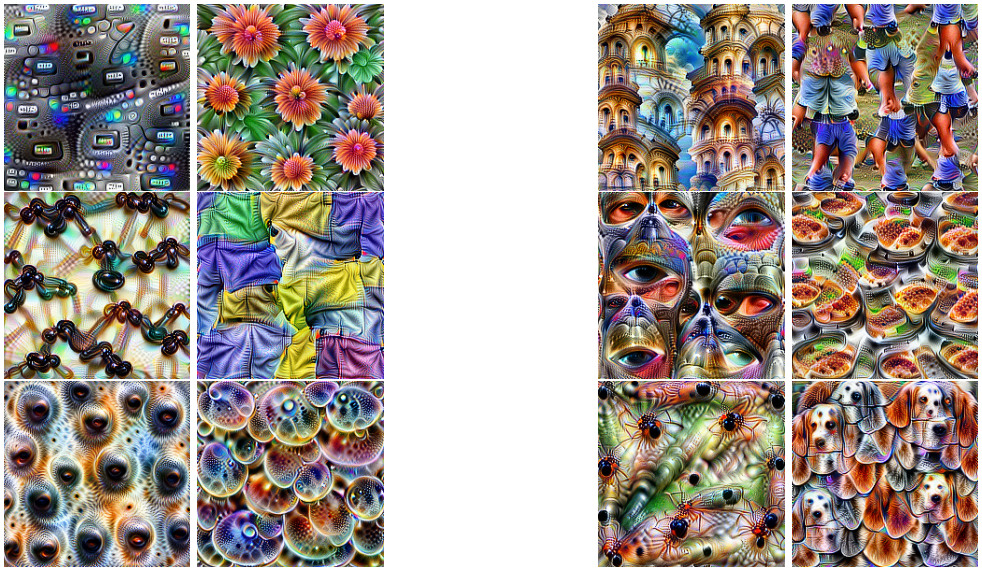
Convolutional layers are one among the special architectural elements that enable deep learning to work on visual and sequential data (images, audio, text, ...). One of the big challenges with visual data, for an instance, is how to make network able to recognize an object no matter where it is positioned, how large it is and how it is rotated. If a neural network can do this, we say that it is invariant to position, size and rotation respectively.

Convolutional networks are very good at providing invariance to position. The intuition is that if we want to train an object detector, which works everywhere in the image, maybe we should just train one and then slide it over the entire image to receive the predictions. The principle is illustrated in Fig. A.8. And, roughly speaking, that is what convolutional layers do. Each convolutional layer has multiple detectors, represented by artificial neurons.

The detectors in the early layers of a network tend to detect simple features of the input image, such as edges. Middle layers learn to detect more complex visual concepts, such as textures, patterns, parts of objects and objects. Fig. A.9 shows images, which maximize the output of selected neurons in multiple layers of the GoogLeNet convolutional architecture [117]. The gradual progression from simple patterns to more complex concepts is apparent.



(a) Edges (layer conv2d0). (b) Textures (layer mixed3a). (c) Patterns (layer mixed4a).



(d) Object parts (layers mixed4b and mixed4c). (e) Objects (layers mixed4d and mixed4e).

Fig. A.9: Visualization of pre-images for neurons from various layers of GoogLeNet [The images are available from [117] under the terms of CC-BY 4.0].

APPENDIX B

ADVANCED APPROACHES IN DEEP REINFORCEMENT LEARNING

Since we wanted to keep chapter 7 as concise as possible and refrain from discussing methods, which were not absolutely necessary to understand the following chapters, we have deferred some of the content to this appendix. We will now go over it in order.

B.1 | Prioritized Experience Replay

The standard formulation of experience replay, which uses a replay buffer from which transitions are sampled uniformly, has several issues, such as:

- As the replay buffer grows, it takes progressively longer to sample recent transitions. This is problem, because they are often much more informative than the old ones, which were generated using a policy that has been updated thousands of times since.
- In environments, where rewards are sparse, there is typically only a handful of highly-informative transitions in the replay buffer at any given time and they have a very small chance of being sampled, if we are sampling uniformly.
- ...

Prioritized experience replay is a technique, which seeks to mitigate this problem [118]. Instead of sampling the transitions uniformly, it prioritizes transitions with large temporal-difference (TD) errors. Naturally, any method that samples from the replay buffer non-uniformly will introduce bias – so they also propose a way to correct for this using importance sampling.

In prioritized experience replay, the probability of sampling transition i is set to [118]

$$p(i) = \frac{\sigma_i^\alpha}{\sum_k \sigma_k^\alpha}, \tag{B.1}$$

where σ_i is the priority of transition i and α is the parameters, which determines how much

the priorities matter. By $\alpha = 0$ we can recover uniform experience replay.

As we have mentioned, priorities are based on the TD errors δ_i. The paper considers two versions of prioritized experience replay [118]:

- *Proportional prioritization:* The priority is proportional to δ_i, i.e.

$$\sigma_i = |\delta_i| + \epsilon, \tag{B.2}$$

  where ϵ is a small positive constant, which ensures that the transition has a chance of being visited even if δ_i happens to be exactly 0.
- *Rank-based prioritization:* The priority is based on the rank $\mathrm{rank}(i)$ of transition i, which is equal to its position when all the transitions are sorted in ascending order according to $|\delta_i|$. The priority is then [118]:

$$\sigma_i = \frac{1}{\mathrm{rank}(i)}. \tag{B.3}$$

B.1.1 Correcting for the Bias

To correct for the bias, prioritized experience replay makes use of importance sampling. The importance sampling weights are as follows [118]:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{p(i)}\right)^{\beta}. \tag{B.4}$$

where N is the size of the replay buffer and β is a parameter, which determines how much correction we want to apply. At $\beta = 1$ we applying the importance sampling fully. However, the idea presented in [118] is that the parameter can be linearly annealed – at the beginning of learning, some bias can be allowed, but later on, it is necessary to correct for it more aggressively.

The importance sampling weight w_i can be folded into the Q-learning update, which then uses $w_i\delta_i$ instead of just δ_i (this then corresponds to weighted importance sampling instead of ordinary importance sampling) [118].

The authors verify the approach by applying it to several tasks, such as Atari games. The empirical evidence suggests that their approach works well and that both – the proportional and the rank-based variant – exhibit similar performance. For notes on how to implement prioritized experience replay in an efficient way, the reader is encouraged to refer back to the original paper [118].

B.1.2 The Blind Cliffwalk Problem

The authors of [118] also proposed a good benchmark problem that illustrates the advantages of prioritized experience replay even more clearly – they call it the *blind cliffwalk problem*.

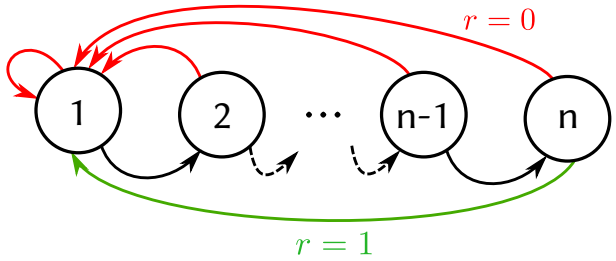


Fig. B.1: The state-action diagram of the blind cliffwalker problem [118].

The structure of the state-action space is shown in Fig. B.1. The agent walks through a series of n states. In each state i it can take one out of two actions. One of the actions leads to state $i + 1$, the other transports the agent back into state 1. The agent receives reward of $r = 1$ when it gets to the final state n from which it is also transported back into state 1. The state representation is such that it offers no generalizable information about which action to pick

In such a setting, rewards are extremely sparse – the agent has to stumble upon the correct sequence of actions by chance before it receives any feedback on how well it is doing. With uniform experience replay the final transition – the only one that carries a non-zero reward – will be placed into the replay buffer. With larger values of n it is very likely that it will take a very long time before it is retrieved and the agent can learn anything from it.

Fig. B.2 illustrates how different approaches will fare on this task. We will suppose that the replay buffer contains the number of samples indicated on the horizontal axis and only one of them has the non-zero reward. The vertical axis shows how many updates are necessary before the agent manages to learn the task. Note the logarithmic axes.

The figure compares several different agents. The oracle agent always performs the update using the sample, which is going to decrease its global loss the most. Naturally, we cannot do this in practice, but it gives us the ideal performance bound that we should be trying to approach. As we can see, uniform sampling performs exponentially worse.

Prioritized experience replay still falls short of the ideal by a wide margin – especially with a greater number of samples – but it already shows a huge improvement on uniform experience replay.

B.2 | Deep Policy Gradients

One of the principal problems with policy gradient methods as described in chapter 6.3, is their low sample efficiency. However, when the policy is represented by a neural network (whether shallow or deep), that makes the problem so much worse that standard PG becomes almost useless.

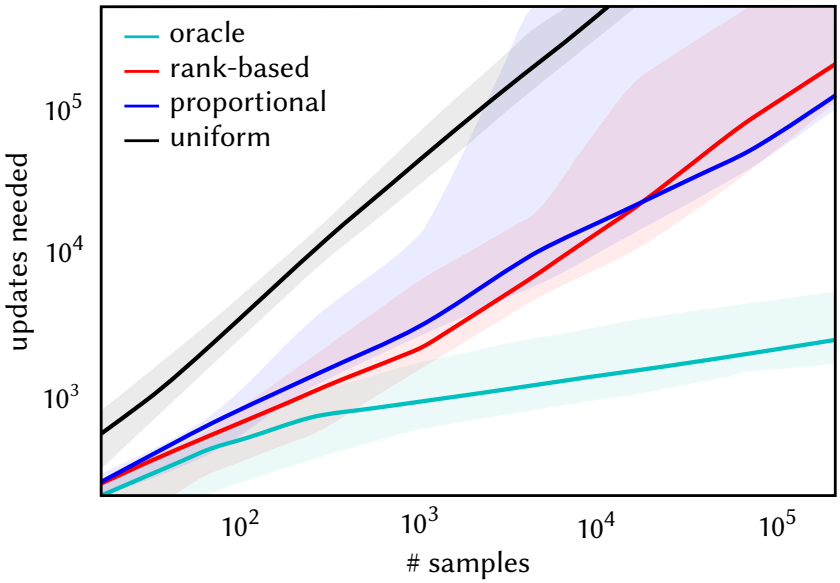


Fig. B.2: The performance of different agents on the blind cliffwalker task [118]. The horizontal axis shows the number of samples in the replay buffer, such that only one of them has the non-zero reward. The vertical axis indicates how many updates each method needs to learn the task. Note the logarithmic axes.

This problem arises because neural networks usually require a lot of successive weight updates to converge. Standard PG methods are on-policy and they need to collect completely new samples after each update. If multiple optimization steps are taken with the same samples, the method no longer follows the correct policy gradient and this often leads to destructive policy updates.

This is because even a small change in the policy parameters θ can cause the policy π_θ to behave very differently. This means that any previously collected samples can give a completely incorrect idea of the expected rewards. We will next describe several recent methods, which alleviate this problem to the point where a PG approach can be used with deep learning in a practical setting.

B.2.1 Trust Region Policy Optimization

One of the first approaches that have successfully managed to make policy gradients work relatively efficiently with deep learning was *trust region policy optimization* (TRPO) introduced in [119].

The essential idea behind TRPO is this: one can make arbitrarily large steps in the parameter space, provided that the policy π_θ does not change drastically enough to invalidate the samples that the update is based on. Thus, if we could constrain the policy so that it does not change too much, this would provide us with a trust region, on which the samples will still be valid. Then we can make large updates robustly, without destroying the policy.

The authors of TRPO start by deriving a theoretically motivated algorithm, which opti-

mizes a certain surrogate objective function. Using the relationship between PG and policy iteration, they are then able to guarantee monotonic improvements to the policy. Finally, they form several approximations to the algorithm so as to arrive at an approach, which can be implemented in practice.

The Surrogate Objective Function

The full process of derivation is described in [119] and we will therefore not go over it in detail here. We will merely reproduce the final surrogate objective, which is as follows [119]:

$$\begin{aligned} \max_{\theta'} \quad & \mathbb{E}_{s\sim\rho^{\pi_\theta}, a\sim\pi_\theta}\left[\frac{\pi_{\theta'}(s,a)}{\pi_\theta(s,a)} Q^{\pi_\theta}(s,a)\right] \\ \text{subject to} \quad & \mathbb{E}_{s\sim\rho^{\pi_\theta}}\left[D_{KL}\left[\pi_\theta(s,\cdot)\,||\,\pi_{\theta'}(s,\cdot)\right]\right] \leq \delta. \end{aligned} \tag{B.5}$$

Let us now discuss this equation in some more detail. First of all – the expectations are approximated using samples. The parameter vector θ corresponds to the old policy and θ' to the new, updated policy. The first part of the equation, which describes the objective, is essentially trying to maximize the expected $Q^{\pi}_{\theta}(s,a)$ by making high-valued actions more probable under the new policy $\pi_{\theta'}$.

When we approximate the expectation over states and actions, we use samples collected using the old policy π^{θ}. In order to correct for the fact that the updated policy $\pi^{\theta'}$ is going to behave differently, we use importance sampling over the actions (hence the $\frac{\pi_{\theta'}(s,a)}{\pi_\theta(s,a)}$). Unfortunately, it is not as easy to correct for the shift in the discounted state distribution ρ^{π_θ}, which will also be different under the new policy.

In essence, this is why we need the constraint. If we can make sure that the policy does not change too much, then the state distribution should also remain reasonably similar. To ensure this, the constraint specifies that the average Kullback-Leibler divergence D_{KL} between the old and the new policy must be smaller than or equal to some δ.

To remind the reader – the Kullback-Leibler (or KL) divergence is a commonly used measure of how one probability distribution differs from another and it is defined as follows [120]:

$$D_{KL}[r(x)\,||\,s(x)] = \mathbb{E}\left[\log\frac{r(x)}{s(x)}\right], \tag{B.6}$$

where $r(x)$ and $s(x)$ are two probability distributions.

How the Objective Function Is Optimized

To optimize the surrogate objective function, the authors apply conjugate gradient descent. At each step, they first compute a search direction, using a linear approximation to the objective and quadratic approximation (exploits the relationship between the Fisher informa-

tion matrix and the KL divergence) to the constraint. Then they perform a line search in that direction, which ensures that the non-linear objective is improved and the non-linear constraint is satisfied [119].

For details, one may again consult the original paper. For a more comprehensive introduction to higher-order optimization methods, the reader may also refer to [121], or to [108], if one prefers a source written specifically with deep learning in mind. The way in which conjugate gradient is applied in TRPO also has its roots in hessian-free optimization and natural gradient descent for deep learning: topics, which were explored earlier in [122–125].

B.2.2 Proximal Policy Optimization

While TRPO works reasonably well in practice, it is rather complex and it can be quite difficult to implement. This is especially true of the optimization procedure based on conjugate gradient, which it uses. Schulman et al. have therefore designed another method, which starts from similar intuitions, but it is designed in a way, which only requires standard gradient-based optimization. The method is called proximal policy optimization (PPO) [126].

Like TRPO, proximal policy optimization also works by optimizing a surrogate objective. However, in PPO's case, there is no constraint – the requirement that the new policy should not be too different from the old policy is incorporated by clipping the original objective function in a clever way.

If we rewrite TRPO's objective in line with the more compact notation from [126], we obtain:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta'}(s_t, a_t)}{\pi_\theta(s_t, a_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[\sigma_t(\theta) \hat{A}_t \right]. \tag{B.7}$$

where $\hat{\cdot}$ denotes approximation and $\sigma_t(\theta)$ is the importance sampling ratio $\sigma_t(\theta) = \frac{\pi_{\theta'}(s_t,a_t)}{\pi_\theta(s_t,a_t)}$ and $\hat{A}_t$ is the approximate advantage.

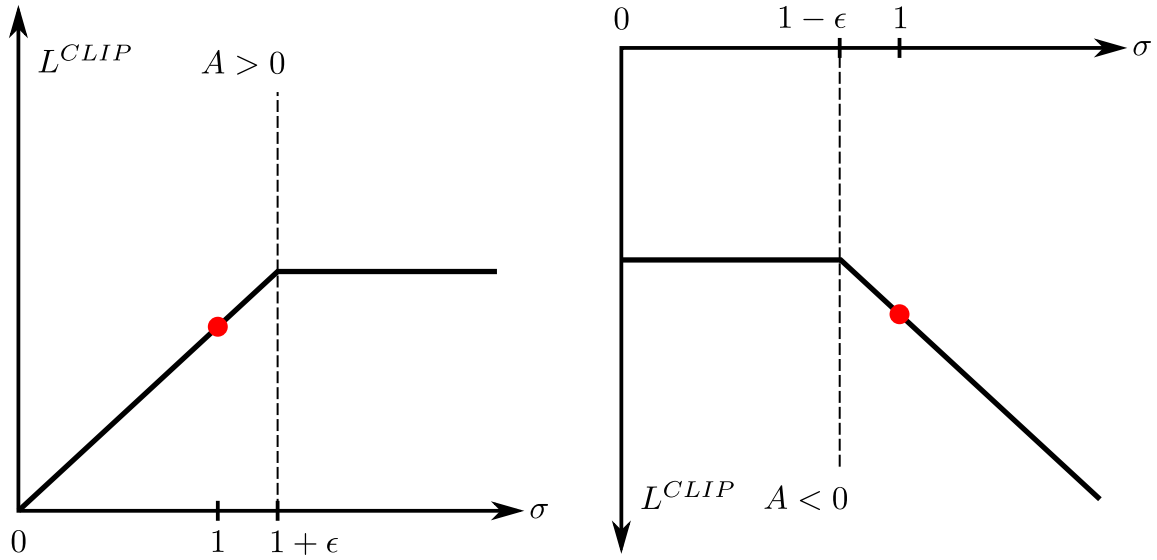
Note that ratio $\sigma_t(\theta)$ does, in fact, also provide an indication of how far $\pi_{\theta'}$ is from π_θ. The authors of [126] make use of this fact: they dispense with the constraint from TRPO and instead choose to penalize changes, which move $\sigma_t(\theta)$ too far from 1.

To this end, they introduce the following new objective [126]:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(\sigma_t(\theta)\hat{A}_t,\ \mathrm{clip}(\sigma_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right], \tag{B.8}$$

where ϵ is a hyperparameter (the authors suggest e.g. $\epsilon = 0.2$). Intuitively, clipping removes the incentive to move σ_t outside the interval $[1-\epsilon, 1+\epsilon]$ and thus diverge too far from the old policy.

Why is this true? When maximizing the objective, we only change the parameters of the



(a) For a positive advantage.

(b) For a negative advantage.

Fig. B.3: The clipping in PPO's surrogate objective function. The red circle indicates the starting point, i.e. $\sigma = 1$ [126].

new policy θ'. This means that the only thing we can influence is $\pi_{\theta'}(s_t, a_t)$ and indirectly σ_t. We try to make actions with high advantages more probable under the new policy. This means that the probability $\pi_{\theta'}(s_t, a_t)$ under the new policy will be larger than probability $\pi_{\theta}(s_t, a_t)$ under the old policy. Consequently, σ_t will be greater than one. The converse is true for actions with low advantages.

If we clip the ratio σ_t, it means that moving it outside of the clipped range will not increase the objective any further – we will still only be multiplying the advantage $\hat{A}_t$ by $1+\epsilon$ at most. Moreover, taking the minimum of the clipped and unclipped objective ensures that we are forming a lower bound (a pessimistic estimate) of the objective. If the clipped objective is smaller than the unclipped objective, we take the clipped value – but if it is larger, we keep to the value of the unclipped objective.

The clipping is further illustrated in Fig. B.3: Fig. B.3a shows the case when the advantage is positive and Fig. B.3b when it is negative. As shown, in either case we make sure that moving σ out of the clipping interval does not yield any further improvements.

The greatest advantage of using this clipped objective $L^{CLIP}(\theta)$ over the TRPO objective should be obvious. It does not require complex optimization methods such as natural gradient descent or conjugate gradient descent and can instead be optimized by taking multiple steps of regular gradient descent (or by any one of the more advanced first-order methods, such as Adam).

The Actor-Critic Style and the Entropy Exploration Bonus

The surrogate objective $L^{CLIP}(\theta)$ can be further augmented if PPO is used as an actor-critic algorithm. In that case in addition to the policy, the method also learns the state-value function $V_\theta(s)$. In such cases it is often advantageous to share parameters between the policy and the value function (e.g. if image recognition is required to select good actions, similar visual features will probably be required to estimate state-values). Parameter sharing requires that an additional term is added into the objective function. Moreover, an entropy term can be included as well to encourage exploration. This leaves us with a modified surrogate objective

$$L^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 \mathrm{H}(\pi_\theta(s_t, \cdot)) \right], \tag{B.9}$$

where c_1 and c_2 are coefficients, term $L^{VF}(\theta)$ corresponds to the squared-error difference between the learned state-value and the target state-value:

$$L^{VF}(\theta) = (V_\theta(s_t) - V_{\text{targ}})^2, \tag{B.10}$$

and $H(\pi_\theta(s_t, \cdot))$ is the entropy term

$$\mathrm{H}(\pi_\theta(s_t, \cdot)) = \mathbb{E}_{a \sim \pi(s_t, \cdot)} \left[-\log \pi(s_t, a) \right]. \tag{B.11}$$

Having an approximation of the state-value function, we can now estimate the advantage $\hat{A}_t$. Paper [126] suggests running the policy for τ time steps (such that $\tau \ll T$, where T is the episode length) and approximating the advantage using [126]:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{\tau-t+1}\delta_{\tau-1}, \tag{B.12}$$

where $\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)$. Recall that the advantage expresses how much better an action is than average; δ_t is a point estimate of this, because it compares the immediate reward r_t, received after performing the action, with the expected immediate reward, as expressed by $\gamma V_\theta(s_{t+1}) - V_\theta(s_t)$.

Note also, that the approximation is computed over τ time steps. We are, in fact, running a τ-step algorithm in the forward view (we really wait for τ steps before making an update, as opposed to backward-view algorithms such as eligibility traces, which would make an incremental update at every time step). This approach also works well with recurrent policies according to [126].

The pseudocode of this actor-critic version of PPO is shown in Algorithm 4.

B.2.3 Deep Deterministic Policy Gradient

The deep deterministic policy gradient (DDPG) method can be thought of as an actor-critic extension of the DQN. Essentially, DDPG still trains a Q-function approximation, but in addition, it also trains an actor network, which tries to maximize this Q-function.

Algorithm 4: Pseudocode of PPO in the actor-critic style [126].

```
for iteration = 1, 2, ... do
    for actor = 1, 2, ..., N do
        Run policy π_θold in environment for τ time steps;
        Compute advantage estimates Â_1, ..., Â_τ;
    end
    Optimize surrogate L w.r.t. θ, with K epochs and mini-batch size M ≤ τN;
    θ_old ← θ;
end
```

For some time, methods based on deterministic policy gradients did not receive much consideration. It was, in fact, thought that the deterministic policy gradient did not exist, or that it would at least not be available to a model-free method [61]. However, as paper [61] shows, the deterministic policy gradient not only exists, but it also has a convenient form, which happens to correspond to the limiting case of the stochastic policy gradient as policy variance tends to zero [61].

The main issue with most stochastic policy gradient approaches such as REINFORCE, TRPO and PPO is that they are on-policy. This makes them less sample efficient, because after every change to the policy, all the old samples have to be discarded and the information contained in them can not be put to any further use.

DDPG – much like DQN – is an off-policy method and uses a replay buffer. It reuses the old samples, which is critical for complex tasks.

Training the Critic

The critic network is trained in much the same way as the Q-network in the DQN algorithm. The most notable difference is in the architecture of the network. While the DQN receive the current observation on its input and outputs the value of each action, DDPG also works for continuous actions spaces, where this approach is not feasible. As a result, the critic receives both – the observation and the action – and only outputs one action-state value.

Training the Actor

The actor is trained using the off-policy deterministic policy gradient from [61]:

$$\nabla_\theta J_{\pi'}(\theta) \approx \mathbb{E}_{s\sim\rho^{\pi'}}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)}\right], \tag{B.13}$$

where $\pi_\theta(s)$ is the deterministic policy implemented by the actor; it maps to an action, not to a probability. In contrast to that, π' is a stochastic exploration policy, which the agent actually follows during training; $\pi'(s,a)$ denotes the probability of taking action a in state s under that policy.1

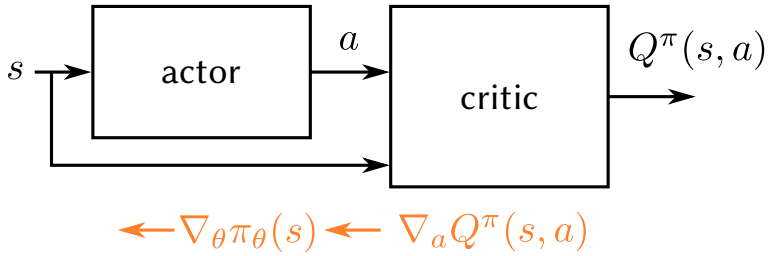


Fig. B.4: Training the actor in DDPG: to maximize the value, we backpropagate through the critic and then through the actor.

Recall that $\rho^{\pi'}(s')$ denotes the (improper) discounted state distribution:

$$\rho^{\pi'}(s') =_{\text{def}} \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{I}(s) p(s \to s', t, \pi) \mathrm{d}s, \tag{B.14}$$

where $p(s \to s', t, \pi)$ is the density at state s' after transitioning from state s for t time steps under policy π' and $\mathcal{I}(s)$ denotes the initial state distribution.

Note that what equation (B.13) actually says, is that we need to backpropagate the gradient from the Q-function back to the actor network (this is what is happening when we multiply the two gradients). This allows us to move the action in the direction, which increases the value most steeply. The idea is illustrated in Fig. B.4.

The actor and the critic are both trained simultaneously on mini-batches drawn from the replay buffer. Just like the DQN, DDPG also uses target networks – this time for both: the actor and the critic. Unlike the original DQN paper, the authors of [127] suggest updating both targets networks softly.

They also found that using batch normalization (see [116]) in the architecture helped – especially in tasks, where different dimensions of the input were scaled differently.

Adding Noise to Achieve Exploration

Since DDPG itself is looking for a deterministic policy, to ensure adequate exploration, it is necessary to add some noise to the actions. The original paper [127] suggests using an Ornstein-Uhlenbeck process to generate the noise. In many continuous control tasks it is best if actions change as smoothly as possible. Action stuttering can cost extra energy and – more importantly – it can also cause unnecessary wear to the hardware. The advantage of the Ornstein-Uhlenbeck process is that the noise generated by it is temporally correlated and it changes smoothly as a result.

If the application is such that action stuttering is not a major problem, noise can also be generated using a Gaussian – which is much easier to implement.

Stability and Sensitivity

The off-policy nature of DDPG and its resulting higher sample efficiency present a huge advantage in comparison to methods such as PPO. However, DDPG has a huge problem: much like DQN itself, it is known to be relatively unstable and it is extremely sensitive to hyperparameters, which makes it difficult to use.

B.2.4 Soft Actor-Critic

As we have mentioned, two different groups of methods have formed in deep RL and they have somewhat opposed properties. There are the stochastic policy gradient methods, which are relatively stable and well-behaved. However, they are also on-policy, which means that their sample complexity is very high. Then there is a group of off-policy methods, such as DDPG, which have better sample-efficiency, but they are extremely fragile and overly sensitive to hyperparameters.

Soft actor-critic (SAC) is a method, which is trying to overcome the problem. It is an off-policy method and uses all the tricks from DQN and DDPG such as the replay buffer and the target network. However, the policy it actually trains is stochastic and not only that – in addition to maximizing the return, the algorithm is also trying to maximize the policy's entropy (there is a similarity to the entropy bonus in PPO).

The idea is that while learning, SAC will tend to explore more fully. This will not only make it more robust in comparison to methods like DDPG – it also often helps to discover and retain more than one way of reaching the goal, which can help substantially when transferring to a different, but related task. Also, while the framework tends to explore more widely, it can also abandon behaviours, which are clearly unpromising [89].

Soft Policy Iteration

As we have already mentioned, in the maximum-entropy reinforcement learning framework the goal is to not only maximize the long-term return, but also the entropy of the policy π. This can be stated formally as follows [89]:

$$J(\pi) = \sum_{t=0}^{T-1} \mathbb{E}_{(s_t,a_t)\sim\rho^\pi} \left[r(s_t, a_t) + \alpha \mathrm{H}(\pi(s_t, \cdot)) \right], \tag{B.15}$$

where $\mathrm{H}(\pi(s_t, \cdot))$ is the entropy of policy π:

$$\mathrm{H}(\pi_\theta(s_t, \cdot)) = \mathbb{E}_{a\sim\pi(s_t,\cdot)} \left[-\log \pi(s_t, a) \right]. \tag{B.16}$$

and α is the coefficient, which determines the relative importance of the rewards and the entropy. This coefficient can, in fact, be dropped without any loss of generality, because the same effect can be achieved by rescaling the rewards.

This translates into the following definition for the state-value function:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi(s_t,\cdot)} \left[Q(s_t, a_t) - \log \pi(s_t, a_t) \right]. \tag{B.17}$$

It is, of course, also possible to state the infinite-horizon discounted version of these equations, but they are slightly more involved. Based on these definitions, paper [89] first derives a theoretical algorithm – *soft policy iteration* – for the tabular case and proves that it converges to the optimal policy (optimal under the soft RL criterion). The authors then proceed to derive an approximate version of this, which can be used in practice.

One of the key ideas of the approach is as follows – the algorithm learns the action-value function $Q^\pi(s, a)$ of the current policy π, which can be done off-policy. This action-value function should then be used to improve the policy. Ideally, we would like the new policy to assign the highest probabilities to the actions with the highest action-values. It is also true that all the probabilities should sum up to 1.

If we think about this in terms of the softmax distribution, we could formulate the desired action probability under the improved policy π' as:

$$p_{\text{desired}}(a|s_t) = \frac{e^{Q^\pi(s_t,a)}}{\sum_{a' \in \mathcal{A}} e^{Q^\pi(s_t,a')}}. \tag{B.18}$$

We can do this, because we are still considering the discrete tabular case. However, the same formulation can be extended to the continuous formulation as well. At this point, we do not want to worry too much about the denominator (the normalizing term) – there is a trick which will help us to get rid of it. We will therefore denote it with $Z^\pi(s)$ and rewrite as follows:

$$\begin{aligned} p_{\text{desired}}(a|s_t) = \frac{e^{Q^\pi(s_t,a)}}{Z^\pi(s_t)} &= \frac{e^{Q^\pi(s_t,a)}}{e^{\log Z^\pi(s_t)}} \\ &= \exp(Q^\pi(s_t, a) - \log Z^\pi(s_t)). \end{aligned} \tag{B.19}$$

It is understood that our learned policy will come from a certain policy class Π. This can, for an instance, be the class of all possible Gaussian policies. The policy class does not have to be expressive enough to represent the desired probability distribution p_{desired} exactly – we will need to do a projection of p_{desired} into the policy space Π. We can do this by minimizing the Kullback-Leibler divergence between p_{desired} and our new improved policy $\pi' \in \Pi$:

$$\pi' = \arg\min_{\tilde{\pi} \in \Pi} D_{KL} \left[\tilde{\pi}(s_t, \cdot) \,||\, \exp(Q^\pi(s_t, \cdot) - \log Z^\pi(s_t)) \right]. \tag{B.20}$$

This will ensure that our new improved policy π' is as close as possible to p_{desired} while still belonging to policy class Π.

With this kind of setup, the authors of [89] were able to prove that soft policy iteration will converge in the tabular case.

The Practical Algorithm

The practical algorithm – the soft actor-critic – is learning 3 things in parallel: the action-value function $Q_\theta(s,a)$, the state-value function $V_\psi(s)$ and the policy $\pi_\phi(s,a)$.

This makes it possible to specify the targets for the action-value function in terms of the state-values and vice versa. Again, the actual training of both is similar to the DQN. A mini-batch of transitions is sampled from the replay buffer and targets are formed. The targets for the state-value function are formed according to [89]:

$$\begin{aligned} y_{V,t} &= Q_\theta(s_t, a') - \log \pi_\phi(s_t, a') \\ a' &\sim \pi_\phi(s_t, \cdot) \end{aligned} \tag{B.21}$$

and the squared errors are minimized. Note that the action used to compute the target is not the one recorded in the replay buffer – it is the one sampled from the current policy. This means that we will be pushing $V_\psi(s)$ towards the expected action-value under the current policy π_ϕ plus the entropy term (compare this to equation (B.17)).

Similarly, for the action-value function we have [89]:

$$y_{Q,t} = r_t + \gamma V_{\bar{\psi}}(s_{t+1}), \tag{B.22}$$

where $V_{\bar{\psi}}$ is the target network, which is updated softly like the target network in DDPG.

Finally, for the policy we are still trying to minimize the Kullback-Leibler divergence, so our loss function can be described as [89]:

$$J^\pi(\phi) = D_{KL}\left[\pi_\phi(s+t,\cdot) \,||\, \exp(Q_\theta(s_t,\cdot) - \log Z_\theta(s_t))\right]. \tag{B.23}$$

If the policy class is simple, such as a Gaussian, this can be minimized in DDPG style – by directly backpropagating the gradients from the critic to the actor. Since random number generators are not differentiable, this requires the use of the reparametrization trick*.

The paper also proposes using a more complex policy class – *Gaussian mixtures*. This enables the agent to learn multimodal policies (the probability density function has several local maxima). For policy classes such as this, the reparametrization trick cannot be used, so they instead propose the use of a likelihood gradient estimator. They pick a baseline, which conveniently gets rid of the logarithm of the partition function $\log Z_\theta(s_t)$. Thus, they end up with the following gradient estimator [89]:

$$\hat{\nabla}_\phi J^\pi(\phi) = \nabla_\phi \log \pi_\phi(s_t, a_t)\left(\log \pi_\phi(s_t, a_t) - Q_\theta(s_t, a_t) + V_\psi(s_t)\right), \tag{B.24}$$

which can be used even when the reparametrization trick cannot be applied.

*The randomness enters the network as an additional input, which the network only transforms into the appropriate distribution by differentiable operations.

PRÍLOHA C

RESUMÉ V SLOVENSKOM JAZYKU

Jedným z dlhodobých cieľov v oblasti umelej inteligencie je návrh inteligentných agentov. Nejde, prirodzene, o jednoduchú – a rozhodne ju neuľahčuje fakt, že stále nevládne konsenzus o tom, čo to vôbec (umelá) inteligencia je. O nič lepšie to nie je s pojmom agent. Ak však necháme bokom filozofické otázky tohto typu, návrh inteligentných agentov je mimoriadne široký, multidisciplinárny problém. So súčasnými pokrokmi v oblasti hlbokého učenia, počítačového videnia, učenia s odmenou, spracovania a generovania prirodzeného jazyka a reči, sme sa k tomuto cieľu priblížili bližšie než kedykoľvek predtým – napriek tomu však platí, že pred nami stále stoja veľké výzvy.

Máme k dispozícii systémy hlbokého učenia, ktoré dosahujú silné výsledky – napríklad aj v spracovaní obrazových a audio dát. Existujú systémy, ktoré vedia na vysokej úrovni realizovať rozpoznávanie obrazu a detekciu objektov, lenže si na to vyžadujú obrovské množstvá manuálne anotovaných dát. Pri súčasnom stave poznania nie je navyše zaručené, že budú vedieť poskytnúť dostatočné záruky ohľadom správnosti výsledkov a bezpečnosti.

Máme k dispozícii vynikajúce systémy na rozpoznávanie reči, ktoré však majú stále problémy s neideálnymi akustickými podmienkami, prípadne s netypickými rečníkmi. Sme schopní trénovať čoraz silnejšie jazykové modely a systémy strojového prekladu – lenže kým neexistujú systémy, ktoré by boli schopné logicky uvažovať o obsahu textu, s ktorým pracujú, dajú sa ďalej zdokonaľovať len po určitú hranicu. Ešte kritickejším problémom je možno to, že stále nevieme ako ukotviť prirodzený jazyk – t.j. zabezpečiť, že učiaci sa systém bude nielen modelovať jeho štruktúru, ale mu bude vedieť priradiť aj zmysel: t.j. spojiť si symboly s určitými konkrétnymi konceptami, ktoré pozná prostredníctvom vlastnej skúsenosti.

Okrem schopnosti vnímať a komunikovať musí mať inteligentný agent samozrejme aj schopnosť rozhodovať sa a konať – t.j. zvoliť akcie, ktoré naplnia jeho ciele a potreby. Keď je to potrebné, musí byť schopný robiť krátkodobé a dlhodobé plánovanie. A samozrejme,

akcie, ktoré zvolil, musí byť schopný aj vykonať.

Sľubným frameworkom, ktorý by mohol byť schopný spojiť veľa z už spomenutých komponentov do jedného systému – a to spôsobom, akým to doteraz nebolo možné – je hlboké učenie s odmenou. To má však samozrejme tiež svoje vlastné otvorené problémy, napríklad nízku vzorkovú efektívnosť existujúcich metód, ktorým môže na niektorých úlohách učenie trvať povedzme aj 200 rokov simulovaného času alebo viac.

Ďalším problémom je, že vo svojej štandardnej formulácii vyžaduje učenie s odmenou odmeňovaciu funkciu, ktorá by agentovi dávala pozitívnu spätnú väzbu za vykonanie požadovaného správania a negatívnu spätnú väzbu za nekorektné správanie. Pre niektoré úlohy je návrh takej funkcie porovnateľne zložitý s navrhnutím požadovaného správania ručne.

Niekoľko dôležitých objavov v tomto odbore však poskytuje dôvody na opatrný optimizmus – zdá sa, že prinajmenšom niektoré z týchto problémov sa (v najbližšom období) podarí úspešne vyriešiť. Ako príklad možno spomenúť prácu na zefektívňovaní učenia s odmenou prostredníctvom predtrénovania agentov alebo niektorých ich komponentov tak, aby sa neskôr nemuseli cieľovú úlohu učiť od nuly. Tieto prístupy zahŕňajú napríklad: *(a)* učenie sa reprezentácie, ktorá by agentovi poskytla predučenú schopnosť vnímať relevantné príznaky v rámci danej úlohy; *(b)* meta učenie s odmenou, t.j. agenti učiaci sa ako sa učiť; *(c)* intrinsická motivácia, napr. zvedavosť, ktorá pomáha agentovi naučiť sa užitočné typy správania sa aj bez vonkajších odmien a pod.

V tejto práci si kladieme za cieľ venovať sa problému návrhu inteligentných agentov pomocou hlbokého učenia s odmenou. Naše hlavné hypotézy sú nasledujúce:

1. **Kognitívne preferencie prostredníctvom návrhu architektúry:** Domnievame sa, že do agenta sa dá prostredníctvom návrhu hlbokej architektúry zaniesť množstvo užitočných poznatkov o štruktúre úlohy a o tom, ako treba pri jej riešení postupovať. Tieto poznatky tak môžu poslúžiť na vytvorenie užitočných kognitívnych preferencií pre agenta.
2. **Abstrakcia a pozornosť:** Jedným zo spôsobov, ako agentovi sprostredkovať užitočné kognitívne preferencie, je aj zabudovať do jeho architektúry mechanizmy ako sú abstrakcia a vizuálna pozornosť, ktoré vedia zvýšiť jeho efektivitu.
3. **Multisenzorické učenie s odmenou:** Hlboké učenie s odmenou je dostatočne flexibilné na to, aby bolo schopné zvládnuť multisenzorické toky dát (napr. audio, obrazové dáta, tabuľkové dáta, prirodzený jazyk, ...). Podobnú flexibilitu by bolo v minulosti bývalo veľmi ťažké dosiahnuť pomocou ľubovoľných iných metód. Chceme však ukázať, že aby sa dali tieto možnosti naplno využiť, je znovu potrebné poskytnúť agentovi vhodné kognitívne preferencie voľbou príslušnej architektúry.

Je pochopiteľné, že sa takýmto a súvisiacim otázkam už dlhšie venujeme, keďže hlavným zameraním nášho pracoviska je už veľa rokov oblasť automatizácie, ktorá je v súčasnosti

intenzívne informovaná pokrokov v strojovom učení a umelej inteligencii. Dá sa dokonca povedať, že pokrok, ktorý posledných niekoľko rokov v oblasti automatizácie pozorujeme, by bol nepredstaviteľný bez nedávnych prelomov v daných oblastiach.

C.1 | Hlboké učenie

Hlboké modely sa v súčasnosti využívajú prakticky vo všetkých úspešných systémoch na učenie s odmenou. Uvedieme preto aj v tomto resumé najprv aspoň krátky úvod o tom, čo je to hlboké učenie a v čom spočívajú jeho hlavné myšlienky.

Umelé neurónové siete patria medzi známe oblasti strojového učenia už viacero desaťročí. Ich výsledky však tradične neboli schopné významnejším spôsobom prekonať iné, často podstatne jednoduchšie metódy strojového učenia. Ako sa nedávno ukázalo, tento problém súvisel s viacerými faktormi, ktoré znemožňovali úspešný tréning sietí s hlbokými architektúrami (t.j. takými, kde umelá neurónová sieť má oveľa viac vrstiev než tradičné 3 – povedzme aj viac než 50).

Ku prelomu došlo pomerne nedávno – postupne sa identifikovalo viacero dôvodov, pre ktoré hlboké učenie v minulosti nefungovalo. Patria medzi ne napr.:

- nedostatočné množstvo tréningových dát;
- nevhodné aktivačné funkcie;
- nevhodná inicializácia váh;
- nedokonalé architektúry;
- ...

C.1.1 V čom je výhoda hlbokých architektúr

Nie je ťažké intuitívne pochopiť, v čom je výhoda hlbokých architektúr. Jedna vec, ktorú si je dobré uvedomiť je, že hlboké neurónové siete sú schopné naučiť sa automaticky si predspracovať dáta a extrahovať z nich užitočné príznaky. Hlboká sieť bude potom niekoľko prvých vrstiev používať na tento účel, zatiaľ čo zvyšné vrstvy budú realizovať samotnú úlohu – napríklad rozpoznávať objekty v obraze.

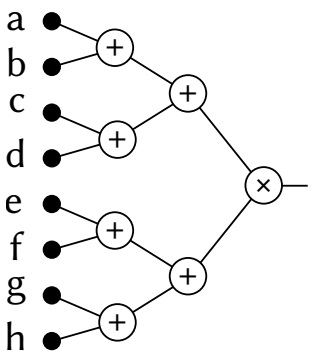
Výhody viacvrstvových reprezentácií je však možné pochopiť aj pomocou nasledujúceho jednoduchého príkladu. Predstavme si, že máme vytvoriť sieť reprezentujúcu nasledujúci aritmetický výraz:

$$[(a+b)+(c+d)]\cdot[(e+f)+(g+h)], \tag{C.1}$$

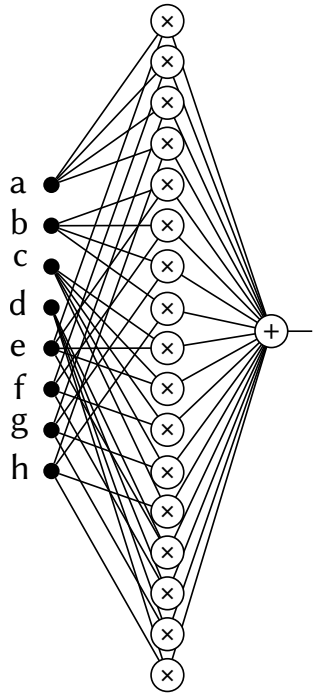
kde a, b, c, d, e, f, g, h sú premenné.

Obr. C.1 ukazuje hlbšiu sieť pozostávajúcu z funkčných blokov „$+$“ a „$\times$“, ktorá vyjadruje (C.1). Celkovo je na reprezentáciu výrazu v tomto prípade potrebných 7 blokov.

Ak však roznásobíme všetky zátvorky v (C.1), získame výraz, ktorý je možné reprezen-



Obr. C.1: Výraz (C.1) reprezentovaný pomocou hlbokej siete. Potrebných je 7 blokov.



Obr. C.2: Výraz (C.1) reprezentovaný pomocou plytkej siete. Potrebných je 18 blokov.

tovať sieťou funkčných blokov s dvomi vrstvami:

$$\begin{aligned} a \cdot e + a \cdot f + a \cdot g + a \cdot h + b \cdot e + b \cdot f + b \cdot g + b \cdot h + \\ c \cdot e + c \cdot f + c \cdot g + c \cdot h + d \cdot e + d \cdot f + d \cdot g + d \cdot h. \end{aligned} \tag{C.2}$$

Ako vidno, hoci je výraz možné pomocou zápisu (C.2) reprezentovať len dvomi vrstvami, bude na to potrebné podstatne väčšie množstvo blokov – vizualizácia výslednej siete je na Obr. C.2.

Ak tú istú intuíciu aplikujeme na umelé neurónové siete, máme dôvod očakávať, že hoci plytká neurónová sieť bude možno schopná reprezentovať tú istú funkciu ako hlboká sieť, bude na to pravdepodobne potrebovať omnoho väčší počet neurónov. Tento počet bude typicky neprijateľne vysoký a môže narastať až exponenciálne [113].

Podstatným problémom je, že s nárastom počtu neurónov narastá prudko aj počet parametrov, ktoré sa sieť musí naučiť. To máva samozrejme výrazne negatívny dopad na jej schopnosť správne zovšeobecňovať.

C.1.2 Špeciálne architektúry

Jedným z kľúčových aspektov pri aplikácii hlbokých neurónových sietí je voľba vhodnej architektúry. Ako bolo povedané už vyššie, voľbou architektúry je možné do siete zakódovať určité kognitívne preferencie: nasmerovať ku určitému spôsobu, ako problém riešiť.

Zrejme najznámejším typom špeciálnej architektúry sú konvolučné vrstvy, ktoré sa po-

užívajú v spracovaní obrazu. Ich princíp spočíva v tom, že ten istý filter (realizovaný konvolučným jadrom) aplikujú na všetky pozície v obraze, čím sa dosahuje invariancia na pozíciu: ten istý príznak je možné detegovať, nech sa v obraze nachádza kdekoľvek.

Ak sa konvolučných vrstiev uloží viacero za sebou, kognitívne preferencie sieť vedú k tomu, aby sa snažila vo vstupnom obraze hľadať najprv lokálne závislosti – t.j. jednoduché vzory a z nich postupne skladať čoraz zložitejšie a abstraktné koncepty. Vizuálne ilustrované je to na Obr. C.3, kde vidno aké typy príznakov detegujú neuróny v sieti GoogLeNet [117]. Vidno, že skoršie vrstvy detegujú hrany, potom jednoduché textúry, zložitejšie vzory, časti objektov atď.

Existujú prirodzene aj iné špeciálne architektúry, napr. dlhá krátkodobá pamäť (angl. long short-term memory; LSTM), ktorá sa využíva keď sieť na riešenie úlohy potrebuje pamäť, rôzne typy architektúr využívajúce sekvenčnú pozornosť, či napríklad reziduálne architektúry, ktoré napomáhajú učenie vo veľmi hlbokých sieťach.

C.2 | Učenie s odmenou

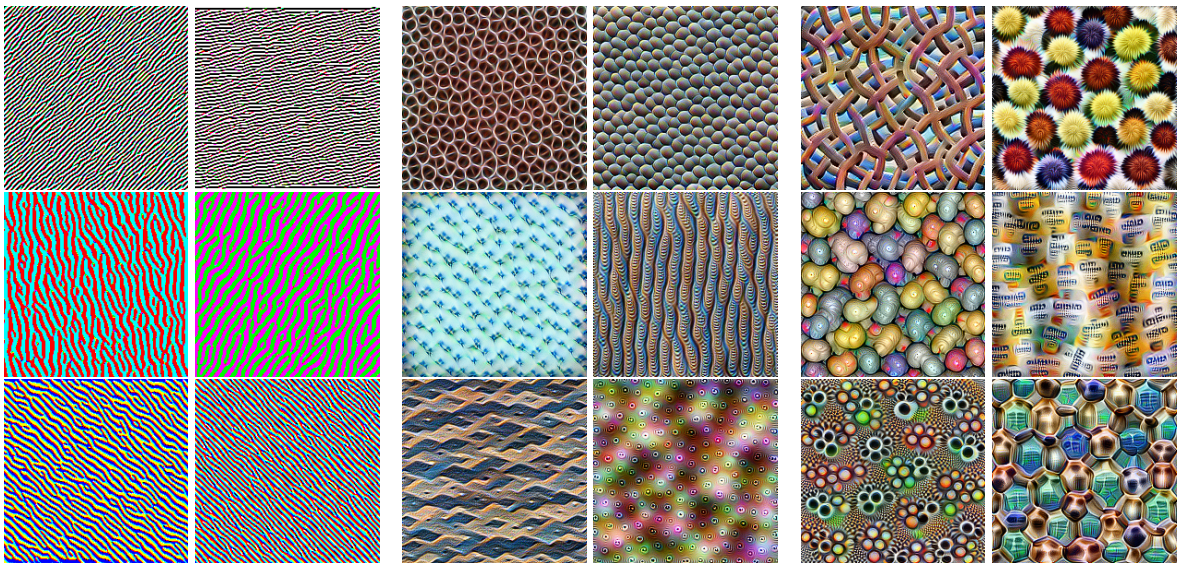
Učenie s odmenou je typ úlohy strojového učenia, kde učiaci sa systém v čase vykonáva rozhodnutia a nie je (na rozdiel od kontrolovaného učenia) vopred jasne stanovené, ako by mal presne reagovať na každý jeden vstup. Je iba k dispozícii odmeňovacia funkcia $r(s_t, a_t)$, ktorá učiacemu sa systému (agentovi) dáva pozitívnu spätnú väzbu za korektné a negatívnu za nekorektné správanie. Symboly s_t a a_t označujú stav prostredia v časovom kroku t a akciu, ktorú agent v tom istom časovom kroku zvolil (v uvedenom poradí).

Cieľom agenta je nájsť stratégiu správania, ktorá by z dlhodobého hľadiska maximalizovala získané odmeny. Tento cieľ sa dá formálne vyjadriť rôznymi spôsobmi – napríklad aj nasledujúcou jednoduchou formuláciou (model s nekonečným horizontom a znehodnotením) [60, 61]:

$$\theta^* = \arg\max_{\theta} \mathbb{E}\left\{ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \middle| \pi_\theta \right\}. \tag{C.3}$$

kde $\pi_\theta(s_t, a_t)$ je stratégia, ktorou sa agent riadi – určuje pravdepodobnosť, že agent v state s_t zvolí akciu a_t. Stratégia π_θ je parametrizovaná parametrickým vektorom θ. Naším cieľom je nájsť taký parametrický vektor, ktorý by maximalizoval očakávanú hodnotu dlhodobého súčtu odmien – za predpokladu, že sa agent riadi stratégiou π_θ.

Symbol r_t označuje okamžitú odmenu, ktorú agent dostane v čase t, t.j. $r_t = r(s_t, a_t)$. Všimnime si, že vo výraze $\sum_{t=0}^{\infty} \gamma^t r_{t+1}$ je každý člen násobený mocninou konštanty $\gamma \in (0, 1\rangle$ – ide o tzv. mieru znehodnotenia, ktorá určuje, že vzdialeným odmenám sa prikladá menší význam než tým blízkym. Modeluje to známy princíp „lepší vrabec v hrsti ako holub na streche“ – o tom, či bude agent schopný získať odmeny, ktoré sú v čase veľmi vzdialené panuje omnoho väčšia neistota než o blízkych odmenách.



(a) Hrany (vrstva conv2d0). (b) Textúry (vrstva mixed3a). (c) Vzory (vrstva mixed4a).

(d) Časti objektov (vrstvy mixed4b a mixed4c). (e) Objekty (vrstvy mixed4d a mixed4e).

Obr. C.3: Vizualizácia predobrazov pre neuróny z rôznych vrstiev GoogLeNet [Obrázky sú dostupné z [117] pod licenciou CC-BY 4.0].

Očakávaná hodnota prirodzene berie do úvahy nielen stochasticitu stratégie π_θ, ale aj stochasticitu samotného prostredia. Táto sa v učení s odmenou opisuje prechodovou funkciou $\mathcal{T}(s_t, a_t, s_{t+1})$, ktorá určuje pravdepodobnosť prechodu zo stavu s_t do stavu s_{t+1} pri vykonaní akcie a_t.

V teórii učenia s odmenou existuje veľké množstvo prístupov k tomu, ako sa snažiť dlhodobé odmeny maximalizovať. Vo všeobecnosti sa však dajú jednotlivé prístupy rozdeliť do nasledujúcich troch skupín:

- metódy na báze hodnotových funkcií;
- metódy na báze stratégií;
- aktor-kritik metódy.

C.2.1 Učenie na báze hodnotových funkcií

Odmeňovacia funkcia poskytuje agentovi len krátkodobú spätnú väzbu. Maximalizovať lačným spôsobom odmenu v najbližšom časovom kroku nebude vo väčšine úloh viesť ku maximalizácii dlhodobých odmien. Základnou myšlienkou metód založených na hodnotových funkciách je naučiť sa z interakcie s prostredím hodnotovú funkciu, t.j. funkciu, ktorá by vypovedala už priamo o dlhodobých odmenách. Lačným výberom akcie, ktorá v danom stave maximalizuje hodnotovú funkciu sa potom už dá dosiahnuť aj maximalizácia celkových odmien.

Hodnotové funkcie sú dvoch druhov:

- Hodnotová funkcia stavov $V^\pi(s_t)$ určuje, aké dlhodobé odmeny možno očakávať, ak sa agent nachádza v stave s_t a vieme, že sa riadi stratégiou π. Formálna definícia môže vyzerať takto [59]:

$$V^\pi(s) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s \right\}. \tag{C.4}$$

- Hodnotová funkcia akcií $Q^\pi(s_t, a_t)$ zase určuje, aké dlhodobé odmeny môže agent očakávať za vykonanie akcie a_t v stave s_t – znovu za predpokladu, že sa riadi stratégiou π:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \,\middle|\, s_t = s, a_t = a \right\}. \tag{C.5}$$

Existuje viacero metód, ako sa hodnotovú funkciu naučiť. Obľúbená je napríklad metóda známa ako Q učenie, ktorá hodnotovú funkciu akcií reprezentuje tabuľkou a na začiatku ju nejako inicializuje – napríklad všetky hodnoty nastaví na nuly. Potom na základe skúseností túto pôvodnú tabuľku postupne aktualizuje pomocou nasledujúceho pravidla (ktoré vychádza z Bellmanovej rovnice):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{C.6}$$

Hlboká Q sieť

Problém Q učenia v klasickej formulácii uvedenej vyššie je tabuľková reprezentácia hodnotovej funkcie. Dá sa použiť len keď je priestor stavov aj akcií diskrétny a veľmi malý – jednak preto, že nekonečný počet hodnôt sa nedá uložiť do tabuľky, ale aj preto, že tabuľková reprezentácia vôbec nezovšeobecňuje.

Tabuľkovú hodnotovú funkciu je preto v praxi snaha nahradiť určitou aproximáciou. Osobitne silné sú metódy, ktoré vedia na aproximáciu použiť hlboké neurónové siete – keďže tie sa vedia samy naučiť, ako si predspracovať vstupy.

Problém s použitím neurónových sietí v učení s odmenou je však v tom, že neurónové siete nie sú dobré v inkrementálnom učení. Zmena hodnoty jedného stavu môže mať (často nepredvídateľný) dopad aj na hodnoty ďalších stavov. Stavy, ktoré za sebou nasledujú v čase, sú navyše väčšinou navzájom silno korelované. Ak sa s nimi sieť postupne trénuje, dochádza väčšinou ku katastrofickému zabúdaniu – sieť si síce zapamätá nové vzorky, ale zabudne všetko, čo sa naučila predtým.

Metóda známa ako hlboká Q sieť (angl. deep Q-network; DQN) [77, 78] tento problém rieši tak, že všetko, čo agent zažije sa ukladá do špeciálnej pamäte. Z nej sa potom v každom kroku náhodne vyberú niektoré zapamätané vzorky a hlboká neurónová sieť sa trénuje na nich. Vzorky sú vybrané náhodne, tým pádom nie je medzi nimi taká silná korelácia. Navyše sa stále trénuje aj so staršími vzorkami. Problém katastrofického zabúdania preto odpadá.

Na to, aby metóda DQN fungovala, je potrebné použiť ešte niekoľko ďalších trikov, ale na tomto mieste už nie je priestor sa vecou podrobnejšie zaoberať – informácie je možné nájsť v texte práce.

Metóda DQN je známa pomerne nízkou stabilitou a sú tiež známe prípady, kedy nekonverguje ku správnemu riešeniu. Napriek tomu sú jej výsledky v praxi typicky celkom dobré. Jej hlavnou výhodou je však pomerne vysoká vzorková efektívnosť vyplývajúca z toho, že ide o metódu neviazanú na stratégiu (angl. off-policy). Dokáže sa učiť aj zo vzoriek, ktoré vyprodukovala iná než aktuálna stratégia. Preto si môže staré vzorky uložiť do pamäte a opätovne ich použiť, hoci sa medzi časom stratégia učením zmenila.

C.2.2 Učenie na báze stratégií a aktor-kritik metódy

Popri metódach založených na hodnotových funkciách existuje aj skupina metód, ktoré sa hodnotovými funkciami nezaoberajú, ale namiesto toho explicitne reprezentujú stratégiu, ktorú potom priamo optimalizujú. Najefektívnejšie metódy tohto druhu sú založené na gradientnom učení a väčšinou vychádzajú z teorémy o gradiente stratégie, ktorá hovorí [60, 61]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi}\left[Q^\pi(s,a)\nabla_\theta \ln \pi(s,a)\right], \tag{C.7}$$

kde ρ^π je (nenormalizované) rozdelenie stavov so znehodnotením [61]:

$$\rho^\pi(s') = \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathcal{I}(s) p(s \to s', t, \pi) \mathrm{d}s, \tag{C.8}$$

Ako vidno, postupom v smere gradientu (C.7) sa robí vykonanie akcie a v stave s o to pravdepodobnejšie, čím má táto akcia v danom stave vyššiu hodnotu. Hodnota $Q^\pi(s, a)$ prirodzene nie je známa, preto sa musí nejakým spôsobom odhadnúť. Najjednoduchším spôsobom, ako to urobiť, je otestovať niekoľko epizód, zmerať skutočné celkové odmeny a odhad vytvoriť na základe toho. Metóda používajúca tento prístup sa nazýva REINFORCE.

Problém tohto jednoduchého prístupu je, že odhad má typicky obrovský rozptyl, čo učenie veľmi sťažuje. Existuje viacero spôsobov, ako sa dá rozptyl znížiť. Zrejme najsofistikovanejší spôsob je učiť sa okrem stratégie celý čas aj hodnotovú funkciu $Q^\pi(s, a)$ – ako sme to robili pri metódach založených na hodnotovej funkcii. Metódy tohto typu sa označujú ako aktor-kritik metódy.

Výhodou metód založených na stratégii a aktor-kritik metód je jednak to, že sa dajú aplikovať aj v prípade spojitých akcií (hodnotové metódy sú založené na tom, že sa porovná hodnota všetkých akcií a vyberie sa z nich tá najlepšia, čo v prípade spojitých akcií nie je možné urobiť) a jednak to, že tie najlepšie z nich zvyknú byť podstatne stabilnejšie než metóda DQN.

Podstatnou nevýhodou je naopak to, že sú spravidla viazané na stratégiu (angl. on-policy) a majú v dôsledku toho omnoho nižšiu vzorkovú efektívnosť. Výnimku v tomto smere však tvorí nedávno navrhnutá metóda *soft actor-critic* (SAC) [89], ktorá je neviazaná na stratégiu a má dobrú stabilitu aj vzorkovú efektívnosť. V čase, keď sme realizovali prípadové štúdie, ktoré sú súčasťou práce, nebola ešte táto metóda zverejnená – navyše nie je ani doteraz dobre otestovaná pre použitie s diskrétnymi akciami. Preto sme sa priklonili k použitiu metódy DQN. Určite by však stálo za to v oboch aplikáciách v budúcnosti otestovať aj SAC – je možné, že by to viedlo ku zlepšeniu výsledkov.

C.3 | Abstrakcia a vizuálna pozornosť v učení s odmenou

Odkedy sa v rámci učenia s odmenou začali aplikovať hlboké modely, dosahuje vynikajúce výsledky na mnohých úlohách. Súvisí to s tým, že nie je pre každú novú úlohu potrebné ručne realizovať zložité a dlhotrvajúce príznakové inžinierstvo. Hlboká neurónová sieť sa dokáže naučiť extrahovať užitočné príznaky sama.

Paradoxne však tento istý dôvod súvisí s nízkou vzorkovou efektívnosťou hlbokého učenia s odmenou, ktorou trpia aj metódy ako DQN, ktoré majú ešte na pomery hlbokého

učenia s odmenou nízku vzorkovú zložitosť. Problém je v tom, že agent sa musí učiť nielen realizovať cieľovú úlohu, ale sa musí predtým ešte naučiť ako spracovať obraz, zvuk, či iný typ vstupu, ktorý dostáva. Je pochopiteľné, že je na to potrebné podstatne väčšie množstvo dát než ako keby sa systém učil z už preddefinovaných príznakov.

V súčasnosti sa aktívne pracuje na viacerých prístupoch, ktoré pomáhajú vzorkovú efektívnosť hlbokého učenia s odmenou zlepšiť – znovu sa napríklad v tomto kontexte autori venujú konceptom ako je intrinsická motivácia, zvedavosť a pod., ktoré už boli známe v klasickom učení s odmenou. Zaujímavá je z tohto pohľadu aj oblasť abstrakcie a toho, ako ju dokáže agent pri učení využívať.

Zdroj [5], ktorý sa venuje abstrakcii v kontexte klasického učenia s odmenou, rozlišuje medzi tromi typmi abstrakcie:

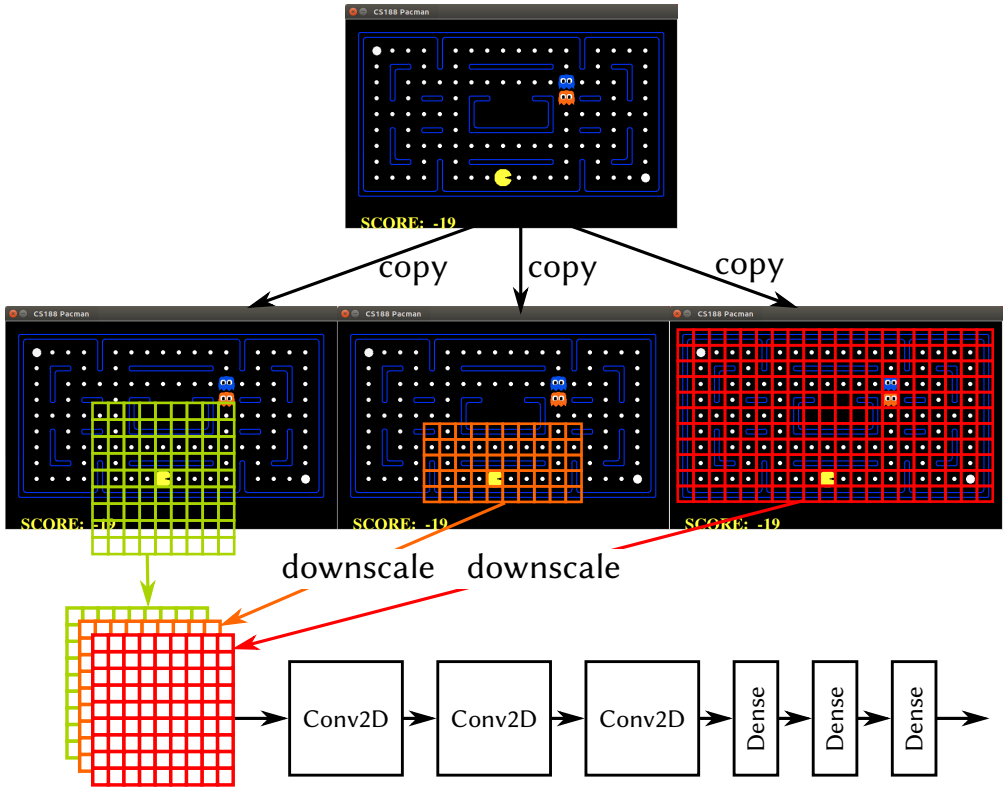
- *Aspektualizácia:* abstrahuje od určitých aspektov – napr. neberie od úvahy farbu objektu alebo jeho tvar. Aspektualizácia znižuje rozmer vstupných pozorovaní.
- *Zhrubozrňovanie:* znižuje rozlíšenie, zmenšuje rozmery priestoru možných pozorovaní.
- *Konceptuálna abstrakcia:* využíva existujúce príznaky na vytvorenie nových abstraktných konceptov (napr. skombinuje oči a ústa a vytvorí tvár). Tento typ abstrakcie je najvšeobecnejší a druhé dva typy je možné chápať ako jeho špeciálne prípady.

Formálne definície zodpovedajúce jednotlivým typom je možné nájsť v práci.

Hlboké učenie s odmenou má schopnosť vykonávať konceptuálnu abstrakciu. Súvisí to práve so schopnosťou hlbokých modelov naučiť sa automaticky realizovať predspracovanie a extrahovať príznaky: počínajúc jednoduchými a pokračujúc ku čoraz abstraktnejším. Úzko to tiež súvisí so schopnosťou globálne zovšeobecňovať.

Keďže konceptuálna abstrakcia principiálne v sebe zahŕňa aj ostatné dva typy, mohli by sme sa domnievať, že v rámci hlbokého učenia s odmenou už abstrakciu nie je potrebné riešiť a postará sa o ňu samotný framework. To je čiastočne aj pravda, ibaže ak do architektúry hlbokej siete nezakódujeme vhodné kognitívne preferencie, bude sa musieť o užitočnosti všetkých troch typov abstrakcie dozvedieť priamo z dát – čo si môže vyžadovať nerealisticky vysoký počet vzoriek a nesmierne veľký počet krokov učenia.

Ide približne o rovnaký prípad, ako keby sme na spracovanie obrazu nepoužili konvolučné vrstvy, ale klasické. Neurónová sieť by sa principiálne mala byť schopná korektné správanie naučiť napriek tomu – ibaže ak má potom rozpoznať určitý typ objektu na ľubovoľnom mieste v obraze, bude sa musieť učiť z dát, v ktorých sa daný objekt na každom mieste naozaj nachádzal – pretože sme do nej prostredníctvom architektúry nezakódovali informáciu o tom, že sa má správať invariantne na pozíciu.



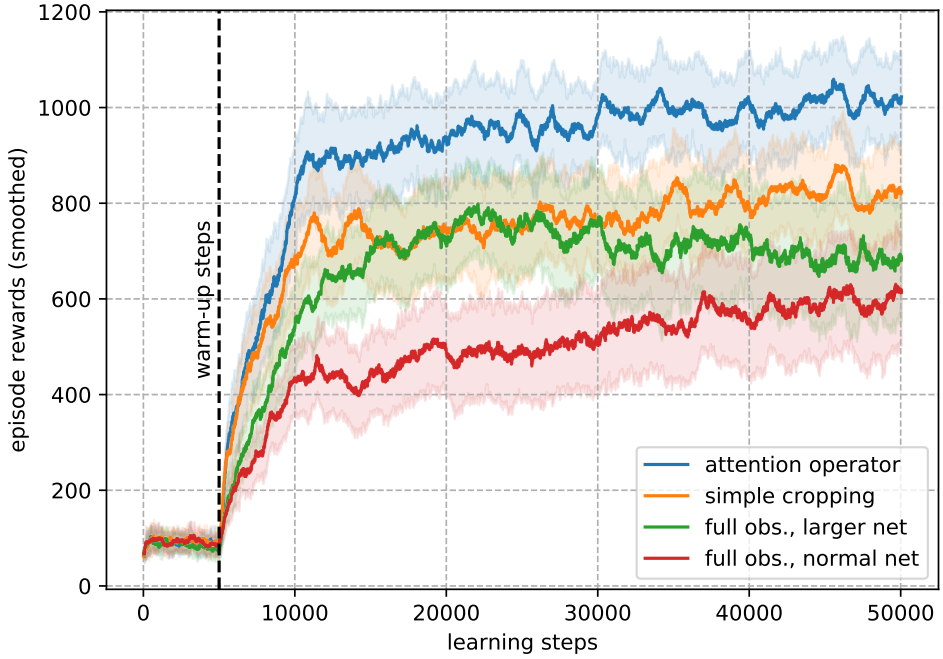
Obr. C.4: Navrhnutý operátor vizuálnej pozornosti: okno s plným rozlíšením je centrované okolo agenta. Ostatné okná sú centrované len slabo – tak, aby vždy ležali vo vnútri a nemuseli sa dopĺňať nulami.

Abstrakcia na báze vizuálnej pozornosti v hre Pac-Man

Ako sa dá teda sieť prispôsobiť tak, aby efektívne abstrahovala? V práci to overujeme na modelovom probléme, ktorým je známa hra Pac-Man. Na zavedenie agresívnejšej abstrakcie využívame koncept vizuálnej pozornosti, ktorý zaviedol v kontexte rozpoznávania obrazu článok [92] a ktorý sme my zodpovedajúcim spôsobom upravili. Vytvorili sme agenta, ktorý dokáže – napriek tomu, že vstup neurónovej siete má fixný rozmer – vnímať prostredie rozličných rozmerov. Realizuje to tak, že svojmu bezprostrednému okoliu venuje maximálnu pozornosť – vnem o ňom dostáva na vstupe v pôvodnom rozlíšení. V tomto kroku sa teda aplikuje len aspektualizácia, ktorá pôvodný vnem orezáva o niektoré rozmery.

Agent však vníma aj vzdialenejšie okolie – ibaže len hmlisto. Tu sa zase aplikuje zhrubozrňovanie: pôvodný vnem sa najprv podvzorkuje a agent vníma až jeho sumarizovanú podobu. Treba však zdôrazniť, že agent má prehľad o celom svojom okolí – hoci vzdialenejšie oblasti vníma len v menej ostrej podobe. Princíp graficky ilustruje Obr. C.4 a do podrobností je opísaný v texte práce.

Výsledky – na Obr. C.5 a Obr. C.6 – ukazujú, že tento prístup („attention operator“) funguje podstatne lepšie, než iné prístupy – či už použitie celého vstupného vnemu bez navrhnutého operátora (a teda bez preferencie abstrahovať; „full obs. larger net“, „full obs.,



Obr. C.5: Krivky učenia (odmeny získané počas 1 epizódy verzus čas).

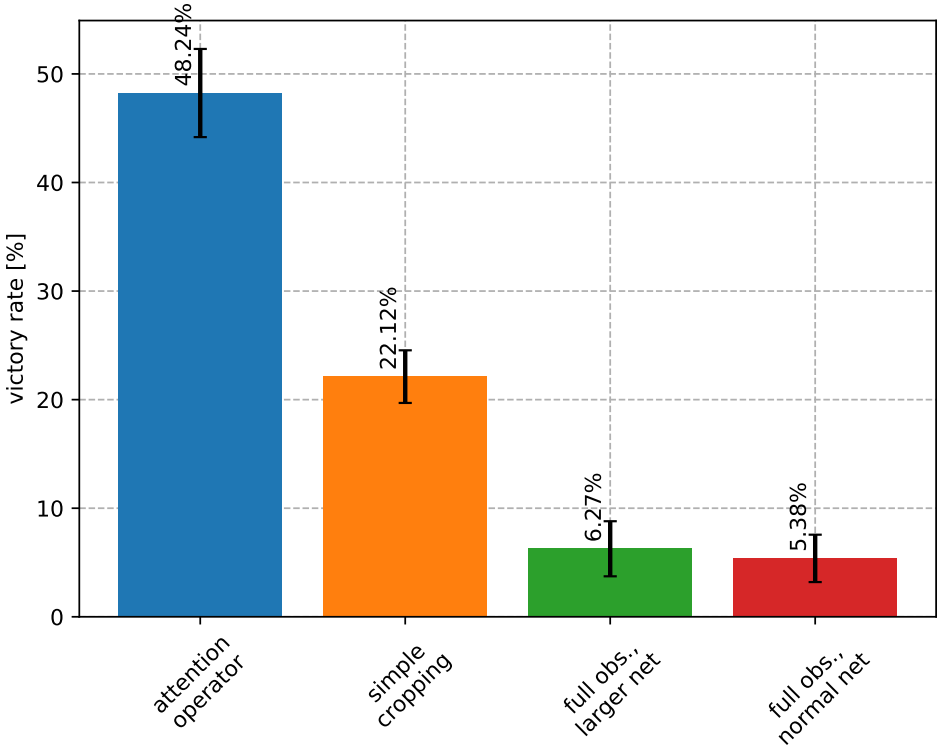
normal net“) alebo použitie výrezu z najbližšieho okolia v pôvodnom rozlíšení, ale bez menej ostrého prehľadu o zvyšku okolia („simple cropping“).

Operátor vizuálnej pozornosti v aktuálnej verzii používa na sumarizáciu obrazu maximálne združovanie (t.j. vyberá z pixelov maximálnu hodnotu). Keďže na vstupe agenta sú binárne príznakové mapy, dáva tento typ sumarizácie zmysel. V iných aplikáciách však môžu byť potrebné sofistikovanejšie metódy sumarizácie – preto v práci navrhujeme aj spôsob, ako sa sumarizáciu učiť. Metóda, ktorú sme vyvinuli zatiaľ nedosahuje dobré výsledky – uvádzame však niekoľko príčin, ktoré by to mohli spôsobovať a veríme, že sa problém podarí odstrániť pomocou opatrení, ktoré sme navrhli. To už však bude až predmetom ďalšieho výskumu.

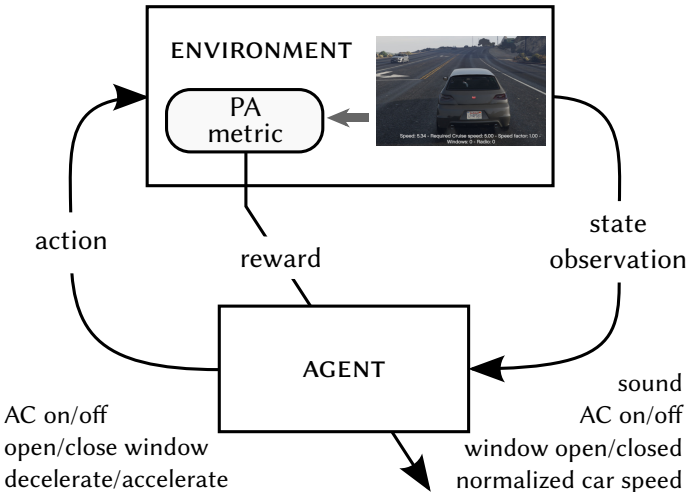
C.4 | Multisenzorické učenie s odmenou

Ďalšia kapitola práce sa venuje multisenzorickému učeniu. V praktických aplikáciách inteligentných agentov sa ako samozrejmosť vyžaduje schopnosť kombinovať a využívať dáta rôznych typov a z viacerých zdrojov – napríklad obrazové dáta, audio dáta, tabuľkové dáta, sekvenčné dáta a pod. Hlboké učenie s odmenou predstavuje dobrý framework, kde je možné viacero typov dát kombinovať – aj v tomto prípade je však potrebné návrhom architektúry dať sieti vhodné kognitívne preferencie.

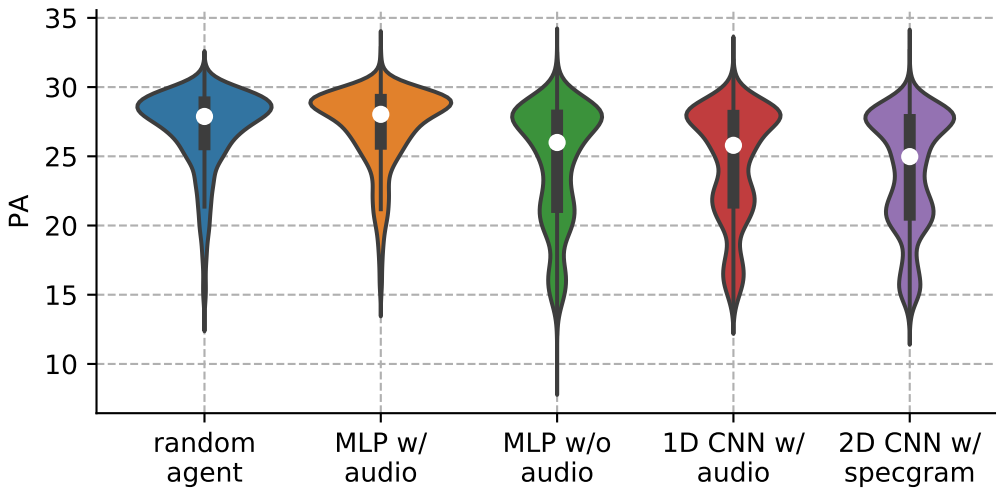
Prípad multisenzorického učenia s odmenou overujeme na štúdii uskutočniteľnosti z oblasti dopravnej psychoakustiky. Štúdia aplikuje hlboké učenie s odmenou a kladie si otázku, či je možné vhodnými akciami znížiť psychoakustické rušenie, ktoré pociťuje počas jazdy



Obr. C.6: Pomer výhier pre jednotlivých agentov.



Obr. C.7: Štruktúra úlohy učenia s odmenou v štúdii uskutočniteľnosti.



Obr. C.8: Profily psychoakustického rušenia po prvých 11 000 krokoch učenia. Vertikálna os označuje psychoakustické rušenie a šírka grafu vyjadruje početnosť vzoriek s danou hodnotou rušenia – podobne ako to je v prípade histogramu.

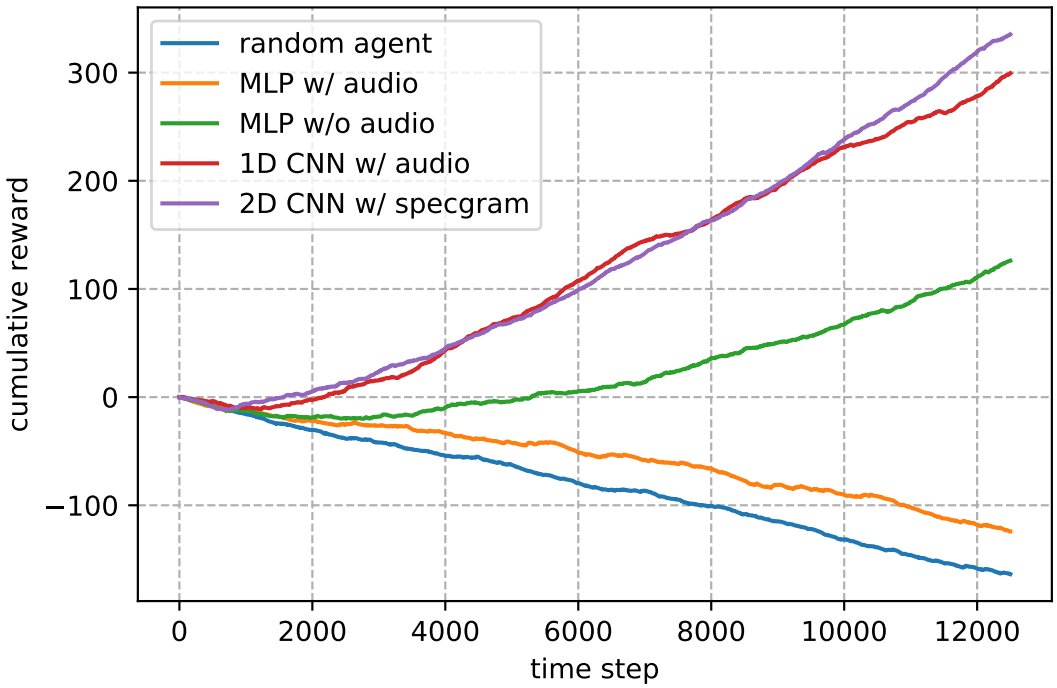
vodič vozidla. Hlavné experimenty boli vykonané v simulácii, pričom miera psychoakustického rušenia sa vyhodnocovala pomocou existujúcej metriky dostupnej v literatúre. Štruktúru úlohy zobrazuje Obr. C.7.

Agent dostáva ako vstup tabuľkové dáta o stave a rýchlosti vozidla, ktoré má kombinovať s audio signálom. Odmeny sú priamo založené na metrike psychoakustického rušenia – sú len lineárne preškálované. Testovali sme niekoľko architektúr. Pre porovnanie sme zaradili náhodného agenta („random agent“), viacvrstvový perceptrón využívajúci len tabuľkové dáta („MLP w/o audio“) a viacvrstvový perceptrón, ktorý má na vstupe naivne zreťazené tabuľkové dáta a audio („MLP w/ audio“).

Ďalej sme testovali dve architektúry: Jedna vstupné audio predspracováva pomocou 1D konvolučných vrstiev (1D CNN w/ audio). Druhá z audia najprv vypočíta spektrogram a naň potom aplikuje 2D konvolučné vrstvy (2D CNN w/ spectrogram). Výsledky vidno na Obr. C.8 a Obr. C.9.

Výsledky indikujú, že voľbou vhodných akcií je skutočne možné redukovať psychoakustické rušenie. Osobitnou otázkou prirodzene je, ako také akcie inkorporovať pri riadení skutočného vozidla – bezpochyby by na to bolo potrebné navrhnúť vhodné rozhranie, aby akcie v skutočnosti samy nespôsobili ešte väčšie rušenie, než aké eliminujú.

Z pohľadu využitia multisenzorických dát možno konštatovať, že architektúra naivne zreťazujúca tabuľkové a audio dáta má ešte horšie výsledky než architektúra, ktorá audio dáta nevyužíva vôbec. Naopak architektúry, ktoré audio dáta najprv vhodne predspracujú a s tabuľkovými dátami ich zreťazia až potom, ako sa dostatočne znížil ich rozmer, dosahujú podstatne lepšie výsledky. Z toho vidno, že voľba vhodnej architektúry je naozaj kľúčová aj v prípade multisenzorického učenia s odmenou.



Obr. C.9: Kumulatívne odmeny získané počas učenia jednotlivými agentmi.

V ďalšej časti štúdie sa venujeme dátam zozbieraným z reálneho vozidla a ukazujeme, že v čiastočnom rozsahu podporujú závery simulačných experimentov. Informácie o tej časti štúdie si je možné preštudovať priamo v práci – v resumé ich nebudeme znovu opakovať kvôli rozsahu.

C.5 | Záver

V práci sme videli, že návrh inteligentných agentov je mimoriadne komplexný a multidisciplinárny problém. Zdá sa však, že framework hlbokého učenia s odmenou je – vďaka vlastnostiam hlbokých neurónových sietí, na ktorých je založený – dostatočne flexibilný na to, aby mnoho z potrebných komponentov dokázal doteraz bezprecedentným spôsobom spojiť.

Aplikácie hlbokých neurónových sietí sú veľmi široké a ich schopnosť automaticky objaviť vhodné príznaky priniesla veľký pokrok v oblasti strojového učenia. Ako sme však videli, aj tieto sofistikované modely fungujú správne len vtedy, keď ich prostredníctvom návrhu architektúry vybavíme vhodnými kognitívnymi preferenciami, ktoré odrážajú predošlé znalosti o tom, aká je štruktúra úlohy a ako treba ku jej riešeniu pristupovať. Práve to im pomáha správne zovšeobecňovať.

Tento fakt je evidentný vo všetkých oblastiach, kde hlboké učenie dosiahlo úspech. V aplikáciách počítačového videnia napríklad konvolučné architektúry inkorporujú poznatky o tom, že obraz sa skladá z lokálnych vzorov, z ktorých sa hierarchicky skladajú zložitejšie a

zložitejšie príznaky. Pri spracovaní sekvenčných dát, kde je potrebná pamäť, dosahujú zase dobré výsledky špeciálne architektúry ako sú dlhá krátkodobá pamäť (angl. long short-term memory; LSTM) alebo hradlovaná rekurentná jednotka (angl. gated recurrent unit; GRU). V poslednom čase sa navyše tieto architektúry v niektorých aplikáciách úspešne nahrádzajú mechanizmami sekvenčnej pozornosti.

Ukázali sme, že to iste platí v prípade učenia s odmenou a že koncept vizuálnej pozornosti môže agentovi pomôcť agresívnejšie aplikovať určité formy abstrakcie, čo mu opäť pomôže lepšie zovšeobecňovať. Ďalej sme ukázali, že môže tento prístup do určitej miery zvýšiť aj vzorkovú efektívnosť – a vysoká vzorková zložitosť je v súčasnosti jednou z hlavných bariér voči širšej aplikácii systémov hlbokého učenia s odmenou v praxi. Navrhnutý operátor vizuálnej pozornosti bude tiež pravdepodobne schopný pomôcť pri transfere agenta na iné varianty tej istej úlohu – tento predpoklad však bude ešte treba verifikovať.

Aby sa dal tento prístup ľahšie aplikovať aj na iné problémy, ktoré si môžu vyžadovať sofistikovanejšie formy vizuálnej sumarizácie než je tá, ktorú sme použili v experimentoch, pokúsili sme sa navyše navrhnúť operátor, ktorý by sa dokázal vizuálnu sumarizáciu sám naučiť. Tento prístup však, vo svojej aktuálnej podobe, nedosahuje dobré výsledky. Identifikovali sme však niekoľko možných príčin, prečo daný prístup zlyháva a je možné, že v budúcnosti sa súvisiace problémy podarí vyriešiť.

Ďalším bodom nášho záujmu je fakt, že v realistických aplikáciách musia byť inteligentní agenti schopní využiť bohaté, multisenzorické vstupy. Schopnosť vykonať efektívnu fúziu dát viacerých typov z rôznych zdrojov je absolútnou nevyhnutnosťou. V práci sme prezentovali štúdiu uskutočniteľnosti, ktorá ilustruje, že hlboké učenie s odmenou je takú fúziu schopné vykonať, ale že vhodný návrh architektúry má znovu kľúčový význam.

Štúdia sa venuje experimentom, kde tabuľkové dáta a akustický signál vstupujú do agenta, ktorého cieľom je voľbou vhodných akcií znížiť psychoakustické rušenie ľudského vodiča vo vozidle. Experimenty boli vykonané v bohatom simulačnom prostredí, kde sa psychoakustické rušenie vyhodnocovalo pomocou už existujúcej metriky známej z literatúry. Ako sme ukázali, je skutočne možné – prinajmenšom v princípe – zvoliť akcie, ktoré metriku znížia. Výsledky sú už v určitej šírke podporené aj reálnymi dátami, ktoré sme zozbierali.

Ten istý experiment sme v tejto práci využili aj na to, aby sme ukázali, že nevhodne navrhnuté architektúry sú nielenže neschopné benefitovať z informácie obsiahnutej v oboch dátových tokoch, ale že zahrnutie akustických dát môže dokonca zhoršiť výsledky, ak sú oba signály len naivne zreťazené a vložené na vstup siete.

Celkovo teda aktuálna práca ukazuje, že návrh architektúry je silným spôsobom, ako možno do učiaceho sa systému inkorporovať predchádzajúce poznatky. Výsledky tiež podporujú myšlienku, že hlboké učenie s odmenou predstavuje nádejný a flexibilný framework

pre návrh inteligentných agentov.

Veľa problémov prirodzene zostáva otvorených. Mnoho nevyhnutných mét – ako je napríklad schopnosť automaticky objavovať koncepty, vykonávať efektívne nekontrolované učenie alebo ukotvenie jazyka a následná možnosť poskytovať agentom povely a spätnú väzbu v prirodzenej reči – sa nám zatiaľ nedarí dosiahnuť. Napriek tomu existujú dobré dôvody na opatrný optimizmus. Stále síce nepoznáme všetky odpovede, ale je možné, že v sa nám v budúcnosti aj tak podarí nájsť zlatý grál umelej inteligencie: schopnosť navrhovať inteligentných agentov.