
Master's Thesis

# Reinforcement Learning for Navigating Particle Swarms by Global Force

Matthias Konitzny

May 27, 2020

**Institute of Operating Systems and Computer Networks**
**Algorithms Group**
**Prof. Dr. Sándor Fekete**

Supervisor:
Prof. Dr. Sándor Fekete
Dominik Krupke

**Statement of Originality**

This thesis has been performed independently with the support of my supervisor/s. To the best of the author's knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text.

Braunschweig, May 27, 2020

_____

**Abstract**

Since the early years of computer science navigation tasks were heavily explored. Since most of these tasks are NP-hard, even modern computers struggle with the optimal solution of larger instances for those problems. To overcome this problem various approximation algorithms have been developed to get usable results in a relatively short time. In this work we focus on a specific problem: The navigation of particles by a single global force - also known as tilt problem. We exploit recent developments in the area of reinforcement learning to navigate all particles to a given goal position and compare the results to state-of-the-art approximation algorithms.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# 1 Introduction

In this chapter we want to give some introduction into the main focus of this thesis. We start by giving some motivation and historical context on reinforcement learning in Section 1.1. We further give an overview over related work in Section 1.2, while giving a short summary of our own results in Section 1.3.

## 1.1 Motivation

**History on Machine Learning.** The idea of intelligent machines which are able to learn more or less autonomously, which are able to solve any complex task by themselves and which thrive for human capabilities or even exceed them was theorized and discussed long before computers were able to learn anything at all. For a long time fiction was way ahead of reality, with films and novels drawing a future of human-like robots - be it positive or negative - while in reality machine learning still struggled at the easiest tasks and was computationally way too heavy to ever run on any machine in a real-time scenario. While we all dreamed of our own Wall-E or feared the rise of Skynet the birth of modern computing not only inspired authors, but also inspired researchers to explore self-learning computers.

This wish to develop intelligent machines found its first success in 1943, when McCulloch and Pitts described how neural networks can be used to build mathematical models for logical or arithmetic expressions [52]. With the birth of artificial neural networks, which were introduced in 1959 by Frank Rosenblatt in the form of perceptrons [66] a new field of computer science was born.

Even though we had the building blocks to artificial neural networks, there was still a long way before they became as powerful as they are today. Researchers were alternating between breakthroughs and decades of standstill. Even 20 years ago most scientists agreed that artificial neural networks will never become as powerful as other machine learning technologies with stronger mathematical foundations like support vector machines. Today, after the introduction of non-linearity to perceptrons, the breakthrough of the backpropagation algorithm and countless other advances, state-of-the-art models are capable of surpassing humans in a lot of complex tasks like handwriting recognition and are used to solve all kinds of problems where traditional algorithms fail to deliver results. Artificial neural networks developed beyond a point where their capabilities outclass most of the other competing techniques for machine learning and with the ever rising computational power their capabilities only grow further.

**Learning by Doing: Reinforcement Learning** In this work we will be using techniques from a specific sector of machine learning called reinforcement learning or more specifically its modern combination with artificial neural networks called Deep Reinforcement Learning (also *DRL* or Deep RL).

Deep RL is one of the most exciting fields of machine learning today, because it is capable of achieving actual superhuman performance without any human ever creating any training data for it. In reinforcement learning the data is created solely by an agent interacting with its environment. Reinforcement learning can therefore be compared to how humans learn themselves. Therefore all Deep RL algorithms are unsupervised machine learning algorithms.

Like traditional machine learning, reinforcement learning has been around for decades, but just recently gained notable attention. In 2013 researchers from the British startup DeepMind were able to train a system to play any game from the game console Atari without prior knowledge and only with raw pixel data as input. [55] They later even improved their system and were able to outperform trained human players [56].

But these two successes were only the beginning of a series of advances for deep reinforcement learning. After Google acquired DeepMind their new system *AlphaGo* was able to defeat one of the worlds top class Go players Lee Sedol with 4:1 [16] and even the world champion Ke jie in 2017. The system was then expanded and generalized to also play shogi and chess and renamed to *AlphaGo Zero*. AlphaGo Zero not only defeated its predecessor, but also defeated state-of-the-art alpha-beta search engines for chess like Stockfish [75]. In 2019 they reached a new milestone, when their system *AlphaStar* was able to beat a professional player at StarCraft II - a complex multiplayer strategy game [9].

We will take an in-depth look of current deep reinforcement learning techniques and how they work in Chapter 4.

**Everything can be a Game.** We saw reinforcement learning had great success for games, but what happens if we want to solve more abstract problems like wayfinding? Can we still apply these algorithms? The short answer is yes: If we look at it, most algorithmic problems can be easily transformed into a game. We always have something we can express as a state - be it as an actual image or just some numbers - and a well-defined objective for the "player", like finding the shortest path between two points. The player generates the solution for our problem step by step by playing a single round of the "game".

In this work we want to tackle a specific problem: Navigating a swarm of particles to a goal position in a maze-like environment, by applying a global uniform force. This problem finds its application in medicine in form of targeted drug delivery. The goal is to localize medical treatment to efficiently combat cancer, localized infections or internal bleeding, without causing unwanted and potentially harmful side effects for the rest of the body. The delivery requires navigation of the distributed microscopic particles through pathways of blood vessels to a target location. As the particles are too small to build microrobots with sufficient energy to swim against flowing blood, a global external force

like an electromagnetic field is used for motion control. This means all particles are subjected into the same direction unless their path is blocked by obstacles. Since at the start all the particles are in different locations, navigating all particles by a uniform force to a single destination is not an easy task.

In this work, we want to explore the capability of modern reinforcement learning algorithms on solving this problem. As previous work [12] showed that the problem is NP-hard, finding optimal solutions will not be the goal. Instead we will compare the results to state-of-the-art approximation algorithms. Becker et al. also showed, that reinforcement learning is capable of solving this problem for small instances in relatively short time. With this prior knowledge we want to speed up the learning process, aim for larger instances and investigate the possibilities of transfer learning for targeted drug delivery. Our results may also be useful for the application of reinforcement learning on other algorithmic problems.

## 1.2 Related Work

Some related work. Many nice sources.
Two parts: RL and targeted drug delivery
RL: Large Scale Study, PhD Thesis/Paper TDD: Papar, Details on tdd

## 1.3 Our Results

Hopefully we have some by the time.

# 2 Guiding Particle Swarms with Global Force

In this chapter we want to give an introduction into the problem which we will later try to solve using reinforcement learning. Controlling large particle swarms with a global force is a challenging navigation problem. Particles can either be partially autonomous robots, which move in unison into a direction given by an external signal (e.g. a light source), or the particles can be without any autonomy where we have a global force like magnetism which controls their movement. Because particles can be seen as marbles on a surface which are controlled by tilting, the problem is often referred to as tilt problem. Depending on the task there are a number of goals which can be solved in this setting, e.g. the well-known tilt assembly problem, where the goal is to build objects using particles controlled by a global force [11]. In this work we will focus on another problem, which finds its application in medicine and aims at navigating particles to a specific goal position inside of the human body. We will give a more detailed introduction in Section 2.1. We will then continue with a theoretical point of view on the problem in Section 2.2 and present a number of existing algorithmic approaches to solve the problem in Section 2.3. We will finally present previous work on solving the problem with reinforcement learning in Section 2.4.

## 2.1 Motivation

The treatment of medical problems often requires the use of some sort of medication. These medications are given as oral ingestion or intravascular injection and then are further distributed throughout the body via blood circulation. Most if not all of these treatments can produce some potentially serious side-effects. Especially when treating serious medical problems like internal bleedings, localized infections, tumors or cancer (see Figure 2.1), a large portion of the medication does not reach its target destination and the possible worst-case side-effects can be as serious as the original medical problem. The idea of *targeted drug delivery*, deals with the problem to concentrate some given medicine in a certain part of the body to reduce unwanted side-effects and at the same time increase the efficacy.

Over the years many approaches have been developed to perform targeted drug delivery. In recent work, the idea of using magnetic nanoparticles has been explored and proven to be promising, since they combine high drug loading with great targeting abilities [81, 5]. The particles are controlled by an external uniform magnetic force and guided towards a target location. The magnetic fields can be generated by using the coils in an MRI scanner

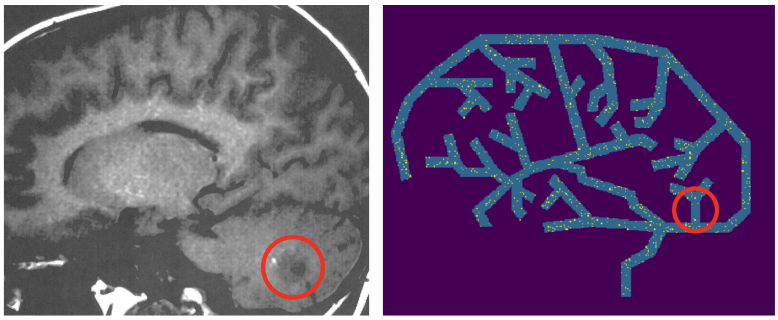


Figure 2.1: Targeted drug delivery for tumor treatment (from [12]). An MRI image on the right shows a tumor (marked by the red circle) and an abstracted version (left) shows particles containing some sort of medicine (yellow dots) which should be delivered to the target region.

[51, 63] or by adding additional magnetic coils [53]. Real-time MRI scanning also allows feedback control to some extend. At their target location, the particles are activated to release their payload by an external or internal stimulus such as temperature or pH.

Guiding particles by a uniform external (*global*) force requires solving a complex distributed navigation problem. The nanoparticles will be distributed randomly in a complex maze-like environment given by the vascular system of blood vessels. Using a uniform force to guide all particles regardless of their initial position to a single destination will be complicated due to many particles being blocked at some point by obstacles. This problem has been studied in the past in terms of bringing two particles to the same location called *rendevous search*. In 2001 Anderson and Fekete introduced methods for two-dimensional rendevous search [8] and Alpern and Gal introduced a number of new settings along with solution strategies in their 2006 paper [6]. However in this chapter we want to focus on a more recent work by Becker et al. [12] from 2020. They prove that the underlying problem is NP-hard and also present some novel approximation algorithms which greatly improve previous worst case guarantees.

## 2.2 Preliminaries

Using particles with individual capabilities (e.g. microrobots) is difficult since the tiny volume of these robots makes it nearly impossible to store enough energy to navigate in presence of flowing blood. The particles we are dealing with therefore have no computational power, memory or any other autonomy. Instead our particles are just some kind of substrate (e.g. iron-oxide) injected into the blood vessels and solely controlled by a uniform global force which affects all particles equally. In our model we will work with a two-dimensional abstraction of the environment and the particles: The blood-vessels will be modeled by a planar two-dimensional integer workspace $P$ which consists of orthogonal cells (*pixels*) and can therefore be seen as *polyomino*. All pixels not belonging to the

polyomino are blocked and cannot be entered by particles. The distance $dist(p, q)$ between two pixels $p$ and $q$ is given by the shortest path on the integer grid between $p$ and $q$ that stays within $P$. The diameter $D$ of the polyomino is then given by the maximum distance between any two of its pixels.

At the beginning, each pixel of $P$ may contain one (or more) particles which we will call the *configuration* of $P$, with the set of all possible configurations $\mathcal{P}$. Since we are dealing with edge-to-edge connected cells, four basic moves are possible to move each particle by one cell in one of the directions "Up" ($u$), "Down" ($d$), "Left" ($l$) or "Right" ($r$). If the particle cannot move to the next cell because it is blocked, it simply remains in its current cell. Since the size of the particles is insignificant compared to the size of the cells, if two particles $a$ and $b$ enter the same cell $p$, they will merge into a single particle. This can happen when $a$ enters $p$, while $b$ remains in $p$ due to being blocked. Merged particles will never be separated again and move in unison for subsequent moves.

We call a series of commands a *motion plan* $C = \langle c_1, c_2, c_3, \dots \rangle$ where each command $c_i \in \{u, d, l, r\}$. We further call a command sequence *gathering* if it merges all particles of a given configuration $A \in \mathcal{P}$ into a single pixel, such that $|A'| = 1$.

## 2.3 Algorithmic Approaches

In this Section we want to present a number of algorithmic approaches to solve targeted drug delivery. To keep things short, we will focus on recent algorithms which provide a reasonably good performance bound and have proven to perform at least as good as previous approaches. All algorithmic approaches use a gathering strategy, which first concentrates all particles into a single cell. The particles are then guided to the target area on a shortest path.

### 2.3.1 Static Shortest Path

In 2016 Mahadev et al. developed a simple greedy algorithm which is able to collect all $m$ particles with a command sequence of length $\mathcal{O}(m \cdot n^3)$, where $n$ is the polyomino's height times its width [49]. The algorithm was later named *Static Shortest Path* (SSP) in a paper by Becker et al. [12].

The algorithm works by iteratively merging two particles until all particles are merged into a single one. We included a sketch in Algorithm 2.3.1. We can see, that the algorithm uses the procedure `StaticShortestPath`, which subsequently moves a particle $a$ to the position of another particle $b$ until they are merged. Mahadev et al. showed, that when moving $a$ to the position ob $b$, the distance between the particles $a$ and $b$ only decreases if $b$ had a collision. Since the polyomino is bounded and $a$ and $b$ always move into the same direction, $b$ must have a collision after $\mathcal{O}(n)$ collision free iterations. Therefore $\mathcal{O}(n^2)$ commands are needed to reduce the distance by at least one, resulting in $\mathcal{O}(n^3)$ commands in total to merge two particles. The choice of particles that are merged in the next step

**Algorithm 2.3.1:** The static shortest path algorithm modified to merge an arbitrary number of particles.

```
Data: Configuration A, Bounded polyomino P
while |A| > 1 do
    Select two particles a and b from A.
    StaticShortestPath(a, b)
end
procedure StaticShortestPath (a : Particle, b: Particle):
    while dist(a, b) ≠ 0 do
        Compute C ∈ {u, d, l, r}^N as the shortest control sequence to move a onto
         pos(b).
        Execute C.
    end
```

can be random, but it has been shown, that choosing the pair with the maximal distance converges faster on average.

### 2.3.2 Dynamic Shortest Path

In 2020, Becker et al. presented a number of improved algorithms which provide stronger performance bounds [12]. For the special case of hole-free polyominos, they developed the *Dynamic Shortest Path* (DSP) algorithm which is able to compute a gathering sequence of $\mathcal{O}(D)$ with $D$ being the diameter of the polyomino. For non-simple polyominos, DSP still always merges two particles, but may need $\mathcal{O}(nD)$ steps, where $n$ is the number of pixels in the polyomino.

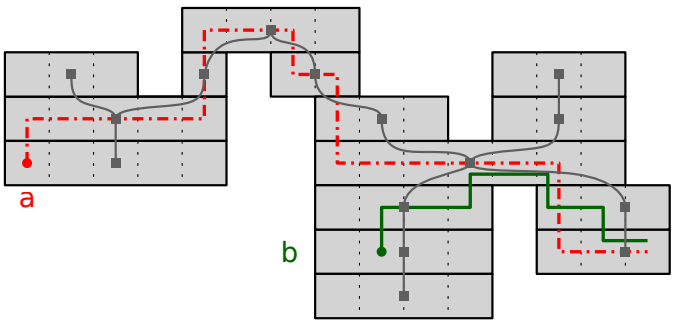



Figure 2.2: A simple polyomino $P$ and its edge-contact graph $\mathcal{C}(R)$. The particle $a$ (red) is moved towards particle $b$ (green) on a shortest path with respect to $\mathcal{C}(R)$. (from [12], edited)

The DSP algorithm starts by decomposing $P$ with horizontal lines through pixel edges which results in a set of rectangles $R$. The edge-contact graph $\mathcal{C}(R)$ is then constructed by creating a vertex for each rectangle and connecting two vertices if their respective rectangles share an edge. For a simple (hole-free) polyomino the edge-contact graph is a tree. Two

**Algorithm 2.3.2:** The dynamic shortest path algorithm modified to merge an arbitrary number of particles.

**Data:** Configuration $A$, Bounded polyomino $P$
**while** $|A| > 1$ **do**
  Select two particles $a$ and $b$ from $A$.
  `DynamicShortestPath`$(a, b)$
**end**
**procedure** *`DynamicShortestPath` (a : Particle, b: Particle)*:
  **while** $R_t^a \neq R_t^b$ **do**
    Compute a shortest path $S_t$ from $R_t^a$ to $R_t^b$ in $\mathcal{C}(R)$
    Move $a$ to $S_t(1)$ via a shortest path in $P$.
    Update $R_t^a$ and $R_t^b$
  **end**
  Compute a shortest path $C$ from $a$ to $b$.
  Merge $a$ and $b$ by executing $C$.

particles are then merged by using an improved shortest path strategy. For two particles $a$ and $b$, let $R_t^a$ and $R_t^b$ be the rectangles of $P$ containing the two particles in step $t$. Let $S_t$ be a shortest path from $R_t^a$ to $R_t^b$ and let $S_t(1)$ be the successor of $R_t^a$ on $S_t$ if it exists ($R_t^a \neq R_t^b$). The particles are then merged, by subsequently moving $a$ on a shortest path from $R_T^a$ to $S_t(1)$ and then recalculating $S_t$ (therefore the name *dynamic* shortest path). If there is no $S_t(1)$ the particles are merged, by moving $a$ towards $b$ by a shortest (horizontal) path, merging both particles within $R_t^a$. The whole process can be found in Algorithm 2.3.2 and an example showing the edge-contact graph and the process of merging two particles is shown in Figure 2.2.

### 2.3.3 Move to Extremum

The strategy *Move to Extremum* (MTE) was also developed by Becker et al. [12] and provides a strategy which - independent of the shape of the polyomino - provides a gathering sequence of length at most $\mathcal{O}(D^2)$ for two particles. The idea of MTE is to iteratively move an extreme particle (e.g. the top-rightmost) to an opposite extremum (e.g. bottom-leftmost) along a shortest path. Similar to the shortest path approach in SSP, the distance then decreases with every iteration.

Let $q$ be the top-rightmost pixel of $P$. To merge two particles, we select the bottom-leftmost particle $a$ and compute a sequence of moves, which moves $a$ to $q$ on a shortest path. We repeat this procedure of selecting the bottom-leftmost particle and moving it to $q$ until the particles have been merged. In each iteration, the sum of distances $\Delta$ of the two particles to $q$ decreases. This happens only if the other particle $b$ has a collision. Similarly to SSP, in every iteration the particle $b$ must have at least one collision (otherwise $q$ could not be the top-rightmost pixel or $a$ not the bottom-leftmost particle), so the sum decreases

at least by 1 every $D$ steps. Since in some polyominos (e.g. a square), the number of pixels $n$ is in $\Omega(D^2)$, we can argue, that the MTE produces sequences which are significantly shorter than $\mathcal{O}(n^3)$ from SSP.

**Algorithm 2.3.3:** The min sum to extremum algorithm to gather all particles. MTE is used as a subroutine and particles are merged into $k/4$ groups in convex corners at the beginning to reduce their total number.

```
Data: Configuration A, Bounded polyomino P
Determine which of the four corner types occurs at most k/4 times.
Apply a sequence of commands to merge all particles into at most k/4 groups.
while |A| > 1 do
    Calculate the sum of distances for all particles d_t for each extremum.
    For each command calculate the change in the sum of distances Δd_t^{c_i} that
     command would produce.
    Select the command j which decreases the sum the most.
    if Δd_t^{c_j} < 0 then
        Execute j
    else
        Select two particles a and b from A.
        MoveToExtremum(a, b)
    end
end
procedure MoveToExtremum (a : Particle, b: Particle):
    Select an extremum q which minimizes the sum to a and b.
    while dist(a, b) ≠ 0 do
        Select the particle which is further away from q.
        Move that particle to q on a shortest path.
    end
```

If we look at a swarm of particles, we can have at most $n$ individual particles in any configuration of a polyomino $P$, which would result in a total of $\mathcal{O}(nD^2)$ commands for a gathering sequence using MTE. Luckily, it is possible to further reduce the total number of particles by first moving them into convex corners. For every polyomino with $k$ convex corners, a diameter $D$ and a configuration $A \in \mathcal{P}$, this can be done with a command sequence of length $2D$. The resulting configuration $A' \in \mathcal{P}$ will have at most $k/4$ particles. This works by first selecting the type of convex corner from all convex corners (northwest (NW), northeast (NE), southwest (SW) and southeast (SE)) which occurs at most $k/4$ times. It is guaranteed by the pigeon hole principle that such a convex corner type must exist. Let us assume the NE corner is the type selected. After applying a sequence of $\langle r, u \rangle^D$ every particle lies in a NE corner. Therefore the total gathering sequence of MTE has a length of $\mathcal{O}(kD^2)$.

In their paper, Becker et al. also propose a variant of MTE called *Min Sum To Extremum* (MSTE) which generalizes the idea of MTE for a swarm of particles. Initially, MTE selects an extremum which has the smallest sum of distances to all particles. It then tries to find a command which reduces this sum the most. If it is unable to find such a command, it continues by merging two particles using regular MTE. We provide pseudocode for MSTE which also includes MTE as a subroutine in Algorithm 2.3.3. The particles are gathered into $k/4$ groups at the beginning.

## 2.4 The Reinforcement Learning Approach

In 2019 Li Huang showed that is is possible to solve the targeted drug delivery problem with reinforcement learning [36]. His approach was later tested against the algorithmic approaches and proved to significantly outperform both MTE and MSTE on individual instances [12]. In this section, we want to give an overview over how the agents for targeted drug delivery can be trained. The methods and terminology which are used for this approach will be explained in detail in Chapter 4.

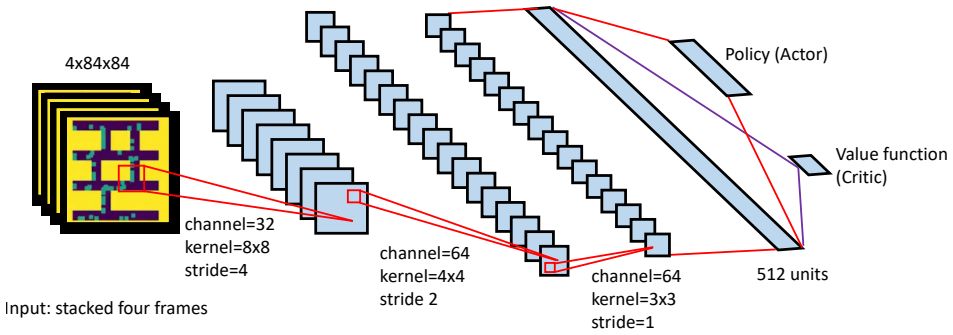



Figure 2.3: The network architecture for the reinforcement learning approach (from [36]).

**The Particle Maze Environment.** For reinforcement learning every problem has to be modeled as an environment where the agent can take actions and from where it receives observations and reward in return. For the problem of gathering particles at a goal location, the environment represents a single instance of the targeted drug delivery problem (a single polyomino) with a single target position. Particles are randomly spawned in empty cells of the polyomino at the beginning of an episode and the episode ends if all particles reach the goal position or after a certain time limit. Observations are generated as a visual representation of the environment as a grayscale image. The image includes the shape of the polyomino and the position of each particle. The observation will the be further preprocessed by downscaling the image to $84 \times 84$ pixels. Additionally the network receives a stack of the last four frames to add information about past particle positions. Images

are also normalized by their variance and zero centered with precomputed values from a random agent playing 10000 steps.

For reinforcement learning, the reward signal is a crucial component. Even a simple problem might become very hard to learn for an agent if the reward signal does not provide enough information about the goal or if it is somehow inconsistent or extremely sparse. Huang et al. generated reward based on a set of predefined goals. There are two main categories for these goals: The maximum distance of any particle to the goal position and the average distance of all particles to the goal position. In each category, the agent receives rewards based on reaching certain milestones - e.g. if the particle with the largest distance to the goal is less than 50 cells away from the goal position, the agent is granted a reward of $+2$. Rewards may increase for higher goal tiers and are only granted a single time for each episode. If the agent is able to bring all particles to the goal position it receives a large final reward of $+100$. The goals that were used during the experiments were sparse and limited to only 4-8 distinct rewards in each category. Rewards are also not normalized.

| Hyperparameter | Default Value |
|---|---|
| Extrinsic Reward Clipping | False |
| Extrinsic Reward Normalization | False |
| Intrinsic Reward Clipping | False |
| Intrinsic Reward Normalization | True |
| Steps Per Episode | 2048 |
| Total Steps | 2e8 |
| Number of Minibatches | 16 |
| Number of optimization epochs | 4 |
| Extrinsic Reward Coefficient | 1.0 |
| Intrinsic Reward Coefficient | 0.5 |
| Number of Environments | 128 |
| Learning Rate | 0.0001 |
| Entropy Coefficient | 0.001 |

Table 2.1: Default parameters for PPO with RND curiosity rewards as used in the reinforcement learning approach on the targeted drug delivery problem by Huang and Becker et al. [36].

**The Targeted Drug Delivery Agent** The agent for the targeted drug delivery problem is inspired by the agent used for the large scale study on curiosity learning by Burda et al. in 2018 [19]. Since the targeted drug delivery problem requires careful exploration of the environment, the agent heavily profits from the addition of a curiosity reward. For reward generation the original methods from [19] were used, as well as the random network distillation method (see Section 4.4.4). RND showed superior performance in all experiments which were done with both rewards.

The agent itself is build with an actor-critic architecture and trained using the PPO

algorithm (see Section 4.4.2). The neural network uses convolutional layers to process the image information (see Section 3.3.2) and follows the frequently used natural CNN design. The architecture can be found in Figure 2.3 and consists out of four convolutional layers with 32 ($8 \times 8, s = 4$), 64 ($4 \times 4, s = 2$) and 64 ($2 \times 2, s = 1$) filters which are then connected to a single fully connected layer with 512 neurons. The original four commands used to move the particles are extended by another four additional commands. Rather than just using "North" (up), "South" (Down), "West" (Right) and "East" (Left), the agent also is allowed to also use "Northwest", "Northeast", "Southwest" and "Southeast" as movement commands. Additional training parameters for PPO and curiosity can be found in Table 2.1.

The reinforcement learning approach proved to have exceptionally good performance on single (small) instances, but is costly due to the long training process. Huang et al. showed, that PPO was not able to successfully solve instances by using the extrinsic reward signal only. Even after extremely long training sessions with over 3 billion timesteps, PPO was not able to solve a single instance. The addition of the intrinsic curiosity reward solved this problem, by dramatically decreasing training time and leading to positive results even for larger and more complex instances.

# 3 Deep Learning

In this chapter we want to take a step back and talk about the the fundamentals of machine learning with a special focus on artificial neural networks. We will take a look at how these networks are built and how they can be trained. The term *Deep Learning* actually refers to the structural size of these artificial networks in modern applications which are build from tens to hundreds of layers.

We will begin with some basic motivation and get familiar with terms and notation for machine learning in Section 3.1. We then dive into the world of artificial neural networks and how they can be trained in Section 3.2, where we also talk about some of the most challenging problems for nowadays architectures and how they can be solved. Finally we will finish with some specialized network structures for specific applications in Section 3.3.

## 3.1 Machine Learning in a Nutshell

In this Section we want to give an introduction into machine learning. We will begin with some motivation and talk about why machine learning matters in Section 3.1.1 and follow up with some basic terms and notation in Section 3.1.2.

### 3.1.1 Motivation

Before we dive into neural networks, let us briefly talk about what machine learning is and what makes it special. Machine learning is a field of study of computer science, which tries to give computers the ability to solve a task without a human writing an explicit algorithm for it. Usually machine learning works by providing *training data* to a specialized learning algorithm which will then try to solve a task by optimizing a performance measure. For example think about a set of images, where some of the images show apples and others show pears. When training, we split our *dataset* into two - a larger one which will be our *training dataset* and a smaller one which will be our *test dataset*. Our algorithm will learn from the images of the training dataset only. When training we also reveal which of the two classes is the true class of a given sample so that the algorithm knows if it classified a sample right or wrong. Its progress is then measured periodically in terms of *accuracy* on the test dataset.

While we can expect a trained model to be fairly good, we can also expect to never reach 100% accuracy. Additionally machine learning is hard and time consuming so why do we not use traditional algorithms? There are a number of good reasons to use ML, but ML is especially useful because there are many problems where traditional algorithms

are either too complex or it does not even exist an algorithm to solve them. For example there is no known algorithm which precisely recognizes and classifies objects in an image. Machine learning can also be very dynamic. Imagine a spam detection engine. Traditional approaches worked with keyword filters which were simply lists of forbidden words (or sentences) created by humans. This approach required a lot of tedious work for a result which is not very robust, since someone who knows the filter list can try to simply use different words. In a setting where we use machine learning, we could let the computer create this list on its own. The only thing we would need to do is to tell it, which email contains spam and which does not. Machine learning can also go a step further and build a language model for us which allows to automatically detect synonyms - improving the robustness of the learned list.

Machine learning is not only great at solving complex tasks, it can also help humans to learn. Machine learning is a great tool to find information in big piles of raw data. This area is called *data mining* and can help to find unknown patterns to better understand problems.

### 3.1.2 Terminology and Notation

In this section we want to introduce some basic notation and terminology from the area of machine learning. We will further try to get a better view on what machine learning is from a mathematical point of view.

**Types of Machine Learning.** Machine Learning is a very broad area and - depending on the task - very different techniques will be used. Therefore learning algorithms can usually be divided in (at least) three main categories:

1. *Supervised Learning.* Supervised learning algorithms aim at building a model from a training dataset where for each sample in the training dataset the desired output is given. Typical uses are classification or regression tasks.

2. *Unsupervised Learning.* Unsupervised learning algorithms try to find structure in a given dataset without the desired output. They are often used for clustering which is done by exploiting some similarity metric. Unsupervised methods can often be combined with supervised methods and then are called semi-supervised methods.

3. *Reinforcement Learning.* Reinforcement learning tries to train an agent to take actions in an environment to reach a certain goal. The agent is usually guided by some reward given to it which it will try to maximize. We will take a detailed look at reinforcement learning in Chapter 4.

**Data Representation.** Data is not only a central element for every computer system, but also highly important for any machine learning algorithm. Without data, there is nothing we can learn from. Data can have arbitrary origin, but if the data comes from the real world

it is usually collected by sensors. The data is often preprocessed and the algorithm gets the data in form of a vector representation $\mathbf{x} \in \mathbb{R}^n$ which we call the *feature vector* $\mathbf{x}$ in the *feature space* $\mathbb{R}^n$. These terms originate from the idea, that each dimension of the vector may contain a certain feature (e.g. weight and color of a fish), but we can also work on "raw" features like the values of pixels in an image. If we have a set of feature vectors (samples), we denote the whole set by $\mathbf{X}$. Data is crucially important for the performance of the model. The best algorithm will not be able to create a good model from a bad dataset. In particular datasets may be too small or unrepresentative to model a real distribution, the dataset might be biased towards or against a certain group, or the samples of the dataset may not be independent which makes it hard to learn from them. Gathering good training data is therefore a central problem of machine learning.

**Machine Learning Models.** To solve a given task, machine learning algorithms build a model which is trained using the training data. We call a model which is done with its training process an *inference* model, which is then used at *test time* to solve the problem it has been trained for. There are currently a number of different models available, with their most prominent member being the artificial neural networks which we will discuss in the remainder of this chapter. Nevertheless there exist many other kinds of models like *decision trees* or *support vector machines* which all have their own advantages and disadvantages.

All models are in some way parameterizable and we will denote their parameters with $\theta$. Depending on the task, models have a specific output, which is usually also in vector form and denoted by $\mathbf{y}$. For example if we look at classification tasks, the output vector usually contains probabilities for each class $i$. The model can then be seen as a function which outputs a probability $p(s = i|\mathbf{x})$ for every class given the feature vector $\mathbf{x}$. Using *Bayesian Decision Theory*, simple models can be created by counting occurrences in the training dataset to estimate the likelihood $p(\mathbf{x}|s = i)$ and the prior $P(s = i)$. The class probability is then given by the Bayes formula

$$P(s = i|\mathbf{x}) = \frac{p(\mathbf{x}|s = i)P(s = i)}{p(\mathbf{x})}$$

The *evidence* $p(\mathbf{x})$ is only a scaling factor which can be ignored when we are only interested in the class with the maximum likelihood for a given feature vector. Even though bayesian models are optimal, in very high dimensional and complex feature spaces it is not possible to model the likelihood and prior exactly, which is why most modern machine learning techniques estimate them.

Of course there are many more applications for machine learning than classification, so the output of a model does not need to be a probability distribution over classes. Depending on the task the output might just be a single number, an image, a time series or an audio signal. There are no limitations.

## 3.2 Artificial Neural Networks

In this Section we want to talk about *Artificial Neural Networks* (ANNs) which is one of the most prominent model family for modern machine learning. We begin by giving a brief overview over their historical development in Section 3.2.1 and continue with their modern architecture in Section 3.2.2. We will then look at how we can train neural networks with the famous backpropagation algorithm in Section 3.2.3 and discuss an important component of ANNs - the activation functions - in Section 3.2.4. Finally we will look at some of the biggest challenges regarding the training of neural networks in Section 3.2.5.

### 3.2.1 History

Technical advances were often inspired by carful observation of the nature around us. In most animals we can find neural cell structures which are connected in a specialized way to control muscles and process information from sensory cells. Neurons are also the building block for our own brain. Naturally the idea of recreating neurons to produce intelligent machines was interesting and explored since the early days of computer science.

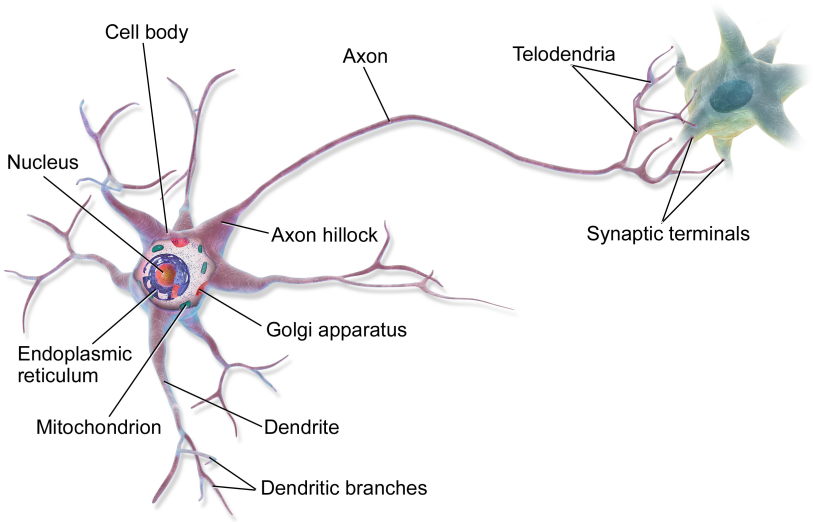



Figure 3.1: Biological Neuron[1]

Before we look at the first artificial neuron, let us first take a look at its biological counterpart for which we included a sketch in Figure 3.1. Since the neuron is a specialized cell, it contains a cell body with a nucleus. Around its core, the neuron branches out into *dendrites*. These dendrites can be seen as the inputs of the neuron. The output of the neuron is a single, long dendrite, called the *axon*. The axon can have extremely different length, ranging from less than a millimeter up to meters. At its end, the axon branches

[1]Image created by Bruce Blaus, released under Creative Commons under `https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png`

out into *telondendrias*, which end in *synapic terminals* (also called *synapses*). The synapses themselves are connected to the dendrites of other neurons or muscles. If the neuron receives a signal through its dendrites and the signal strength rises up to a certain level, the neuron answers by sending short electrical impulses called *action potentials* (APs) down its axon. These APs result in neurotransmitters being released in the synapses - transmitting the signal to the next cell. Neurotransmitters are chemical substances which means, the electrical (digital) signal gets converted into an analog signal at the synapses. This signal at the synapses can lead to a further transmission of signals or inhibit the next neuron from firing.

While a single biological neuron is fairly simple, neurons form extremely complicated structures. The human brain consists of around 86 billion neurons which work as a unit and form a lot of specialized substructures to control our body, process images from our eyes, or acoustic signals from our ears. Even more impressive is the fact, that these structures are not static and most of these capabilities can be learned and thus are flexible. Even though we do not understand all these processes in biological neurons in detail, we can try to use artificial version of these building blocks and experiment with them.

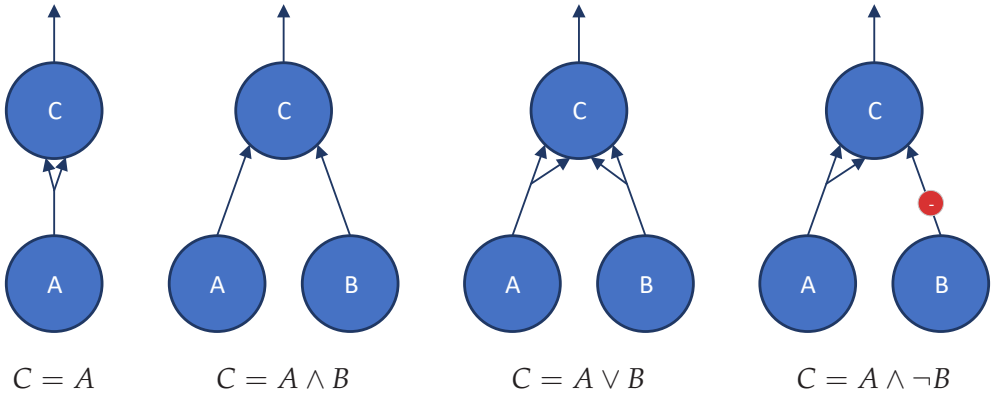



Figure 3.2: Artificial neurons for logical expressions. The neurons (circles) A, B compute the output C via connections (arrows).

The first version of this artificial neuron was proposed early in 1943 by Warren McCulloch and Walter Pitts [52]. They showed how logical expressions can be build with a network of binary neurons. Logical neurons activate their output if at least two of their inputs are active. Figure 3.2 shows an example for the basic logical operations *and*, *or* and *not*. It is clear that we can express arbitrary logical formulas in conjunctive or disjunctive normal form using these neurons. However neurons which only support binary values are not able to compute functions which is a major shortcoming. McCulloch and Pitts therefore also proposed a second version: The *Threshold Logic Unit* (TLU). TLUs are able to perform continuous linear function computations. They achieve this by first computing a weighted sum of the inputs $z = w_1 x_1 + w_2, x_2 + \cdots + w_n, x_n = \mathbf{x}^T \mathbf{w}$ and then applying a step

function. This step function sets the output to $+1$ if $z \geq 0$ and otherwise to 0. We included a graphical version which shows a single TLU in Figure 3.3. The behavior of the step function matches the behavior of biological neurons, which either fire their signal at full signal strength or do not fire at all. A single TLU is able to perform binary classification tasks (output is either 1 or 0 depending on the class). Training the TLU then means to find the right weights for each input. The question is how to find these weights?

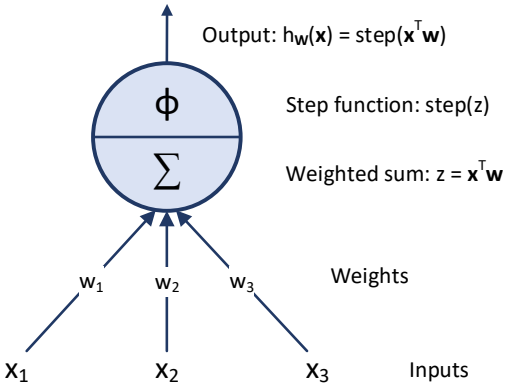



Figure 3.3: A *threshold logic unit* is an artificial neuron which computes its output by a weighted sum of its inputs and a special step function.

In 1957 Frank Rosenblatt invented the *Perceptron* as an extension of the TLU. A perceptron is a combination of multiple TLUs to a layer of neurons, which all are connected to the inputs. We included an example in Figure 3.4. Usually a bias neuron is added to the inputs to give the network more flexibility. The neurons in the first layer build the *input layer* and just pass the input signal to the neurons in the *output layer*. Both layers are *fully connected* meaning every neuron in the input layer is connected to every neuron in the output layer. The output of the output layers TLUs can be computed at once by

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

where $\mathbf{X}$ represents the input, $\mathbf{W}$ is a concatenation of all the weights of the TLUs (except for the bias neurons) and $\mathbf{b}$ represents the bias. For now the function $\phi$ is the step function.

Rosenblatt also proposed a simple algorithm which is able to compute the weights for all TLUs for a given classification problem based on the error the network made. This algorithm is based on the perceptron learning rule

$$w_{i,j}^{new} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

After the network classified the examples $x_i$, it updates the weights depending on the outcome - $y_j$ is the current output of the $j$th neuron and $\hat{y}_j$ the target value. The weights $w_{i,j}$ describe the weights between the $i$th input neuron and the $j$th output neuron. The parameter $\eta$ is the *learning rate* for the network. The perceptron originally was build as a

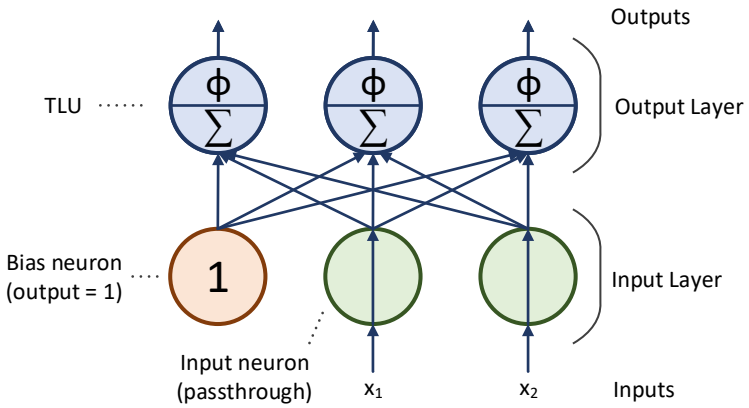



Figure 3.4: Example for a *perceptron* with 2 neurons and a bias in the input layer and 3 neurons in the output layer.

pure hardware implementation. The machine was connected to a camera and was one of the first machines which was able to learn weights for image recognition.
In 1969 it was proven by Marvin Minsky and Seymour Papert, that perceptrons are only able to compute linear functions and cannot compute a logical XOR from their inputs. Therefore perceptrons can only be used for simple linear classification tasks. However if the data is linearly separable, perceptrons are proven to converge.

### 3.2.2 Multi-Layer Perceptrons

So how do we actually learn nonlinear separation with TLUs if perceptrons are unable to do so? A simple but powerful extension to the original idea was to just stack multiple perceptrons on top of each other as shown in Figure 3.5.

This extended network actually allows to compute the XOR operation. Layers between the input and the output layer are called *hidden layers* since they have no connection to the outside world. Computations are done layerwise similar to perceptrons by computing $h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$ for each layer, where $\mathbf{X}$ is the output of the previous layer. Each layer may have an individual number of neurons. The name deep learning rises from the idea of stacking more and more layers on top of each other to build *deep neural networks* (DNNs). Because the input "flows" directly from the input to the output through each layer, these networks are also called *Feedforward Neural Network* (FNN). The idea of stacking multiple perceptrons on top of each other was known for quite some time, but the problem was that it has been unclear how these networks could be trained. The simple perceptron algorithm does not work anymore, cause we now need to compute the error with respect to every layer. Additionally even though stacked perceptrons are able to compute XOR operations they are still limited to learning linear functions, since they are only a combination of other linear functions.

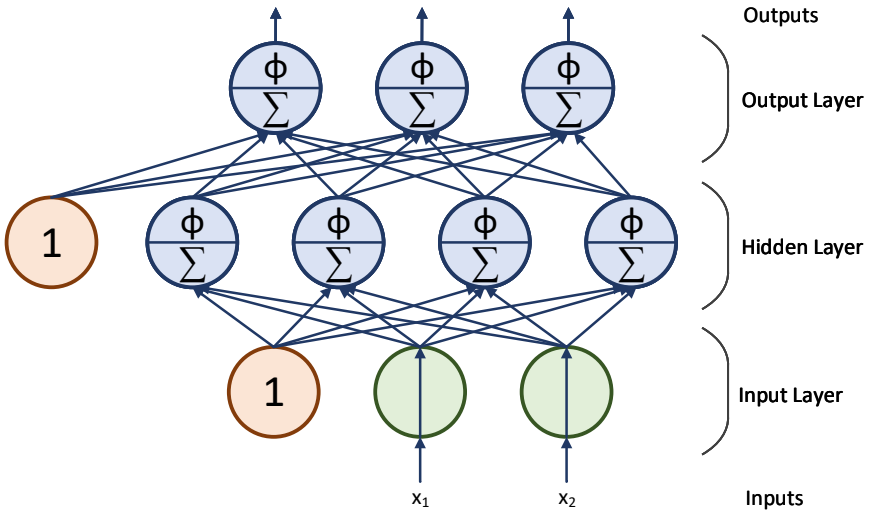



Figure 3.5: Example for a *multi-layer perceptron* network with 2 neurons and a bias in the input layer and 4 neurons and a bias in the hidden layer and 3 neurons in the output layer. The number of hidden layers and neurons per layer in a network can be arbitrarily large.

### 3.2.3 Gradient Descent and Backpropagation

It took a few years, until in 1985 Rumelhart et al. were able to develop an algorithm to train multilayer networks while simultaneously introducing nonlinearity [67]. Their algorithm combines the idea of *gradient descent* with a method called *backpropagation* which allows to calculate gradients for the weights of each layer of the network.

**Gradient Descent.** Let us first look at the gradient descent. Gradient descent is a method to calculate the minimum of a function. Given an arbitrary starting point on the function, gradient descent will iteratively find a series of new points leading towards a minimum (see Figure 3.6). Suppose we are at some point $x_k$ on our function $f$, we can then calculate the next step using the gradient $\nabla f$ of our function. We therefore can calculate

$$\delta_k = -D_k \nabla f(x_k)$$
$$\text{and } x_{k+1} = x_k + \eta_k \delta_k$$

as the next point closer to the minimum. The parameter $\eta$ is known as step size and in the context of deep learning is referred to as *learning rate*. At each step we want to calculate the steepest descent at the current point, since this will lead us quickly to the minimum, therefore we set $D_k = I$, with $I$ being the identity matrix. Note that there exist other approaches for second order optimization which use Hessians for $D_k$ instead.

The choice of an appropriate $\eta$ is crucial for gradient descent. Too large values will lead to points which oscillate around the minimum or even completely diverge from it, while

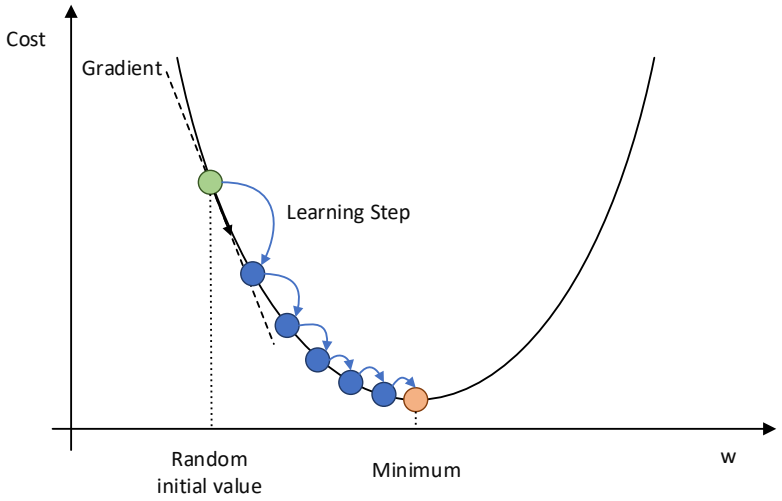



Figure 3.6: The idea of gradient descent.

too small values for $\eta$ might need many iterations to converge. Gradient descent also does not guarantee convergence to a global minimum and smaller learning rates increase the danger of getting stuck in a local minimum.

We now know what gradient descent does, but the question is which function can we minimize in the context of machine learning and how do we train a model by doing so? If we look at a neural network, it outputs a vector $\mathbf{z}$ at its output layer. This output is usually desired to give us some information (e.g. class probabilities in a classification task). When training we know the true output vector $\mathbf{y}$ the network should produce for some given input. Therefore we can calculate the error of the network in terms of a cost function over the dataset. Depending on the task there are several different cost functions which can be used. One of the most common ones is the *mean squared error* (MSE):

$$MSE(\mathbf{x}, \theta) = \frac{1}{n} \sum_{i=1}^{n} (z_i - y_i)^2$$

Since the output vector $\mathbf{z}$ is dependent on the input vector $\mathbf{x}$ and the network weights $\theta$, but we only have control over $\theta$, we often write $MSE(\mathbf{x}, \theta)$ as $MSE(\theta)$ for short. When performing gradient descent we then calculate the gradient with respect to $\theta$.

When implementing gradient descent for neural networks, we are sometimes interested in optimizing for multiple objectives at once. We therefore usually perform gradient decent with respect to a so-called loss function $\mathcal{L}(\theta)$ (which is equal to our cost function at the moment). We then denote the gradient of that function with respect to the network weights as $\nabla_\theta \mathcal{L}(\theta)$. A single gradient descent step for our network is then given by

$$\theta^{new} = \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

Because the calculations for the gradient descent step are costly, it is usually considered sufficient to only calculate the gradient for a single sample from the dataset in each step (as opposed to calculating the gradient as a mean over the whole dataset). This procedure is called *stochastic gradient descent* (SGD) and is a stochastic approximation of gradient descent. Training with SGD is less stable, but much faster. To stabilize training, SGD often uses *minibatches* of data, where the gradient is calculated over a smaller batch of samples instead with a single sample.

**Backpropagation.** Now that we know how we can calculate updates for the neural network, we need to know how it is possible to update the weights for each individual layer. This is where the *backpropagation algorithm* comes in place. It mainly consists out of two phases: A forward pass, which allows us to determine the error (e.g. using MSE) and a backwards pass which determines which layer was involved to which extend with the error.

Let us look at the phases in more detail. The forward pass takes a sample (or a batch of samples) and propagates them through the network. For each layer, we collect the output activations and store them for later. After passing the input through the network and the calculation of the final output, we calculate our error like before.

In the second phase we begin at the output layer and propagate the error backwards layer by layer. This is done by calculating the weight change $\Delta w_{i,j}$ from neuron $i$ from the previous layer to neuron $j$ from the current layer by:

$$\Delta w_{i,j} = -\eta \delta_j o_i$$
$$\text{with } \delta_j = \begin{cases} \varphi'(v_i)(o_j - y_j) & \text{if } j \text{ is in the output layer} \\ \varphi'(v_i) \sum_k \delta_k w_{j,k} & \text{if } j \text{ is in a hidden layer} \end{cases}$$

where the subscript always defines the number of the neuron in the layer. If the subscript is an $i$ we reference the previous layer (closer to the inputs) and $j$ references the current layer. We denote $v$ as the the pre-activation output, $o$ as the final post-activation output and $y$ as the desired output of a neuron. When using minibatch SGD steps the weight changes are accumulated in the backpropagation step and then applied all at once. Since SGD is very sensitive to the learning rate, the parameter needs to be tuned carefully.

### 3.2.4 Activation Functions

When we look at the backpropagation algorithm we can see, that we need a differentiable activation function to perform error backpropagation. Also the first derivate of the activation function must be nonzero, otherwise all our gradients would always be zero. In order to use SGD the step function needs to be replaced with another activation function. Replacing the step function with a nonlinear function also resolves another problem of the original stacked perceptron, since it introduces nonlinearity into the network. This change finally allows the MLP to compute arbitrary functions and makes it the powerful

tool it is today. Figure 3.7 shows an overview over frequently used activation functions. Let us take a short look on each of them:

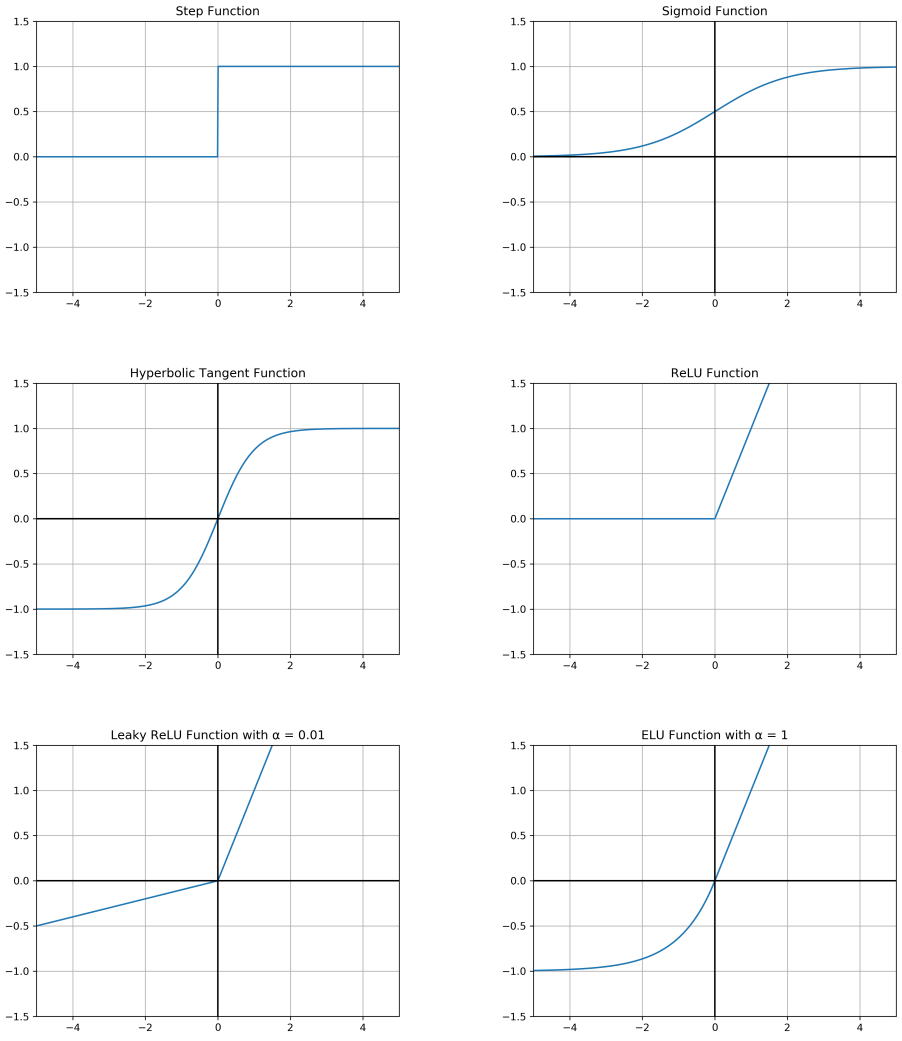



Figure 3.7: Common activation functions. The step function is the only function which cannot be used with the backpropagation algorithm.

1. The *sigmoid function* $\sigma(z) = \frac{1}{1+\exp(-z)}$
   The sigmoid function was originally proposed as a replacement for the step function. It is S shaped and can express values between 0 and 1. It also is continuous, so it has a well-defined nonzero derivate everywhere, while being still similar to the step function in terms of output values.

2. The *hyperbolic tangent function* $tanh(z) = 2\sigma(2z) - 1$
   The hyperbolic tangent function is similar to the sigmoid function, but its values

range from $-1$ to 1, so layer outputs are centered around 0 at the beginning of the training which speeds up convergence. However it was shown, that bounding the value of the activation function for large $z$ reduces convergence speed.

3. The *rectified linear unit function* $ReLU(z) = max(0, z)$
   The ReLU function is not differentiable at $z = 0$ and has a zero derivate for $z < 0$. Both these properties are not ideal, but the ReLU function is very fast to compute and still performs reasonably well in practice. It is also unbounded in terms of $z$ which further improves convergence. This makes the ReLU function one of the most used activation functions today.

4. The *leaky rectified linear unit function* $LeakyReLU_\alpha(z) = max(\alpha z, z)$
   One problem with the ReLU function is, that when the weighted sum of the inputs of a neuron is negative for all training instances, the neuron itself will never output anything other than 0 again - the neuron essentially dies. To prevent this the leaky ReLU function has a leak which makes the derivate of the function nonzero for $z < 0$ and prevents the neuron from never learning anything again. The parameter $\alpha$ controls how large the leak should be and can also be learned at training time [87].

5. The *exponential linear unit function*

$$ELU_\alpha = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

   Just like the Leaky ReLU function, the ELU function has a nonzero gradient for $z < 0$. If $\alpha = 1$ the ELU function is smooth everywhere. Therefore the ELU function performs better than the Leaky ReLU function, but it is harder to compute. [22] There also exists an extension called SELU which stands for *scaled exponential linear unit* [44] and results in network layers which include self-normalization and therefore preserve a mean of 1 and a standard deviation of 0 during training. This further speeds up training and also showed to improve the results at test time.

### 3.2.5 Challenges

We saw how we can build neural networks and how they can be trained using gradient descent. Problems - like handwritten recognition - which were extremely complicated for a long time can now be solved with learned models. While in theory we now have the tools to build and train networks, in reality we would quickly encounter problems with networks which just do not learn what we want them to learn. In this section we want to present some of the most common problems when training neural networks, while also looking at techniques used to avoid them.

**Slow Convergence.** While gradient descent is guaranteed to converge to some minimum if the learning rate is sufficiently small, it makes no guarantees on how much time this convergence process might need. Our simple gradient descent algorithm is only dependent on the first derivate and therefore only "looks" at the slope for the current point. This produces two major problems: If we are on a flat spot of the optimization surface, the gradients will be very small, slowing down training in an "uninteresting" area. The second problem is that always going into the direction of the steepest descent is not the fastest way to the minimum. Instead we would need to additionally look at the second derivate to determine which direction would be optimal.

To reduce the influence of these problems, a variety of techniques were developed over the years. The basic idea is, that the learning rate does not necessarily have to be a constant, instead we change the step size for each step of gradient descent dynamically depending on the curvature of the optimization surface. For flat regions we could increase the learning rate to compensate for the smaller gradients and lower the learning rate if we detect unstable learning. Directly calculating second derivates has shown to be computationally too expensive. Instead algorithms try to make assumptions about the surface by looking at past gradients. An example of these methods is the idea of *momentum*.

The momentum method was also proposed by Rumelhart et al. in their backpropagation paper [68]. The name originates from real physical momentum, where we think of a particle which is accelerated by a force in a direction. For gradient descent the idea is, that successive gradients usually point in roughly the same direction. Therefore we could improve convergence speed, by just applying the same gradient multiple times. To avoid overshooting, we remember the old gradient and at each step compute the new gradient as a linear combination:

$$\Delta w = \eta \Delta w - \alpha \nabla f(x_k)$$

Momentum helps to reduce oscillation and is better at avoiding local minima, since the learning process now has the possibility to go over small "hills" in the optimization surface.

The momentum method can easily be combined with a second idea: *Learning rate schedules*. Picking an initial learning rate that is optimal for the whole learning process is hard. Initially we will be far from the optimum and need a higher learning rate to converge faster and avoid local minima and over time, the training process might get closer to the optimum but begin to oscillate around it, because the learning rate is too high. To always work with an optimal learning rate, we need to adjust it over time and these strategies are the learning schedules. Some often used schedules are:

1. *Linear Scheduling.* The easiest schedule is a linear schedule where the learning rate is a function of $\eta(t) = \eta_0 - (\eta_0 * t/s)$. The effective learning rate is linearly annealed from a starting learning rate $\eta_0$ to zero. The parameter $s$ is set to be the total number of training steps for SGD.

2. *Power Scheduling.* The learning rate is a function of the form $\eta(t) = \eta_0/(1 + t/s)^c$

with the hyperparameters $c$ - the exponent - and $s$ - the number of steps. With power scheduling, the learning rate will decrease every $s$ steps, arriving at $\eta_0/2$ after $s$ steps, $\eta_0/3$ after $2s$ steps and so on. This results in an effective learning rate which quickly drops and then decreases more and more slowly over the course of training.

3. *Exponential Scheduling.* The learning rate is a function of $\eta(t) = \eta_0 \cdot 0.1^{t/s}$. The learning rate will therefore drop by a factor of 10 every $s$ steps.

4. *Performance Scheduling.* Measure the validation error every $N$ steps and reduce the learning rate by a factor $\gamma$ when the error did not drop.

There also exist a lot of other schedules like piecewise scheduling, where the learning rate is defined for specific parts of the training process. The choice of the learning rate schedule is dependent on the task and also influenced by other factors like overfitting.

Since the learning rate is the most important hyperparameter for deep learning over the years many other more sophisticated methods were developed, most notable *AdaGrad* [25], *RMSProp* [78] and *Adam* [43] which are usually used in nowadays neural network frameworks. All these algorithms use the past gradients to calculate per-parameter learning rates based on a given starting learning rate. They often make use of momentum optimization and can be used in conjunction with learning rate schedules.

**Overfitting.** *Overfitting* is a recurring problem for all machine learning techniques. When training on a training dataset, we expect the trained model to have the same performance on new never seen before data. To achieve this, our goal is to learn a representation of our input data. The problem arises when the model is too closely fit to the training dataset, which often happens if our model has too many parameters and if we trained our model too long. Overfitted models often show an extremely low training error of close to 0%, while on test time the model performs way worse. Overfitted models usually did not learn a representation of the data, but instead just "remember" each sample from the training dataset. We therefore also speak about models which fail to generalize. An example for overfitting in a classification task is given in Figure 3.8. We can see, that the overfitted model did not ignore the noise in the training data and instead created a decision boundary which perfectly separates both classes in the training dataset, but does not model the true distribution behind the data.

Neural networks generally have a high tendency to overfit on the training data, because their size and the resulting number of parameters enables the network to very closely fit to the training data. When training, it usually makes sense to create a balance between the number of samples we have to train the network and the number of parameters (neurons) inside of our network. If we do not have much training data we need a smaller neural network to avoid overfitting too quickly. Overfitting is also influenced by the learning rate. If the learning rate is high enough, gradient descent will never converge completely and

[2] Original image created by user Ignacio Icke, released under Creative Commons under `https://commons.wikimedia.org/wiki/File:Overfitting.svg`, edited and extend with underfitting example.

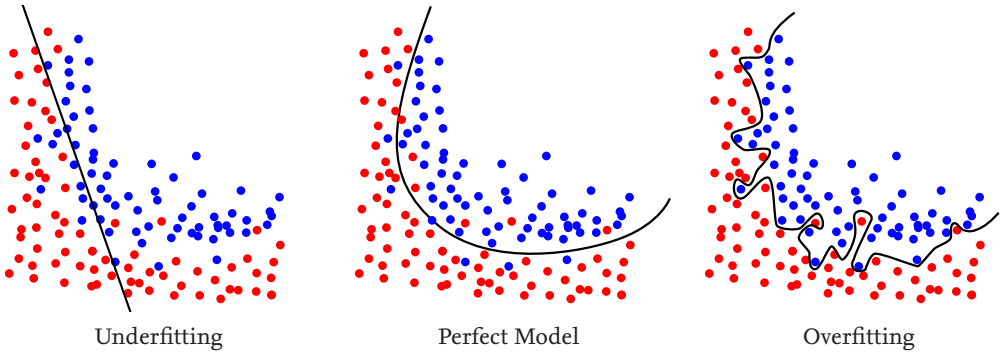



Figure 3.8: This example demonstrates the effects of over- and underfitting for a 2-dimensional dataset with samples of two classes blue and red. We can see, that neither the underfitted nor the overfitted models will work well on test data, even though the overfitted model will have a 0% error rate on the training dataset.[2]

the network will therefore generalize better. Therefore the danger of overfitting increases when using advanced optimizers or learning rate scheduling to adapt the learning rate dynamically.

Luckily there are better ways of preventing overfitting than using a high learning rate or not use advanced optimizers. In the following we present a number of methods, which can be used in conjunction to prevent or detect overfitting.

1. *Early Stopping.* Before we start the training process, we can split our training data into two datasets - a training and a validation dataset. When training the neural network, we periodically test the performance of the model on the validation dataset. If the model did not improve for a certain amount of time on the validation dataset (or even gets worse) we stop training to avoid overfitting. Early stopping is very effective and easy to implement, but requires to split the training data which results in less training data for the learning process.

2. *Regularization.* One of the best ways to prevent overfitting is to use regularization methods to keep the weights of the neural network from getting too large. To achieve this a regularization term is added to the loss function. There are three widely used methods:

   - $l_2$ regularization from *Ridge Regression* (also called *Tikhonov* regularization) can be used to explicitly punish single large weights in the network. To achieve this a regularization term is added to the loss function which adds a weight penalty according to the $l_2$ norm:

     $$\mathcal{L}_{l_2} = \frac{\gamma}{2} \sum_{i=1}^{n} \theta_i^2$$

     The sum starts at $i = 1$, because the bias term is usually not regularized. There

also exists an alternative idea of how to implement $l_2$ regularization called *weight decay* which just multiplies each weight with a constant $c < 1$ after each training step.

- $l_1$ regularization from *Lasso Regression*. $l_1$ regularization also adds a term to the loss function, but this time the $l_1$ norm is used so the weights contribute equally.

$$\mathcal{L}_{l_1} = \gamma \sum_{i=1}^{n} |\theta_i|$$

An interesting characteristic of the $l_1$ regularization is that it produces weights close to zero for the least important features. This produces a sparse model where many connections will have a zeroed out weight.

- *Max-Norm Regularization* is another regularization technique which simply limits the $l_2$ norm of the connection weights $\mathbf{w}$ to a certain value, such that $||\mathbf{w}||_2 \leq r$. Max-Norm regularization can be implemented similar to weight decay by rescaling the weights after each training step to $\mathbf{w} \leftarrow \mathbf{w}r/||\mathbf{w}||_2$ if needed.

3. *Dropout*. Dropout is one of the latest regularization techniques which was proposed in 2012 by Hinton et al. [34]. At each training step, every neuron (excluding output neurons) has a probability of $p$ to be ignored ("dropped out") during the current training step. The probability is usually set surprisingly high to values between 10% and 50%. While being simple to implement, dropout is not only a powerful regularization technique, but also makes the network more robust.

   You can think about dropout in an interesting way: Since every neuron has a chance to be dropped out or be present at each training step, a network with dropout essentially represents $2^N$ distinct networks at once. The resulting network during test is composed out of all these smaller networks building a strong ensemble classifier. Neurons in the network are also less likely to co-adapt, making the network less error prone for slight input changes like noise at test time and improving overall generalization.

**The Unstable Gradients Problem.** When we talked about the backpropagation algorithm in Section 3.2.3 we stated, that it is possible to train arbitrary neural networks with it, regardless of the number of layers. While this is true to some extend the backpropagation algorithm initially struggled to train networks with more than three layers. Since the gradients are computed layer-wise and propagated backwards from the outputs to the inputs using the connection weights, the gradient between the layers $i$ and $j$ (meaning layer $i$ is the input layer to layer $j$) shrinks or grows depending on

$$F_{i,j} = ||\phi(v_j) \cdot \mathbf{w}_{i,j}||_1$$

This means, that if $F_{i,j} < 1$ the gradients tend to get exponentially smaller for each layer which is called the *vanishing gradients problem* and if $F_{i,j} > 1$ the gradients tend to exponentially grow which is called the *exploding gradients problem*. Both problems lead to networks, which contain upper layers which do not learn anything and are therefore often worse in performance than networks with less layers.

To prevent the gradients from exploding or vanishing, we need to look at both, the activation function and the weights. As for most problems, there are multiple solutions which can be used standalone or in conjunction with others, to enable the BP algorithm to properly work for deep neural networks:

1. *Non-Saturating Activation Functions.* We already presented a number of activation functions in Section 3.2.4. The initially used sigmoid activation function had two crucial problems for its application in deep neural network. First, its slope is at most 0.25, which means errors in backpropagation always tend to get smaller with each layer. Second, the function saturates to 0 and 1 for very small or very large inputs, leading to very small gradients at these points and thus very slow learning. The solution is to use one of the other activation functions which do not saturate to a maximum value. Therefore deep neural networks usually use either ReLU, Leaky ReLU, ELU or SELU activations.

2. *Improved Network Weight Initialization.* In 2010 Glorot and Bengio proposed a way to deal with unstable gradients, by using a new method to initialize the weights [28]. Usually weights in neural networks were randomly initialized with a normal distribution with a mean of 0 and a standard deviation of 1. Glorot and Bengio showed, that for backpropagation we need to control the variance of the outputs of each layer in both directions - for the forward pass and for the backpropagation. This means, that the output distribution of each layer should have equal variance to the input distribution and at the same time we need the variance of the gradients needs to be the same before and after each layer. Completely satisfying this constraint is only possible if two adjacent layers have the same number of neurons, but the initialization can still be optimized. If we denote the number of input neurons by $fan_{in}$, the number of output neurons by $fan_{out}$ and the average number by $fan_{avg} = (fan_{in} + fan_{out})/2$, the Glorot initialization is given by

$$\text{A normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{fan_{avg}}$$

$$\text{Or a uniform distribution between -r and +r, with } r = \sqrt{\frac{3}{fan_{avg}}}$$

This initialization strategy has proven to successfully minimize effects of the unstable gradients problem at the beginning of training. Depending on the used activation function, the initialization strategy needs to be adapted slightly. ReLU and ELU

activation functions use the so-called He initialization with $\sigma^2 = 2/fan_{in}$ and the SELU activation function uses the LeCun initialization with $\sigma^2 = 1/fan_{in}$.

3. *Batch Normalization.* The problem with a good initialization of the network is, that the problem is only prevented from appearing directly at the start of the training process and might reappear later on during training. Therefore in 2015 Ioffe and Szegedy proposed a new technique called *Batch Normalization* (BN) to keep the output of each layer normalized with respect to some distribution. Their algorithm adds an additional operation before (or after) the activation function of each layer to zero-center, scale and shift the input for each layer. The parameters for the shift and scaling operations are learned at training time by computing an average over the current minibatch (therefore the name *batch* normalization). The input for each layer can then be calculated in four steps:

   1) Calculate the mean input vector over the minibatch $B$: $\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^i$

   2) Calculate the standard deviation over the minibatch $B$: $\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^i - \mu_B\right)^2$

   3) Calculate zero-centered and normalized input vectors: $\hat{x}^i = \frac{x^i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$

   4) Calculate rescaled and shifted outputs: $z^i = \gamma \otimes \hat{x}^i + \beta$

   The scaling parameter $\gamma$ and the shift parameter $\beta$ are learned via backpropagation together with the other variables of the neural network. Because $\mu_B$ and $\sigma_B$ are only calculated for the current batch, it is not ideal to use them at test time. Instead they are usually calculated by either recalculating them over the whole training dataset, or to calculate them at training time by using a running mean. Batch normalization significantly improves the performance of deep neural networks, but adds additional complexity at both training and test time.

4. *Gradient Clipping.* To avoid overhead like in batch normalization, an easy way to deal with exploding gradients is to just clip the value of the gradients to be in a certain range, e.g in $[-1, +1]$. This was first used by Razan et al. in 2013 for the training of recurrent neural networks, which are complicated to combine with batch normalization [59]. To keep the direction of the gradient it is also possible to clip by normalizing the gradient vector to the $l_2$ norm.

5. *Residual Networks.* Another take on the unstable gradients problem is the use of residual neural networks. These networks contain skip connections (shortcuts) which directly feed the output of one layer to another layer which is not its direct successor. Figure 3.9 shows a residual building block for a network.

   Skip connections are motivated by biological structures in the cerebral cortex. In 2016 Xu et al. first used residual connections in artificial neural networks for image

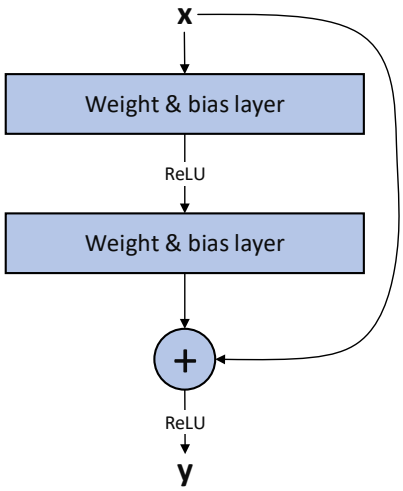



Figure 3.9: The structure of a residual block. Deep neural networks without skip connections are not able to learn the identity for a layer when trained with BP. Therefore larger networks often perform worse than shallower ones. With skip connections, networks can learn deviations from the identity, which works far better and also eliminates unstable gradients.

recognition [30]. They showed, that large networks with residual connections are faster to train and yield better end results without batch normalization. By skipping layers, the network is able to propagate errors from the output directly into layers close to the input of the network, allowing networks which have hundreds of layers. Having lower level features at later stages of deep networks also seems to be beneficial for the overall performance.

## 3.3 Specialized Network Structures

Until now we only were working with fully connected feed forward networks. While every fully connected layer is able to model any other arbitrary connection with the next layer, it is faster to train networks which already contain a certain structure, if that structure directly benefits the task. Especially image recognition has shown to be hard, even for larger multilayer networks. Therefore a special architecture is used when dealing with pixel inputs which we will explore in Section 3.3.2. Sometimes we are also confronted with continuous input data which requires memory. To deal with time dependent input data, another special network architecture exists, which we will take a look at in Section 3.3.1.

### 3.3.1 Recurrent Neural Networks

Feed forward neural networks are great for making predictions on data without a time component, e.g. images classification tasks. But if you want to predict the future, you often need to remember the past. And that is one thing our networks are missing: A memory. Think of a self-driving car, which has to keep track of all its surroundings. A pedestrian might walk on the sidewalk and at some point gets obscured by a parking car. If we remember that we saw that pedestrian some time ago, we can expect him to suddenly reappear and maybe cross the road. This is especially important if our pedestrian is a child which might not watch out for a car. A normal feed forward network would forget about the child as soon as it can not see it anymore, but if we had some kind of memory we would be able to remember the child and slow down. This situation cannot always be solved by adding timed information as input to the neural network (e.g. having a sequence of images from a video stream as input rather than a single image) as we do not know ahead of time for how long our memory has to last.

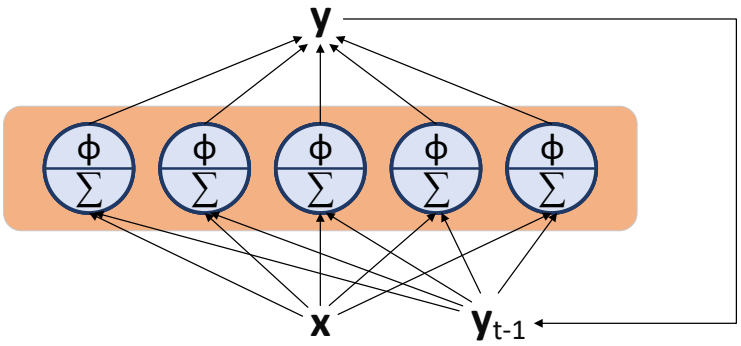



Figure 3.10: A layer in a recurrent neural network. The output $\mathbf{y}$ of the layer is used as a input for the next timestep.

To solve this problem, we modify the neurons of our network. Instead of having only a single flow direction from the input to the output, our new network also has connections pointing backwards. This type of network is called a *recurrent neural network* (RNN). Usually each layer of a RNN takes its own output as additional input vector in the next time step. This way, the network is able to remember its past inputs and its decisions at the same time. We included a representation of a recurrent layer in Figure 3.10. At step $t$ each recurrent neuron receives two input vectors: The current input vector $\mathbf{x}_t$ and its own output vector of the previous step $\mathbf{y}_{(t-1)}$. Accordingly each recurrent neuron has two sets of weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$. If we look at the whole layer of neurons and their weights $\mathbf{W}_x$ and $\mathbf{W}_y$ we can compute the layer output by

$$\mathbf{y}_t = \phi(\mathbf{W}_x^T \mathbf{x}_t + \mathbf{W}_y^T \mathbf{y}_{(t-1)} + \mathbf{b})$$

Recurrent neural networks are harder to train than regular feedforward networks, but essentially still rely on classical backpropagation. The trick is to unroll the network through time and then train it using regular backpropagation. This method for training RNNs is called *backpropagation through time* (BPTT). BPTT unrolls the network for a fixed number of timesteps $T$ into the past. The first recurrent vector is set to 0 and the output of each unrolling step is evaluated by a cost function. The gradients of the cost function are then backpropagated through the unrolled network, so the weights and biases are actually updated multiple times for a single learning step.

Recurrent neural networks can be used for a multitude of tasks which require time series data. They are especially well suited for natural language recognition, processing or translation as well as forecasting of future sequences (e.g. stock market prices) or the processing of video input. Since training of RNNs is hard because the unrolling process produces very deep network structures, often a modern improved version of RNNs called *long short-term memory* (LSTM) is used, which was developed in 1997 by Hochreiter and Schmidhuber [35]. LSTMs implement a completely new and complex model for a single neuron which allows it to remember information for longer periods of time and be trained more efficiently.

### 3.3.2 Convolutional Neural Networks

Although many tasks can quickly be solved by standard feedforward neural networks, performing tasks on images proofed to be notoriously hard. While detecting objects in a picture seems to be effortless for humans, machine learning struggles with every aspect of image recogition. But what makes the task so easy for humans and so difficult for machines? Since humans (and most other animals which live on land) rely on their ability to see, they developed a complex preprocessing pipeline which allows them to see, without learning how to see. Before the signals from our eye reach our visual cortex, they are preprocessed and already transformed into high-level features. Our brain therefore never needs to deal directly with the raw signals from our retina. Even though our brain is incredibly powerful, this preprocessing allows it to handle images in a fast and reliable way (well most of the time, since the preprocessing is also responsible for how many optical illusions work).

In 1958 and 1959 David Hubel and Torsten Wiesel experimented with cats to understand how their visual cortex works [37, 38]. They found out, that neurons that are directly connected to the retina only react to local stimuli on a specific region. They also were able to show, that certain neurons only react to images of horizontal lines, while others only react to lines with different orientations. Later neurons then combine these lower level features and only react to complex features build from these lower level patterns. In 1998 LeCun et al. were able to adapt this structure for artificial neural networks in their *LeNet-5* architecture [46] which is the first *Convolutional Neural Network* (CNN). CNNs introduce two new building blocks which build the foundation of modern image recognition: The *convolutional layer* and the *pooling layer*.

**Convolutional Layers.** Convolutional layers resemble their biological counterparts by not being fully connected. Instead each neuron of the convolutional layer is only connected to a small region on its input layer. This means each neuron only reacts to local signals of a specific $n \times m$ region inside of the previous layer. Each additional layer therefore learns higher level features, based on the lower level ones from the previous layer. In Figure 3.11, we show an example of a convolutional layer. The neuron in the $i$th row and $j$th column is connected to the neurons of the previous layer in a window with size $f_h \times f_w$ and centered at $(i, j)$. To avoid each layer from getting smaller than the previous one, the layer can be padded with zeros. If the layer instead should reduce the complexity, the windows can be used with a stride $s$ to achieve a downscaling effect. The stride defines how far the window is shifted between two neurons.

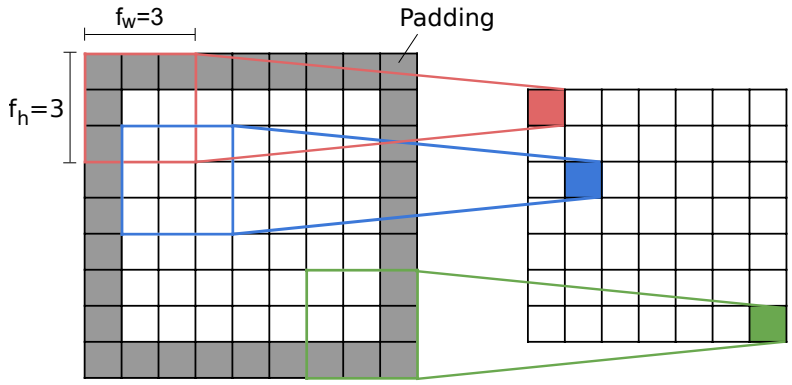



Figure 3.11: Connections in a convolutional layer with a window of $3 \times 3$ and a stride of 1. The input is padded to avoid downscaling. Neurons are only connected locally with rectangular receptive fields.

An important idea when using convolutional networks is, that a feature that is interesting in one area (e.g. horizontal lines) may also be interesting in every other area of the input image. We call the neurons whose weights compute a specific feature a *filter* or *kernels*. If we use the same weights and biases of the filter for the whole layer (called *weight tying*), the output of the layer is a *feature map*. This feature map contains the output of a filter placed at every position at the previous layer. To allow the network to learn multiple filters, a layer in a convolutional network is a stack of feature maps generated by multiple filters. Each feature map is connected to the full depth of all feature maps of the previous layer. For example if we have an $100 \times 100$ RGB image (3 channel input) and construct 20 filters with size $5 \times 5$, every neuron of the first convolutional layer will be connected to $5 \times 5 \times 3 = 75$ input neurons. The layer then will output 20 images of size $100 \times 100$ as output, one for each filter. Filters of size $5 \times 5$ of a second convolutional layer will therefore work on a input size of $5 \times 5 \times 20 = 500$ values.

The fact that each filter only needs to be trained once while being able to use it on every part of the input image produces a large advantage compared to fully connected layers. Its internal structure allows the CNN to vastly reduce the number of parameters needed

to express a filter operation over the complete image. The only downside is, that a CNN produces large intermediate results, because each filter outputs a complete copy of the image which needs a lot of memory.

**Pooling Layers.** Pooling layers work similar to convolutional layers only on a local part of the input, but serve a different function. Pooling layers *subsample* (i.e. shrink) the input. We already saw, that our intermediate results get pretty large, so by downscaling them, we are able to save computational resources, reduce memory usage and also reduce the numer of parameters (which prevents overfitting). Pooling layers also introduce invariance against small translations in the input image.

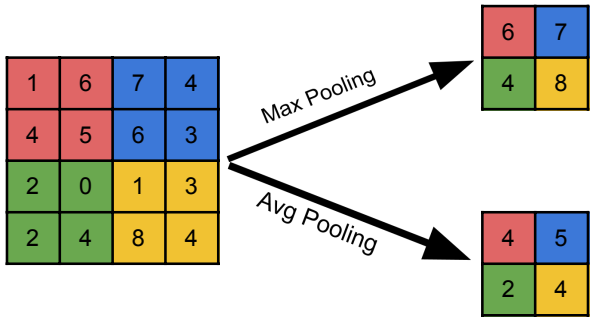



Figure 3.12: A Pooling Layer with size $2 \times 2$ and a stride of 2. The output on the top represents max pooling the output on the bottom average pooling.

Let us look at how pooling layers work. Like convolutional layers they are only connected to a small rectangular receptive field, but they typically do not connect to the full depth of the previous layer - they only subsample the individual "images". Pooling layers have no weights associated to them. Instead they use a pooling function to aggregate the inputs. Usually the *max* or the *mean* function is used. Figure 3.12 shows an example of a pooling layer with $2 \times 2$ kernels with a stride of 2. On the top we have the output of a *max pooling layer*, which means that out of four values from the input image only the maximum will be propagated to the next layer and on the bottom is the output of a *average pooling layer* which averages over all its inputs. While earlier networks often used average pooling, nowadays often only max pooling is used.

Pooling layers typically use a stride which is equal to their own size. While pooling is very useful to save computational resources, pooling larger areas than $2 \times 2$ can lead to information loss, so pooling has to be used carefully. However in current CNN architectures, pooling layers are used after every convolutional layer. By using this structure the intermediate results get smaller and smaller and the repeated pooling results in a larger

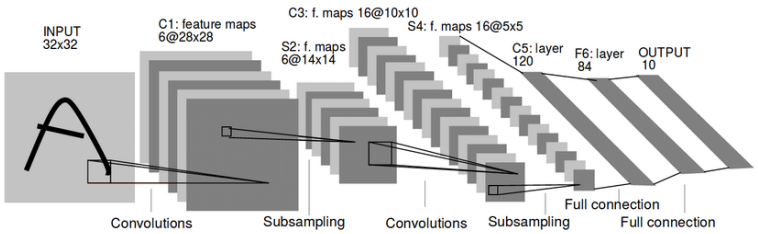



Figure 3.13: The architecture of the first CNN LeNet-5 [46]. It was designed to recognise handwritten digits and used two convolutional and two pooling layers.

invariance against pixel shifts or even smaller rotations in the input image.

Convolutioal networks usually end with one or more fully connected layers. We included a sketch of the original LeNet-5 architecture in Figure 3.13 which demonstrates a typical CNN. Modern architectures are often larger in every aspect and often use convolutional layers in conjunction with residual connections to allow efficient training of very deep networks. Recent papers also suggest to only use convolutional layers (*Fully Convolutional Networks* (FCNs)) for specific tasks like object detection [48]. To further improve invariance against scaling and rotation, the feature output of multiple convolutional layers can be directly combined, which is called Feature Pyramid Network (FPN) [47].

# 4 Reinforcement Learning

In this chapter we want to give an overview over existing methods for reinforcement learning. RL algorithms can be grouped into multiple families. We provide an overview over some of the most popular algorithms from each family in Figure 4.1. Because many algorithms are modular an accurate taxonomy of RL algorithms is difficult. For now we divide the algorithms into three basic groups - the value-based, the policy-based and the model-based methods. While model based methods try to model (or start with a model of) the environment to make predictions which action is the best option, value- and policy-based methods try to learn to predict the best action directly without also modeling the environment.

Over the years these basic algorithms were (partially) combined into methods which inherit ideas from multiple basic groups. On the one hand we mainly have variants of the actor-critic algorithm which all combine value-based and policy-based methods. On the other side we have various combinations of value-based or policy based methods mixed with ideas from the model-based family. For example the earlier mentioned AlphaZero algorithm uses value estimation in combination with *Monte Carlo Tree Search* [75].

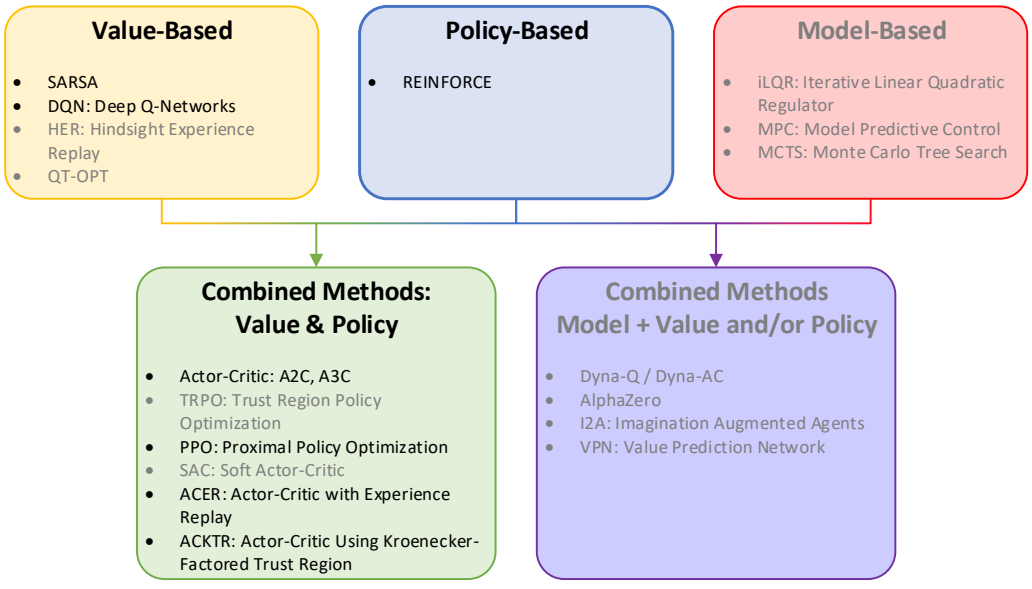



Figure 4.1: Overview of reinforcement learning families (adapted and extended from [42]). Methods that will not be used in this work are greyed out.

In this work we will be focusing on actor-critic variants and value-based algorithms, so we will not cover model-based algorithms here. However we will be taking a look at

policy-based methods, since they provide basic building blocks for the algorithms we will be using.

In this chapter we will begin with some basic ideas and challenges of reinforcement learning, as well as with some mathematical foundations in Section 4.1. After that we will continue by introducing our first family of RL algorithms - the value-based methods - in Section 4.2. We will continue with a short overview over the idea of policy-based algorithms in Section 4.3 and will finally cover combined algorithms in Section 4.4 where we will focus on the actor-critic method and its successors.

## 4.1 Key Concepts

In this Section we give some basic insight into key concepts of reinforcement learning. First we give some basic intuition into how reinforcement learning works and what the challenges are when learning behavior from an unknown environment in Section 4.1.1. We then continue by defining some of the basic terminology in Section 4.1.2 and finish with a different look on the RL problem from a mathematical point of view in Section 4.1.3.

### 4.1.1 Basic Ideas and Challenges

Reinforcement learning at its core tries to solve some arbitrary decision-making problem. This problem is given by an *environment* which produces information about its *state* in form of an *observation*. At each timestep an *agent* receives the current observation and then acts according to some *policy* by choosing from a set of available *actions*. These actions should ultimately lead to the achievement of some objective inside of the environment. The behavior of the agent usually is encouraged by some kind of *reward* which is returned by the environment. This basic agent-environment interaction loop is shown in Figure 4.2.

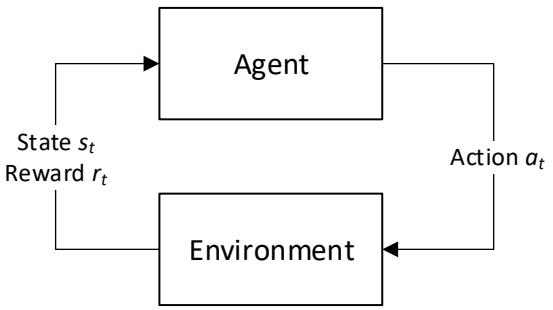
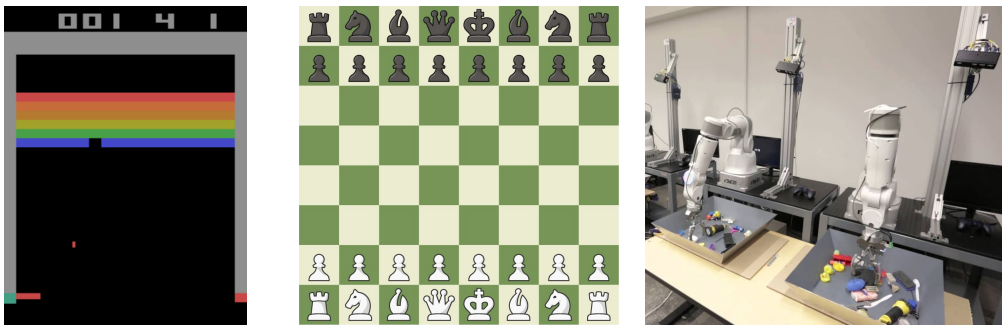



Figure 4.2: The basic reinforcement learning control loop for agent-environment interaction.

This abstract setting can be used to solve a variety of tasks. We have three examples given in Figure 4.3.

1. **Playing the game Breakout.** Breakout is a game from the game console Atari. The goal of the game is to break the bars at the top, by reflecting a moving ball with a paddle at the bottom (like in Ping Pong). The observation is an rgb image of the game itself and the possible actions include resetting or firing the ball as well as moving the paddle to the left or right side of the screen. The reward is the current game score, which increments with every broken piece of barrier at the top.
2. **Playing the game of chess.** The input might be a complex representation of the positions of all pieces, or just a plain rgb-image. The possible actions change after every move, because they are dependent on the valid moves of all pieces. Reward might be as simple as winning $(+1)$ or loosing $(-1)$ the game, but can be more complex.
3. **Grasping objects in a container.** A robotic arm should be trained to grasp arbitrary objects in a container. The observation is a plain image captured from above the robotic arm and a positive reward is given if the robot is able to successfully grasp an object. Actions include positioning the robot arm and opening or closing the "hand".



(a) The Atari game Breakout (b) Chess[1] (c) Google Robot Grasping [2]

Figure 4.3: Reinforcement learning examples.

Let's take a closer look at another example, pictured in Figure 4.4: The game of balancing a stick on a cart in a 2D environment. The stick can only be balanced by moving the cart, which has some velocity and can either be accelerated to the left or to the right. At each timestep the agent gets an observation which describes the current position and angle of the stick and can then choose which action it wants to take. If the stick falls over the agent looses. If it is able to balance the stick for more than 200 steps it wins. At each step it survives it gets a reward of $+1$.

Balancing a stick is harder than it initially seems to be. A simple algorithm which always moves the cart towards the direction the stick is leaning to, will always overshoot after a short time and loose balance. Of course we can think of a more complex algorithm and

[2]Image from chess.com `https://www.chess.com/bundles/web/images/offline-play/standardboard.6a504885.png`

[2]Image from ai.googleblog.com `https://i.ytimg.com/vi/iaF43Ze1oeI/maxresdefault.jpg`

maybe solve the problem this way, but what if we want the computer to figure things out? How can we implement an algorithm which figures out on its own which actions should be taken for some given observation? How would we ourselves solve this problem? The answer is simple: Try and error. A human learns how to balance a stick, by balancing it. This will of course fail at the beginning, but after some time a human would get better and better at *estimating* how the stick behaves in response to the actions that were taken. Similarly our reinforcement learning algorithm should just play around and look at the results. Did the stick fell over? How long were we able to balance the stick? At which point did it fell and which actions led to that outcome? If we can answer all these questions we might be able to learn how to solve the problem.

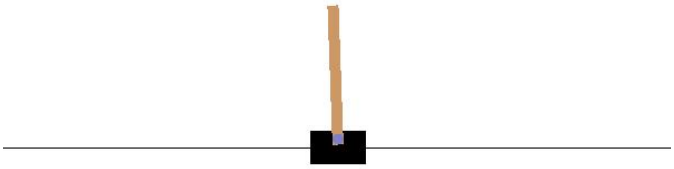


Figure 4.4: A classic toy example from OpenAI gym [18]: The CartPole environment.

Building an association between some action and the resulting outcome in the environment is actually one of the greatest challenges for RL algorithms and known as the *credit assignment problem*. Since rewards are typically sparse and delayed, understanding their origin is hard. Especially if you think about more complex games like chess, it is really unclear to what extend which move was responsible for winning or loosing a game. This problem becomes even greater the more sparse the rewards get: If we only reward winning or loosing, figuring out what made us win or loose is pretty hard, but if we reward or punish every single action, we might end up with a copy of an already known strategy. The challenge is to create an unbiased reward signal which does not distort the learning process. In the case of balancing a stick this is easy. Balancing the stick for a longer time is always good and can be rewarded. But as problems become more complex it is unclear if an action is actually good. If we take another look at chess for example, it might be beneficial to take a piece of your opponent, but it might also be better to go for a check or something which just gives you a positional advantage. Which action should be rewarded to what extend? In the end it is also a balance with how much time there is to learn. Sparse rewards will always result in longer training times, but give more freedom and thus might yield better results.

In the presence of sparse rewards another challenge arises: What if our learning algorithm is not able to find any solution at all? How can the algorithm know that there is a better solution out there, than the solution it already found? The answer is it cannot. The observations the agent makes in the environment are dependent on the taken actions. If

the agent chooses the wrong actions it might never be able to explore the environment to a point where it receives more reward. The agent might get stuck with the false impression that it already achieved the maximum reward that is possible.

The last challenge we want to talk about is closely connected to the second one: Keeping the balance between exploration of the environment and exploitation of already learned behavior. Often diverging from already proven behavior decreases the short-term reward, but might lead to a much larger reward in the future. People face this problem in their everyday life: If you know the route from your home to some place, you are way more likely to take that route, than trying to find a shorter one, but risking the eventual failure. In this scenario the person - or in RL the agent - might get stuck in a situation where it never does anything else than taking the best known route, and get forever stuck in this local optimum.

### 4.1.2 Terminology

We already talked a lot about the existing challenges of reinforcement learning. Before we want to dive deeper into the various techniques that were developed to overcome these problems, we want to take a short look on the common terminology used throughout these algorithms.

**States and Observations.** For every environment there is a difference between the internal state $s$ of the environment and the observation $o$ an agent receives. While the state covers all information about the "world" the environment represents, the observation is a representation of that world and may or may not contain all information. In the remainder of this chapter we will often use $s$ instead of $o$, but it should be clear from context, if an observation or the internal state is referenced.

Depending weather we can observe the whole environment or not, we talk about fully or partially observed environments. Observations are typically given to the agent as a real-valued vector.

**Actions.** Actions are the things an agent can do in the environment. Most of the time they are fairly simple, like deciding to go left or right, but they can be more complex.

The set of actions an agent can perform in an environment is usually called the *action space*. Depending on the environment, the action space might be discrete - with only a finite number of available choices - or continuous. Discrete action spaces are typical for game-like environment like CartPole or Breakout. Usually discrete actions are mutually exclusive, meaning only a single action can be chosen at a time: We cannot go left and right - we have to choose.

Continuous action spaces on the other hand require a value to be attached to each available action and are often found in robot control tasks. For example if we look at a self-driving car, we have an action for the degree of the steering wheel and an action for the position of each pedal which need to be set in each step.

Not all RL algorithms are able to handle both types of action spaces. In some instances it is possible to transform continuous action spaces into discrete action spaces if no absolute precision is necessary. However this often results in a very high number of available actions.

**Policies.** Most of the time the words agent and policy will be used in the same manner. For the sake of clarity we want to define the policy as the rule(s) an agent uses to decide which action should be taken for some observation. We can think of the policy as the actors brain.

We denote the policy by $\mu$ if it is deterministic and $\pi$ if it is stochastic. Later when we are focusing on deep RL all of our policies will be build from artificial neural networks which are parameterized by their weights and biases. We therefore extend our notion to include these parameters as $\theta$ to have parameterizable policies $\mu_\theta$ and $\pi_\theta$. If the agent receives an observation at time $t$ it then generates the action according to its policy by:

$$a_t = \mu_\theta(o_t)$$
$$a_t \sim \pi_\theta(o_t)$$

**Episodes and Trajectories.** We define an episode as a single run of an environment. For example in chess an episode is a single game an agent played. While the agent played the episode it generates a *trajectory* $\tau$, which is a sequence of triples from (state, action, reward), describing the episode:

$$\tau = ((s_0, a_0, r_0), (s_1, a_1, r_1))$$

The initial state of the environment is randomly sampled from some start-state distribution $\rho_0$. State transitions $s_t \to s_{t+1}$ happen according to the laws of the environment and the chosen action: $s_{t+1} = f(s_t, a_t)$. An episode might be infinitely long depending on the environment, which also leads to an infinite trajectory. In literature the words episode, trajectory and rollout are often used in a very similar fashion.

### 4.1.3 Reinforcement Learning as Markov Decision Process

In this section we want to take a look at the theoretical foundations of reinforcement learning. We will do this by looking at how we can model every RL problem with a discrete action space as a *Markov Decision Process*. MDPs provide a mathematical framework for decision making processes and were introduced in the 1950s by Richard Bellman [14]. They were named after an earlier model the *Markov Chains* which were created by the mathematician Andrey Markov who studied stochastic processes without memory. In this section we will see how MDPs build the foundations of value-based RL algorithms.

Before we talk about MDPs, let us take a step back and first talk about Markov Chains. In Markov Chains we have a fixed number of *states* and *transition probabilities* which define how likely it is to move from one state to another. We start in a given state and at each step the system randomly evolves from one state $s$ to another state $s'$. This transition only

depends on the states' transition probability and is not related to earlier transitions. This is called the *Markov Assumption*. Figure 4.5 shows an example of a Markov Chain.

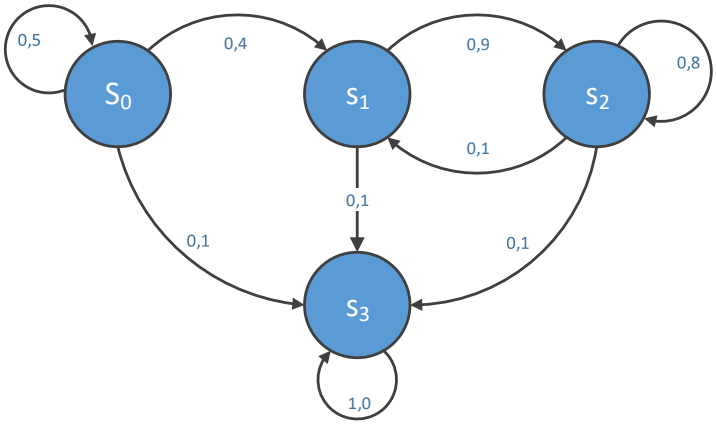



Figure 4.5: An example for a Markov chain. Suppose we start in $s_0$. There is a 50% chance of staying in that state, a 40% chance of transitioning to $s_1$ and a 10% chance of transitioning to $s_3$. No matter how long it alternates between $s_0$, $s_1$ or $s_2$ at some point it will enter $s_3$ and remain there forever, because $s_3$ has a 100% chance of transitioning to itself.

Markov decision processes extend the idea of Markov chains with an active agent. At each step, the agent can choose from a set of actions and the state transition probability depends on the chosen action. Additionally each state transition may return some reward. The agents goal is to find a policy which maximizes the total reward over time. We extended our example from Figure 4.5 with actions and rewards in Figure 4.6. Like for Markov Chains, MDPs transition probabilities must only depend on the current state $s$ and the chosen action.

When modeling a RL problem as MDP we have a problem: The Markov assumption. If we look at the probability of our states, we would normally consider that at step $t$ the next state $s_{t+1}$ is sampled from a probability distribution depending on all the previous states and actions.

$$s_{t+1} \sim P\left(s_{t+1} | (s_0, a_0), (s_1, a_1), \ldots, (s_t, a_t)\right)$$

This sounds reasonable, since the agent might also work with a stochastic policy which picks actions partially at random. Also reaching a state might influence other states or their transition probabilities. This not only violates the Markov assumption, it makes it extremely hard for an agent to approximate that underlying transition function for $P$ to make reasonable decisions in our environment. Therefore we need to simplify our model to:

$$s_{t+1} \sim P(s_{t+1} | s_t, a_t)$$

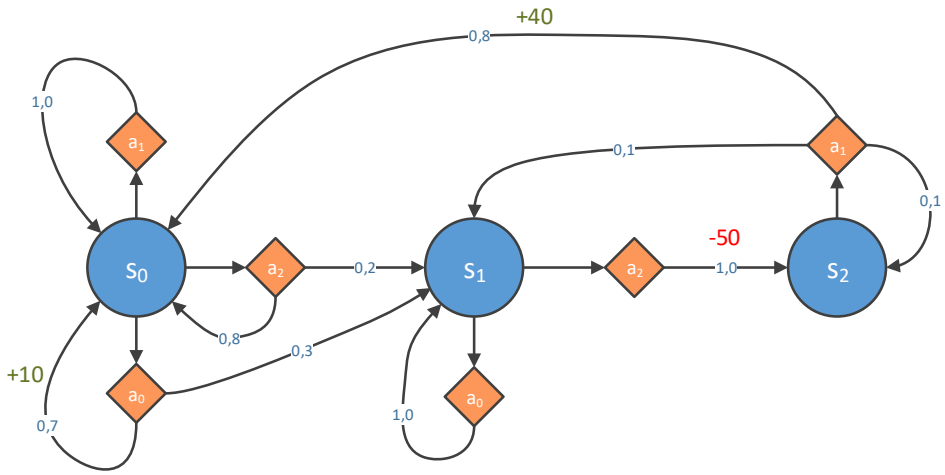



Figure 4.6: An example for the Markov decision process (recreated from [29]). Again we start in $s_0$ and the agent can choose from the actions $a_0$, $a_1$ or $a_2$. Choosing $a_0$ will give it a reward of $+10$. With a probability of 70% the agent will be staying in $s_0$ and with a probability of 30% it will transition to $s_1$. So the agent can likely repeat action $a_0$ in $s_0$ multiple times, but will at some point transition to $s_1$. In $s_1$ it now only has the options to stay there or go through a $-50$ penalty, which might even be repeated. The question is which strategy leads statistically to the most reward? Maybe choosing $a_0$ until we end up in $s_1$ and then never leave $s_1$ by always choosing $a_0$? Or going through the penalty and then collecting positive reward again? We will take a detailed look how we can solve this problem in the introduction of Section 4.2.

This form sacrifices some freedom, but is still powerful enough and leads to much simpler transition functions which can be learned much easier. Despite the fact that a state transition now only can depend on the last state and action, we can look at it from a different angle to reintroduce some of our freedom again. Any state can be defined to include any necessary information and there can be an unlimited (but fixed) number of states in our model. For example imagine a game where you, if you win round one, get a slight bonus for round two. Then the states and their transitions of round two would depend on round one. What we can do is introduce two completely new sets of states for round two where one set has the information attached that we won and one that we lost. This way the states for round two do not depend on round one, we just have more states and transitions. Using this trick helps us model many complex environments as MDP.

Agents in RL only decide upon an observation and not on the internal state of the environment. Also they normally do not know all states or their transition probabilities in the MDP. Therefore we often talk about a *partially observable MDP* (*POMDP*) in the context of reinforcement learning.

**Reward and Return.** In our model, the agent may receive some reward after or during each state transition: $r_t = R(s_t, a_t, s_{t+1})$. The agents goal is to maximize the total sum of expected reward, but - as we discussed earlier - it is complicated to know which action action was responsible for which reward. If we look back at the example from Figure 4.6: How much reward do we generate by choosing $a_0$ in $s_0$? We have a probability of generating $+10$ reward, but we might also get no reward and end up in $s_1$. How much reward we are able to generate once we ended up in $s_1$ is also unclear.

To build a better connection between actions and reward, instead of associating rewards directly with the action which was chosen right before the reward was returned, we calculate the return $R$. The return is defined as a sum of future rewards, which we are able to achieve from the current state when choosing a certain action. Depending on the task, this sum can be formulated in different ways. This way an action which results in negative reward can actually be actually seen as a good action, if it generates much more positive reward in the future and the other way around.

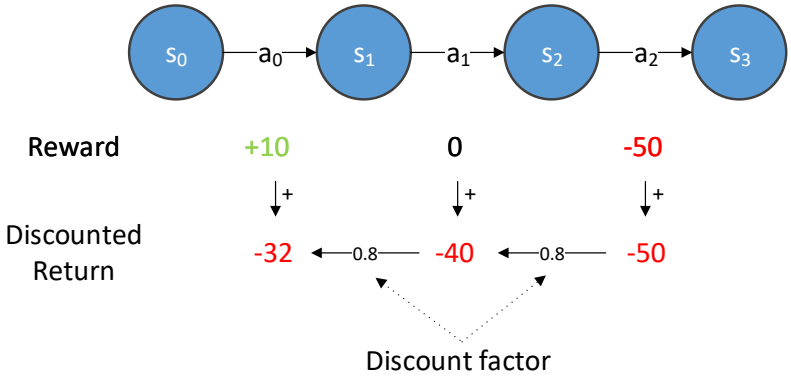



Figure 4.7: Example for discounted reward for a trajectory over four states. The agent will calculate negative reward for all actions, even though the first action produced a positive short-term reward.

Since most of the time, the current action is not directly associated with every future reward, we look at so-called *discounted rewards*. We call this reward the *finite-horizon discounted return* with a discount factor $\gamma \in [0, 1]$:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$$

An example for discounted reward with $\gamma = 0.8$ can be found in Figure 4.7. We can see that action $a_0$ gets associated with negative return, even though the immediate reward is positive. The discount factor is an important variable which controls how much we value future rewards and needs individual tuning for every environment. Small values for $\gamma$ lead to more "shortsighted" actions - with the extreme case $\gamma = 0$ valuing just the initial (next)

reward. The other extreme case with $\gamma = 1$ just means there is no discount and which therefore is called the *finite-horizon undiscounted return*:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

Sometimes we are also dealing with environments with infinite episode lengths. If $T$ is unbounded we talk about an *infinite-horizon discounted return*. In this case $\gamma$ needs to be smaller than 1 to bound our future rewards.

**The RL Problem.** Before we look at the first RL algorithm, we want to give a mathematical definition for the reinforcement learning problem itself. If the goal of the agent is to maximize the total reward from the environment, RL algorithms will select a policy which at each step maximizes the *expected return* by choosing an optimal action $a^*$ under observation $o$. Depending on the chosen discount factor this return will depend more or less on future rewards.

Lets look at an example, where both the environment transitions and the policy are stochastic. The probability for a trajectory with $T$ steps is:

$$P(\tau|\pi) = \rho(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|o_t)$$

We denote the *expected return* for the current policy $\pi$ by $J(\pi)$. For some trajectory $\tau$ this return is equal to:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = E_{\tau\sim\pi}\left[R(\tau)\right]$$

In case of finite-horizon discounted return we get

$$J(\pi) = E_{\tau\sim\pi}\left[R(\tau)\right] = E_{\tau}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

Given this definition for expected return, we can formulate the problem of reinforcement learning as the problem of computing an optimal policy $\pi^*$ which maximizes expected return over a trajectory:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

## 4.2 Value-Based Methods

In this section we want to take a look at the first family of reinforcement learning algorithms: The value based methods. We begin by looking at a technique called *value iteration*, which will allow us to calculate state values in a MDP, in Section 4.2.1. We will then continue to explore how this technique can be used to learn state-action values (called $Q$-values) from a partially unknown MDP in Section 4.2.2, which leads directly to our basic value-based RL

algorithms *SARSA* (Section 4.2.3) and the *Q-Learning algorithm* (Section 4.2.4). Finally we will combine the Q-Learning algorithm with neural networks into Deep Q-Learning in Section 4.2.5 and look at a number of Deep Q-Learning extensions in Section 4.2.6.

### 4.2.1 Value Iteration

In Section 4.1 we already saw that we can express our RL problem as a Markov decision process. Bellman did not only create the model, but also found a way to estimate the optimal state value of any state in an MDP. This state value is calculated by looking at all discounted future rewards an agent can expect if it currently is in state $s$ and then acts optimally for all future transitions. In the Equation 4.1 $T(s, a, s')$ denotes the transition probability for state $s'$ if the agent is in state $s$ and chooses action $a$.

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \tag{4.1}$$

We can see that the equation is recursive, since we calculate the value for the current state by adding the transition reward to the state values of the possible next states weighted by their transition probability. We also discount the value of the next state by our discount factor $\gamma$. Bellman described a simple algorithm called *value iteration* which calculates this optimal state value [14]. It works by initializing all state values to zero and then by iteratively calculating

$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right]$$

Fortunately this algorithm is guaranteed to converge. Depending on how interested we are in exact values, we can always cancel the iteration early and look at the intermediate results in the $k$th iteration. The problem is knowing the current value of the state we are in does not tell us which action we should take next. The equation is not suited to derive a policy from it. Luckily Bellman also found, that it is easy to reformulate the equation to calculate $Q^*(s, a)$ which is the optimal value we can expect if we choose action $a$ and then always act optimally. We define this value in Equation 4.2. This value can again be calculated by value iteration, as can be seen in Equation 4.3:

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{4.2}$$

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_k(s', a') \right] \tag{4.3}$$

With the optimal Q-value at hand, defining the optimal policy is as easy as defining $\pi^*(s) = \arg\max_a Q^*(s, a)$. So are we at the end? Do we have the ultimate tool for every RL problem? Sadly the answer is no, because while we could calculate our policy exactly in that way, we do not have the information to do so. In a real RL scenario we do not know the underlying MDP - we have to discover all states, rewards and transition probabilities first.

### 4.2.2 Temporal Difference Learning

The basic algorithm used for Q-Learning is called *Temporal Difference Learning* (TD Learning). This algorithm is very similar to the value iteration method and also connected to the stochastic gradient decent algorithm. The unknown MDP is usually explored using a stochastic policy. In the easiest case this policy is just sampling a random action from our action space for each state. We then iteratively improve our state estimation by calculating:

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma V_k(s'))$$

In this equation $\alpha$ is a parameter for the learning rate in the interval $[0, 1]$. The term is usually rewritten to emphasize that we change our estimation by having an error term:

$$V_{k+1}(s) \leftarrow V_k(s) + \underbrace{\alpha \underbrace{(r + \gamma V_k(s')}_{\text{TD Target}} - V_k(s))}_{\text{TD Error}}$$

Again the basic TD learning method is formulated for the value function and we have to reformulate it for Q-Value estimation. Also we now have to decide how we explore the environment. Depending on our choices we end up with either *SARSA* or the classical *Q-Learning* algorithm.

### 4.2.3 SARSA

We are finally at the point where we can look at the first reinforcement learning algorithm: SARSA, which was proposed by Rummery and Niranjan in 1994 [69]. SARSA uses the same method or values to sample actions and learn from them, therefore SARSA is a so called *on-policy* learning algorithm. We will talk about the difference to *off-policy* algorithms in a moment when we come to classical Q-learning.

The name SARSA was not originally proposed, but is widely used and comes from the parameters that are needed to make a single update step. The agent which currently is in state $s$ takes action $a$ and gets return $r$. It then samples a new action based on $s'$ from its policy (which is a stochastic policy) and chooses action $a'$, combined we get state-action-reward-state-action $(s, a, r, s', a')$ - the origin of the name.

SARSA uses the TD-learning method to update its Q-value estimation:

$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

For each state-action pair, SARSA calculates a running average of the rewards $r$ combined with the sum of expected discounted future reward. This way SARSA is able to estimate optimal Q-values for every state-action pair given enough iterations. We included a sketch of this procedure in Algorithm 4.2.1.

We just mentioned that actions are sampled from our policy, but what exactly is our policy? To this point we only talked about the calculation of Q-values. If we already

**Algorithm 4.2.1:** SARSA Algorithm for On-Policy TD Q-Value Estimation (adapted from [77])

**Result:** Estimated Q-Values for $Q(s,a)$

Initialize $Q(s,a)$ for $\forall s \in S$ and $\forall a \in A$ randomly. $Q(terminal,\cdot) = 0$
**foreach** *episode* **do**
  $s \sim \rho(\cdot)$ `// Sample initial state`
  Choose $a$ from $s$ using a policy derived from $Q$ (e.g. $\epsilon$-greedy)
  **foreach** *step in episode* **do**
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
    $s \leftarrow s'$, $a \leftarrow a'$
  **end**
**end**

calculated all our Q-values, it seems to be clear that the best policy would be to always choose the action for the current state with the highest Q-value. The problem when learning is we do not know the best Q-value for each state yet. At this point we come back to a problem we discussed in Section 4.1.1: The balance between exploration and exploitation. We need a policy which explores the environment in a way that allows us to calculate Q-values for every state. Therefore we need a policy which will visit each state - not only once multiple times to get sufficient good Q-Value estimations.

If we strictly follow an exploration policy which already exploits our estimated Q-values we will likely end up only exploring a fraction of all states and get stuck in a local optimum. So instead we will use a stochastic policy in SARSA. If we take the other extreme and use a completely random policy the agent will always be able to visit all states eventually, but may need extremely long to do so. Therefore the most commonly used policy is an $\epsilon$-greedy policy, which combines a stochastic exploration with an estimated Q-Value exploitation. At each step we have the probability $\epsilon$ for which we take a random action and with probability $(1-\epsilon)$ we choose the action with the highest Q-value. This way we will explore the environment sufficiently, while also directing learning towards promising states. Often times $\epsilon$ will not be set to a fixed value. Instead $\epsilon$ will be a function over the course of the training process, starting with high values (mostly random actions) and then gradually decreasing over time.

### 4.2.4 Q-Learning

Q-Learning actually predates SARSA, and was developed in 1992 by Chris Watkins [84]. Today, Q-Learning is one of the most popular methods in RL. As for SARSA, in Q-Learning we estimate state-action values with the TD learning algorithm. The difference is that we are now dealing with an off-policy algorithm. This means that the policy that is being

trained is not (or only partially) executed during the training process. You can find the procedure in Algorithm 4.2.2.

**Algorithm 4.2.2:** Basic Q-Learning Algorithm for Off-Policy TD Q-Value Estimation (adapted from [77])

**Result:** Estimated Q-Values for $Q(s,a)$

Initialize $Q(s,a)$ for $\forall s \in S$ and $\forall a \in A$ randomly. $Q(terminal,\cdot) = 0$
**foreach** *episode* **do**
    $s \sim \rho(\cdot)$ `// Sample initial state`
    **foreach** *step in episode* **do**
        Choose $a$ from $s$ using exploration policy (e.g. random or $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
        $s \leftarrow s'$
    **end**
**end**

We can see that while both algorithms are pretty similar, there are a few differences. Instead of using a stochastic policy and using that policy to choose an action $a$ **and** for estimating the future reward (the $Q$ value), we now only use a stochastic policy to sample $a$ and then always use a greedy policy. This can be interpreted in a way that the algorithm considers its estimated Q-Values to be optimal even if they are not. Therefore our update rule becomes:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a))$$

Q-Learning even works if the exploration policy is completely random, even though $\epsilon$-greedy type strategies are often faster. So again, are we finished yet? If Q-Learning is able to calculate optimal Q-Values and thus provides us with an optimal policy we can now always use Q-Learning, right? The problem is while Q-Learning is able to calculate the optimal Q-values, it may need a very long time to do so. And since Q-Learning uses a table for every state that is possible in our environment, problem size exponentially increases for many environments.

If you think of the simple game of Tic-Tac-Toe we have 9 boxes where each player may set his mark, so end up with $2 * 2^9 + 9 = 1033$ possible states. For such a simple game that is a surprisingly high number. And it gets even worse if we look at games which allow even more combinations. If we take the computer game classic *Pac-Man*, we can see every coin the player can capture as a single state. With around 100 coins we are already at $2^{100} \approx 10^{30}$ states. And the coins are not the only thing which may change state in that game. With traditional Q-Learning we would not only need ages to explore all states, we would also end up with a huge table of state-action values which might not fit in our memory.

### 4.2.5 Deep Q-Learning

When we are dealing with a huge number of states, we might want to sacrifice some of the accuracy of our predictions for the sake of actually being able to compute these predictions in a reasonable time. To achieve this instead of computing a table containing a value approximation for every state-action pair we will compute a (nonlinear) function which approximates these values. Ideally this function should have an amount of parameters which is much smaller than the number of possible states. We denote this function as $Q_\theta(s,a)$. As we know from Chapter 3 artificial neural networks are highly nonlinear parameterizable functions. This means we can use neural networks to approximate our Q-values. In 2013, the DeepMind team showed, that using deep neural networks for Q-value approximation has a stellar performance on playing multiple games from the Atari 2600 game console [55]. Their work also proved, that it is not necessary to handcraft features for the neural network. Instead a raw image from the game is sufficient and even results in better performance.

We call a deep neural network that is used for Q-Value estimation a *Deep Q-Network* (DQN) and indicate that our Q-value is estimated by subscripting our Q-function with $\theta$ denoting the parameters of our neural network. Using a DQN for approximate Q-Learning is called *Deep Q-Learning*.

To train a neural network we still need a key element: A loss function. The loss function should represent how much the value calculated by the neural network deviates from the target value. Similar to how we calculated our updates in classical Q-Learning we treat our estimation as if we already knew the best Q-Values. Our target Q-Value then is

$$Q_{target}(s,a) = r + \gamma \cdot \max_{a'} Q_\theta(s',a')$$

Using a common metric like the mean square error we can calculate the difference between the estimated Q-Value of our network and the target Q-Value to get a loss function.

$$\mathcal{L} = (Q(s,a) - Q_{target}(s,a))^2$$

By defining a loss function we can train our neural network in the same fashion as in supervised learning using the well-known stochastic gradient decent algorithm. This will need thousands or even millions of updates to converge - if it ever does. Deep Q-Learning in its basic form suffers from a lot of problems which results in a DQN learning either slowly, not at all, or suffering from a problem called *catastrophic forgetting*. Figure 4.9 shows the reward over time when solving the CartPole environment. As we can see the agent does not learn anything for 300 episodes, then suddenly achieves the maximum possible reward of 200 and then seems to forget what it has learned. This process then alternates back and forth. Problems like this emerge from a number of different problems:

1. The SGD algorithm assumes that we have data that is *independent and identically distributed* (i.i.d), but our training data is neither independent nor identically distributed.

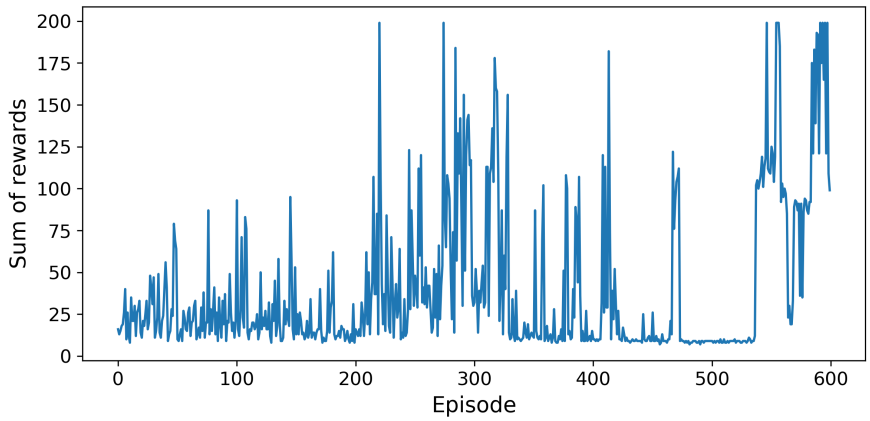



Figure 4.8: Learning Curve for basic Deep Q-Learning only with simple experience replay on the CartPole environment.

Our samples are generated in episodes which typically make state transitions between states which have a lot in common (e.g. two adjacent frames in a game). Also the distribution of our data is not similar to the distribution the data would have if it had been sampled from an optimal policy.

2. We learn from data generated by our current policy (or from a partially random policy in case of $\epsilon$-greedy), which will create samples which are highly correlated. Additionally we use the same model we created our samples with to set our target value and calculate our loss. This may result in a feedback loop which makes training unstable.

3. In training, the agent will likely overestimate Q-Values. This is due to the nature of the update process. In each step, the target model will select the highest Q-Value, but all Q-Values are estimated. Think of a situation in which each action for a given state has the same Q-Value. An approximation will never be exact, so some of the Q-Values will be slightly higher and some will be slightly lower. When selecting the next Q-Value we do not go for an average we select the highest. Thus we are always prone to overestimation.

All these problems combined lead to a very unstable learning process. Fortunately over the years many extensions were developed to reduce their impact dramatically.

### 4.2.6 DQN Extensions

In this section we want to give a brief overview over techniques that were developed to make Deep Q-Learning more stable.

**Experience Replay.** At first we want to focus on breaking the correlation between the samples. If we always train from a single episode all our training data will be highly correlated and states will look almost identical. To overcome this instead of learning directly from the trajectory generated by a single episode we introduce a *replay buffer*. After playing an episode we draw random samples from our buffer and learn from them. This will break up the correlation between samples and also reduces the danger of forgetting already learned behavior. On the negative side it introduces new hyperparameters with the size of the replay buffer and the number of samples we want to draw for learning, which are both dependent on the task we are trying to solve.

**Prioritized Experience Replay.** Lets take the idea of the replay buffer to the next level: Since not all of our samples might be equally important for the learning process, it would be a good idea to replay interesting samples more often than others. This procedure is called *importance sampling* (IS) or *prioritized experience replay* (PIR) and was developed in 2015 by the DeepMind Team [70]. The question is what are interesting samples? Schaul et al. proposed to use the TD error term $\delta = r + \gamma V_k(s') - V_k(s)$ (see Section 4.2.2) as an indication of a sample being interesting. This makes sense if you think about what a large error means for our algorithm: If we estimated a state value wrong, it is equal to saying we are surprised by the result of the state value. Thus the sample is interesting.

In our replay buffer we now assign a priority to each sample, depending on how large the TD error is. We set this priority to a high value for the first time, to ensure that the sample will be sampled at least one time and update the sample priority each time it is sampled to $p = |\delta| + \epsilon$ with $\epsilon$ being a small positive constant to ensure each sample has a non-zero probability of being drawn. A single sample then has the probability of

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

The exponent $\alpha \in [0, 1]$ is a new hyperparameter which defines how much weight is given to importance sampling with $\alpha = 0$ being standard uniform experience replay.

With some samples being sampled more often than others we again violate our goal of learning uniformly on samples from the target distribution that would be observed if the agent would behave optimally. Without a compensation for this bias, our agent would overfit on those high priority samples. To compensate, we downweight more important samples in training by

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

where $\beta \in [0, 1]$ is another hyperparameter which controls how much we want to compensate (0 means no compensation). PIR has shown to vastly improve convergence speed and overall performance of Deep Q-Learning, but also introduces two additional hyperparameters which need to be tuned.

**Target Networks.** The next extension aims at improving training stability and reducing the problem of a feedback loop. Since we use the same network to generate samples and to calculate the target Q-Values, situations where training is unstable, diverges or oscillates are very likely. This can be broken up by separating the prediction when generating samples from the calculation of the target values. This is done by working on two similar - but not identical - copies of the same network. We call one of them the *online model* and one the *target model*. The online model will interact with the environment and get updated in the regular update step, while the target model only makes the predictions for the target Q-Value. Every once in a while we copy the weights of the online model to the target model.

**Double DQN.** Next up we want to reduce Q-Value overestimation. This can be done by improving the usage of our target network: Instead of always selecting the action with the highest Q-Value for the Q-target in the update step, we select the action with the highest Q-Value from the online model, and then use the Q-Value estimation for that action from the target DQN. This method was also developed by DeepMind and is called *Double DQN* (DDQN) [79].

**Dueling DQN.** Another technique which aims at stabilizing learning is called Duelling DQN [83] (not to be confused with Double DQN). Duelling DQN changes the architecture of the DQN to not compute Q-Values directly, but rather compute two values: The state value $V(s)$ and the so-called *advantage* $A(s,a)$. The advantage expresses how good it is to take an action $a$ in state $s$ in comparison with every other action. If we look at the advantage for an optimal policy $A^*(s,a)$, the best action $a^*$ will have an advantage of 0. We can express the Q-Value as $Q(s,a) = V(s) + A(s,a)$. Computing multiple values which are connected to the same task proves to be beneficial for the training of neural networks and greatly stabilizes training. Therefore we extend our neural network to compute both $V(s)$ and $A(s,a)$ for every action. [21, 23]

**Even more extensions.** We already discussed a lot of important extensions for Deep Q-Learning, but there are even more. Update steps can be combined into multi-step learning which further stabilizes the training process [76]. Instead of learning to approximate the returns we can also learn to approximate the distribution of returns [13]. Like for every neural network the introduction of noise often results in more robust predictions. Additionally the introduction of noise is beneficial in RL because it improves exploration [26, 62].

Most of the techniques to improve Deep Q-Learning that were discussed can also be used in conjunction. In 2018 the DeepMind team created an agent which they called *Rainbow* and which was able to outperform all standalone techniques on the Atari 2600 benchmark [31]. Even though this proves that the combination of these extensions can be beneficial, the introduction of several new hyperparameters requires careful fine-tuning which might take a lot of time even with modern hyperparametersearch algorithms.

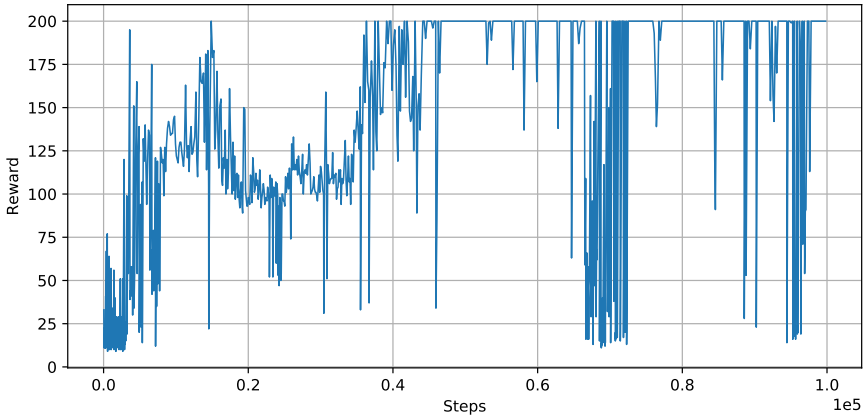



Figure 4.9: Learning Curve for Deep Q-Learning with Double DQN, Duelling DQN and prioritized replay on the CartPole environment over 100k steps ($\sim$ 600 episodes). Learning is much more stable than before, but still suffers from brief performance collapses. Learning can be further stabilized by optimizing the used hyperparameters, especially the learning rate and the size of the replay buffer.

## 4.3 Policy Gradient Methods

We already saw, that Q-Learning (with its extensions) is a very powerful tool to solve RL problems. Nevertheless we also saw that Q-Learning suffers from a lot of problems and with all its extensions is hard to parameterize correctly. Also we are not able to solve RL problems with continuous action spaces, since there would be an infinite number of actions to choose from. Therefore we need an alternative way to deal with reinforcement learning problems.

Let us look at what we are doing in Q-Learning: We estimate values $Q_\theta(s,a)$ and then derive an optimal policy by $\pi^*(s) \approx \arg\max_a Q_\theta(s,a)$. So we are only indirectly learning an optimal policy. While the Q-Value might be interesting, at the end we do not need to know how much better an action is compared to others. We are only interested in acting optimally. So instead of estimating Q-Values and then deriving an optimal policy, we want to directly learn the optimal policy. This is the idea behind *Policy Gradient* (PG) methods which aim at iteratively computing an optimal policy by using gradients in policy space.

We will only look into a single member of this family in Section 4.3.1: The REINFORCE algorithm. This is because the PG-methods quickly involved into combined methods which take ideas from both Q-Learning and REINFORCE and combine them to achieve even better results.

### 4.3.1 REINFORCE

The REINFORCE algorithm was invented in 1992 by Ronald Williams [85]. The algorithm computes a policy which produces action probabilities directly from states. Like in Deep

Q-Learning we use a neural network as our policy, which directly outputs probabilities for actions in the discrete case and values for actions in a continuous setting. The name REINFORCE originates from the simple idea of positively reinforcing actions which proved to yield good results while discouraging actions which yield bad results. This reinforcement is done iteratively in small steps by application of policy gradients.

So let us begin by defining what a policy gradient should be. Remember our definition for expected return $J(\pi)$ from Section 4.1.3. We extend our notion to also include the parameters of our neural network. We can then define our goal as finding the parameters for our network which maximize the expected return:

$$\max_{\theta} J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[R(\tau)\right] \tag{4.4}$$

So how do we maximize the expected return and find the correct parameters? If we look at our objective $J(\pi_\theta)$ we can think of it as an abstract hypersurface for which we are trying to find a maximum. The dimensions of the surface are given by the network parameters $\theta$ and the "height" is our return. To gradually improve our policy, we are using gradient ascent (which is the same as gradient descent, but now we are using the gradients to find a maximum) for $J(\pi_\theta)$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$$

The gradient $\nabla_\theta J(\pi_\theta)$ is our *policy gradient* and $\alpha$ is our learning rate. But at this point we have a problem: If we look at Equation 4.4 we can see, that $\nabla_\theta J(\pi_\theta)$ is not differentiable with respect to $\theta$ because $R(\tau)$ is not dependent on $\theta$. Also the function $R(\tau)$ itself is unknown. Since the expected reward is directly dependent on the decisions of the current policy, it is possible to rewrite the equation into a form that is differentiable:

$$\nabla_\theta J(\pi_\theta) = E_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{4.5}$$

We omit the proof here, but it can be found in [42]. Now that we have a gradient we can optimize, we can take a look at the final REINFORCE algorithm shown in Algorithm 4.3.1.

After initializing the policy, we sample trajectories $\tau$ from the *current* policy. These trajectories are then used to *estimate* $\nabla_\theta J(\pi_\theta)$. The estimations are calculated by collecting individual gradients for each sample of the current episode and calculating the gradient according to Equation 4.5 in Line 7. Note that this estimation of $\nabla_\theta J(\pi_\theta)$ may not be exact and may lead to problems in the training process.

Once we have estimated the gradients, they are applied in a single gradient ascent step. This is usually done by an advanced optimizer like Adam (see Section 3.2.5), but we always use vanilla gradient ascent for clarity. Because we estimate the gradients for our policy, by sampling data from our policy, REINFORCE is an on-policy algorithm. This also means that we have to discard our samples after each update - they cannot be used to estimate gradients for the next update step. Because this is true for all policy-based methods, they

**Algorithm 4.3.1:** The REINFORCE Algorithm (adapted from [42])

**Data:** Learning Rate $\alpha$
**Result:** Optimal policy parameters $\theta$

Initialize weights $\theta$ of policy network $\pi_\theta$ randomly.
**foreach** *episode* **do**
  Sample trajectory $\tau = ((s_0, a_0, r_0), \ldots, (s_T, a_T, r_T))$ with $\pi_\theta$ from the environment
  $\nabla_\theta J(\pi_\theta) \leftarrow 0$
  **for** $t = 0, \ldots, T$ **do**
    $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$
    $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$
  **end**
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta)$
**end**

are usually called sample inefficient. This means that algorithms which estimate policy gradients need a high number of environment interactions for their training process, which could be a problem if the environment is computationally expensive or we are dealing with real world interactions.

Note that in REINFORCE we do not need a dedicated exploration policy. Instead, exploration is now build into our network, which directly outputs action probabilities for a given state. We then use these probabilities to sample an action according to the given probability distribution. In comparison with Q-Value estimation with all its extensions, we now have a much simpler way of computing our optimal policy. There is no need for a replay buffer or a target network copy. We also extended our possibilities into continuous actions and are now able to model stochastic environments. Unfortunately there are also downsides to policy gradient methods: Because we are dealing with relatively small sample estimations, our estimations may be unstable and cause a lot of variance in our gradients. Samples from a single trajectory are also highly correlated. Because the exploration now completely depends on the current agent, policy-based learning is prone to getting stuck in local optima. We will look at ideas of how to overcome all these problems in Section 4.3.2 and Section 4.4.

### 4.3.2 REINFORCE Improvements

We already talked a little bit about problems with the REINFORCE algorithm and policy gradient methods in general in Section 4.3.1. We now want to take a detailed look at how we can improve policy gradient methods and overcome or at least reduce the impact of these problems.

**Learning from Steps.** Let us go back and take another look at Algorithm 4.3.1. The algorithm is written in a way that we need to complete a full episode, before we get into training. This might become a problem if episodes are really long or even never end at all. With long episodes it will take a very long time before we can actually learn anything and we have to discard all that data immediately after. Therefore it would be beneficial, if it was possible to learn from a single or just a few steps of environment interaction.

Currently we are computing our gradient with a sample estimate over the entire episode. If you look at Equation 4.5 we can actually substitute $R_t(\tau)$ by $Q_\theta(s_t, a_t)$. This means we do not actually need a full episode to train our network, we only need an estimate for the expected return according to our policy. The problem is unlike with DQN we now only have a network which can tell us the optimal action. We do not know the Q-Values anymore. There are two possible solutions for this problem:

1. **Unrolling.** First let us reformulate the equation again and substitute $R_t(\tau)$ with $r_t + \gamma V_\theta(s')$ (which is equal to $Q_\theta(s_t, a_t)$). Since $V_\theta(s)$ is recursive, we can see, that future state values get exponentially discounted by $\gamma$. Since the impact of future state values on the current state value gets exponentially smaller, we can ignore recursions after a certain depth depending on how small we set $\gamma$. This means we can approximate $V_\theta(s)$ by unrolling for a fixed number of steps $N$ in the future. This way we only need a trajectory which is at least $N + 1$ steps long to calculate an update for the first step.

2. **Value Networks.** We can extend our neural network to also output the state value for the current state. This is done in most of the combined algorithms we will look at in Section 4.4.

**High Gradient Variance.** If we sample a trajectory from the environment using our current policy, the return we get for that episode may include a significant amount of variance. This variance comes from multiple factors. Since we start in a randomly sampled state, choose our actions at random from the action probability distribution of our policy and (in some environments) need to deal with stochastic state transitions, two adjacent episodes might result in a totally different reward. Take a card game for example where we get an initial random set of cards drawn to our hand. If the cards are good, we are way more likely to win the round and get much reward, but that does not necessarily tell us, that we did choose good actions, we just were lucky. Also depending on the environment it is not clear what a reward actually means. If rewards are always positive our gradients are always positive, but not all actions are actually good and should be reinforced. To counter this problem we need to calculate the reward depending on the value of the current state and the next state. We therefore introduce an action independent *baseline* into our gradient estimation, which is given by

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} (R_t(\tau) - b(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

There are multiple options on how to estimate the baseline. The easiest method is to use an average over the current trajectory

$$b = \frac{1}{T} \sum_{t=0}^{T} R_t(\tau)$$

However this baseline does not take the value of the current state $s_t$ into account, it just centers the returns around zero. This encourages half of the actions and discourages the other half. A better method is the introduction of a value function for the network which actually estimates $V(s)$. We will discuss this in Section 4.4.

**Sample Correlation.** Like for Deep Q-Learning we have the problem of training on samples from a single trajectory. These samples are highly correlated in most environments and thus not optimal to learn from. We previously solved this with the introduction of a replay buffer, but this time we cannot learn from old data. On the other hand playing multiple episodes in an environment can take a very long time. Therefore the solution is to create several copies of the same environment but with randomly sampled initial states. This way we can generate only a few samples from each environment and obtain less correlated data for training.

**Exploration.** Contrary to DQN, exploration is part of the network structure in PG methods, because we directly sample actions from probability distribution given by the network. This integrated exploration has many benefits as we do not need to control the exploration with hyperparameters, but it does not prevent our algorithm from getting stuck in a local optimum. Therefore we extend our optimization problem to include entropy, to encourage the agent to not be too certain about its actions. The entropy for a given state $s_t$ is then given by

$$H_t(\pi) = -\sum_a \pi(a|s_t) \log \pi(a|s_t)$$

The entropy for any state will always be positive, with a minimum of zero, if a single action has a probability of 1. Therefore we have a minimal entropy if our policy is very certain and a maximal entropy if all actions have uniform probability. We extend our optimization to include this term and maximize entropy (to a certain degree) counteracting an agent which is too certain about its actions and thus encouraging exploration.

## 4.4 Combined Algorithms

We already explored two different ideas of how to solve RL problems. In Section 4.3.2 we saw, that it makes sense to combine value estimation - the idea behind Q-Learning - with policy gradient learning. In this Section we want to explore a well known family of RL algorithm which combine these two ideas into the so-called *actor-critic* algorithm. We will

give an introduction into the actor-critic model in Section 4.4.1 and then continue to more advanced actor-critic methods like PPO in Section 4.4.2, as well as its two competitors ACER and ACKTR in Section 4.4.3. We will also present an extension which modifies the reward function to include intrinisic reward which has proven to greatly improve learning for actor-critic variants in complicated, exploration-focused environments in Section 4.4.4.

### 4.4.1 Actor-Critic

In this Section we want to introduce the RL algorithm family of *Actor-Critic* methods, which combine policy gradient and value estimation techniques. We will also present the popular *Advantage Actor-Critic* (A2C) algorithm which builds the foundation for many other recent developments.

As we can suggest from its name an agent which learns with the actor-critic model consists out of two parts:

- **The Actor.** The *actor* represents a policy which takes an observation and outputs a probability distribution for the actions (it "acts" in the environment). The actor is trained using policy gradients, similarly to an agent trained with REINFORCE.
- **The Critic.** The *critic* is a state-value estimator. Depending on the actor-critic variant different state values are calculated. For now, the critic calculates the state value $V(s)$, but we will look at critics which calculate other values like the advantage $A(s, a)$ later on. The critic represents the component from the value-based family and stabilizes the training process.

We illustrated an example for the actor-critic network architecture in Figure 4.10. While we could build the actor and the critic separately with two different networks, they will usually be constructed with a single network with separated network heads. The network then shares a certain amount of layers which learn basic feature representations. For example if we are dealing with images as input and are using a CNN to process them, the convolutional layers should be shared between the actor and the critic. This improves efficiency as we do not need to learn the feature representation twice. Since both the actor and the critic are propagating their error through the upper layers, they can be trained faster and unstable learning is reduced. After the shared layers, the actor and the critic have their own dedicated network output, which can consist out of a single layer or can be a complex network on its own.

The vanilla actor-critic method can be trained in a similar fashion as we did with REINFORCE and is shown in Algorithm 4.4.1. We changed a few things from REINFORCE, so lets quickly go through them:

- We changed our notion for the application of the gradients. While we directly collected the gradients in Algorithm 4.3.1, we now calculate a loss. This makes it easier to add additional goals to the optimization (e.g. entropy). The loss is later converted into a gradient and applied by the SGD optimizer. Because we are now

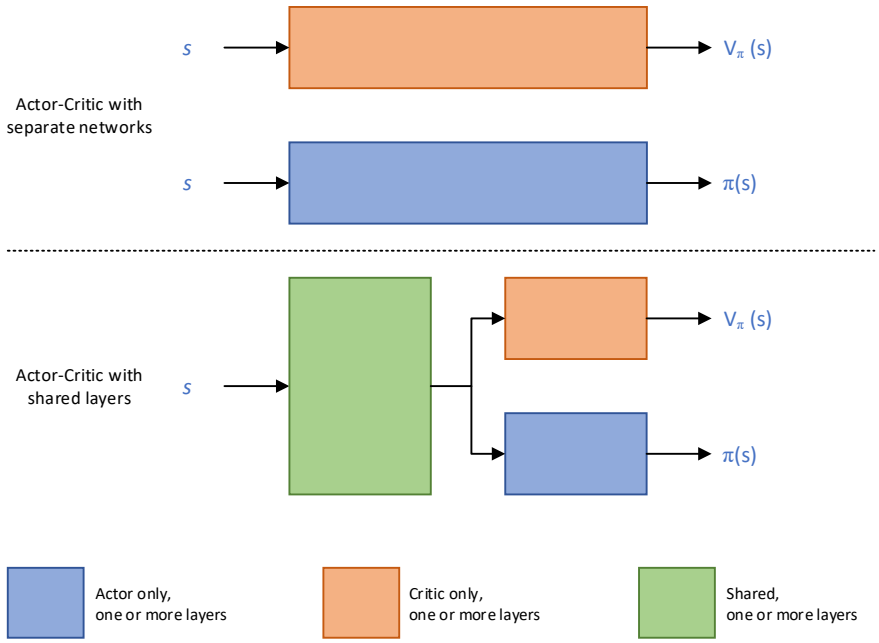



Figure 4.10: Network architecture variants for actor-critic algorithms. The networks might be separate or with shared layers. (Adapted from [42])

dealing with loss and the goal is to minimize the loss we have to change sign. By minimizing the loss we are performing gradient ascent on our objective function.

- We introduce integrated exploration and regularization with the addition of the entropy $H$ to the loss as we discussed in Section 4.3.2. The entropy is added to the loss for the actor and can be scaled with a new hyperparameter $\beta$.
- We are still using a for-loop over episodes, but it is possible to substitute this loop with a loop over an arbitrary number of individual steps (see Algorithm 4.4.3).

**Estimating Advantage.** The vanilla actor-critic algorithm is usually not used on its own. Instead a first extension is used: The *Advantage Actor-Critic* (A2C) algorithm. When we talked about DQN Extensions in Section 4.2.6 we briefly mentioned the advantage function $A(s, a)$. The advantage is a measure of how good an action is compared to all other actions. For DQN we compared the action only to other actions given that we know the optimal policy. This resulted in a maximal advantage of zero for the best action and a negative advantage for all other actions. This time we are instead looking at $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. This is the advantage which measures the change in return if we take action $a$ and then always act according to our current policy. In other words, the advantage provides us with a relative measure of how good an action is compared to the average action our policy would choose. Before we look at how to estimate the advantage, let us first discuss why we

**Algorithm 4.4.1:** The vanilla actor-critic algorithm with entropy regularization.

**Data:** Actor learning rate $\alpha_A$, Critic Learning Rate $\alpha_C$ Entropy coefficient $\beta$
**Result:** Optimized parameters for the actor $\theta_A$ and critic $\theta_C$

Initialize all the weights $\theta$ randomly
**foreach** *episode* **do**
  Sample trajectory $\tau = ((s_0, a_0, r_0), \ldots, (s_T, a_T, r_T))$ with $\pi_\theta$ from the environment
  **for** $t = 0, \ldots, T$ **do**
    $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$
    Calculate $V_{\theta_C}(s_t)$ using the value network
    Calculate Entropy $H_t = -\sum_a \pi_{\theta_A}(a|s_t) \log \pi_{\theta_A}(a|s_t)$
  **end**
  /* Calculate loss */
  $\mathcal{L}_{val}(\theta_C) = \frac{1}{T} \sum_{t=0}^{T} (V_{\theta_C}(s_t) - R_t(\tau))^2$ // Value loss using MSE
  $\mathcal{L}_{pol}(\theta_A) = \frac{1}{T} \sum_{t=0}^{T} ((V_{\theta_C}(s_t) - R_t(\tau)) \log \pi_{\theta_A}(a_t|s_t) - \beta H_t)$ // Policy loss
  /* Update parameters with SGD optimizer */
  $\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C} \mathcal{L}_{val}(\theta_C)$ // Update Critic
  $\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A} \mathcal{L}_{pol}(\theta_A)$ // Update Actor
**end**

should even care about advantage. Why do we need advantage when we have state-values?

Remember that we are currently training our policy by computing gradients based on the return we get by following the current policy in the environment. Our goal is to reinforce good actions and discourage bad actions. Since we compute our gradients from the return, we need the return to be negative for bad and positive for good actions to actually compute gradients which reflect reinforcement or discouragement of actions. Unfortunately this is not what usually happens: Many environments do not provide negative but only positive reward. So all actions are actually reinforced. To change this we need a relative measure which tells us how good an action is compared to all other actions and this is exactly what the advantage function does.

To compute the advantage we need two things: $V^\pi(s)$ and $Q^\pi(s, a)$. Currently our network only computes the value function, so should we further extend it to also compute the Q-Values? While this would be possible, it actually adds overhead into the network and Q-Values are harder to estimate than state-values, since Q-Value functions are more complex. Also learning Q-Values in continuous environments produces additional problems. So instead we want to estimate the Q-Values from the already estimated state-values. If we assume, that our estimate $V^\pi(s)$ is sufficiently accurate, we can estimate the Q-Value by:

$$
\begin{aligned}
Q^{\pi}(s_t, a_t) &= E_{\tau\sim\pi}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n}\right] + \gamma^{n+1} V^*(s_{t+n+1}) \\
&\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n} + \gamma^{n+1} V^{\pi}(s_{t+n+1})
\end{aligned}
$$

$$
\begin{aligned}
A^{\pi}(s_t, a_t) &= Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \\
&\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n} + \gamma^{n+1} V^{\pi}(s_{t+n+1}) - V^{\pi}(s_t)
\end{aligned}
$$

Using this $n$-step unrolling technique, we can estimate the Q-value with $n$ as a tradeoff parameter between variance and bias. Using more steps reduces bias, since $V^{\pi}(s)$ may be inaccurate, but at the same time it induces variance, since all rewards are sampled from a single trajectory. The idea is that variance increases the further we look into the future starting at step $t$ while the bias decreases. This means we are combining both to get the best result. Unfortunately deciding for an $n$ is not an easy choice and would require careful tuning. Therefore unrolling is done with a similar technique called *Generalized Advantage Estimation* (GAE) which was proposed by Schulman et al. in 2015 [72]. Instead of using a fixed $n$ we instead compute the advantage using a weighted average over $n$-step advantages with $n = 1, 2, 3, \ldots, k$. This technique further reduces the influence of variance and bias, while also making the choice of a hyperparameter easier. GAE can be written as:

$$
\begin{aligned}
A^{\pi}(s_t, a_t) &= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \\
&\text{with } \delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)
\end{aligned}
$$

While GAE still introduces a new hyperparameter in the form of $\lambda$ which controls how much we weight estimated advantages with further unrolling, this parameter does not introduce a hard boundary and therefore is much easier to tune.

Algorithm 4.4.2 shows an updated version of the original actor-critic algorithm with GAE. Note that in praxis $V_{tar}$ is often calculated as $V_{tar} = A_{GAE}(s_t, a_t) + V^{\pi}(s_t)$ when using advantage estimation to avoid additional computation.

**Parallel environment interactions.** We already developed techniques to solve most of the problems shown in Section 4.3.2, but there is still one remaining problem: Sample correlation. Since we cannot introduce a replay buffer - we are dealing with an on-policy algorithm - we have to generate a lot of training data to reduce sample correlation and variance before each update. Sampling all that data from a single environment running in a single process takes a lot of time and since episodes may be significantly dependent on the random initial state, we would need the agent to play through several whole episodes to gather good training data. In 2016 Mnih et al. showed, that it is possible to train an A2C agent in parallel on multiple environments with their *Asynchronous Advantage Actor-Critic* (A3C) algorithm [54].

**Algorithm 4.4.2:** The Advantage Actor-Critic (A2C) algorithm with entropy regularization.

**Data:** Actor learning rate $\alpha_A$, Critic Learning Rate $\alpha_C$ Entropy coefficient $\beta$
**Result:** Optimized parameters for the actor $\theta_A$ and critic $\theta_C$

Initialize all the weights $\theta$ randomly
**foreach** *episode* **do**
  Sample trajectory $\tau = ((s_0, a_0, r_0), \ldots, (s_T, a_T, r_T))$ with $\pi_\theta$ from the environment
  **for** $t = 0, \ldots, T$ **do**
    Calculate $V_{\theta_C}(s_t)$ using the value network
    Calculate the advantage $A^\pi(s_t, a_t)$ using GAE.
    Calculate $V_{tar}(s_t)$ using the advantage.
    Calculate Entropy $H_t = -\sum_a \pi_{\theta_A}(a|s_t) \log \pi_{\theta_A}(a|s_t)$
  **end**
  /* Calculate loss */
  $\mathcal{L}_{val}(\theta_C) = \frac{1}{T}\sum_{t=0}^{T}(V_{\theta_C}(s_t) - V_{tar}(s_t))^2$ // Value loss using MSE
  $\mathcal{L}_{pol}(\theta_A) = \frac{1}{T}\sum_{t=0}^{T}(-A^\pi(s_t, a_t)\log \pi_{\theta_A}(a_t|s_t) - \beta H_t)$ // Policy loss
  /* Update parameters with SGD optimizer */
  $\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C}\mathcal{L}_{val}(\theta_C)$ // Update Critic
  $\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A}\mathcal{L}_{pol}(\theta_A)$ // Update Actor
**end**

When training the agent in parallel on multiple environments we want to create multiple copies of the same environment (and our agent) which gather trajectories independently. Further, we want to initialize them with a different random seed to generate different trajectories due to randomness in environment transitions or agent action choices. The workers then continuously gather the training data from agent-environment interactions. This data will be used to calculate updates for a global network, which will then be pushed back to the workers periodically. This procedure can be done in two different ways: Synchronous and Asynchronous. Though A3C only uses asynchronous parallelization we will also briefly discuss synchronous parallelization as well.

- **Synchronous Parallelization.** In synchronous parallelization all workers are working with the same copy of the network. Workers will gather a trajectories of a certain length and then wait for all other workers to finish. The trajectories are then combined and used to update the global network. After the global network has updated, each worker pulls the updated parameters to its own network copy and then continuous to collect new trajectories. This type of parallelization is often used for A2C. The use of dedicated network copies can be avoided, if training is only done on a single machine.
- **Asynchronous Parallelization.** In asynchronous parallelization workers also collect

trajectories, but they additionally compute the gradients and then directly push the gradients to the global network. The workers do not wait for each other and thus may work with a slightly outdated network copy.

Both parallelization methods have their pros and cons. While synchronous parallelization works well on a single machine, the waiting mechanism may significantly slow down a distributed training process. On the other hand with non-blocking asynchronous parallelization, the computation of gradients on slightly outdated network copies makes training less stable. Nevertheless, both methods work well and vastly reduce the impact of sample correlation and variance for on-policy PG methods.

### 4.4.2 Proximal Policy Optimization

The A2C algorithm addresses every problem we talked about in Section 4.3.2. Still if we look at the learning curve in Figure 4.11 which shows the training progress on the simple environment CartPole, we can see that learning tends to still be unstable and even suffers from *performance collapse*. How is that possible? To understand the origin of this problem, we need to look back at our basic procedure of how we are improving the policy using policy gradients.

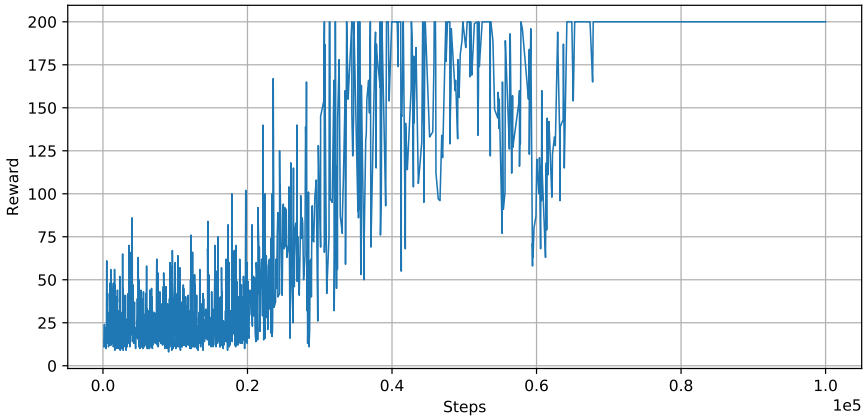



Figure 4.11: Reward per episode for an agent trained with the A2C algorithm on the CartPole environment over $100k$ training steps (roughly 600 episodes). Notice the brief performance collapse around 600k training steps.

In PG algorithms, we use a policy $\pi_\theta$ which is parameterized by the weights of our neural network. This policy is used to generate a trajectory in our environment from which we compute the policy gradient $\nabla_\theta J(\pi_\theta)$. During optimization we iteratively generate a sequence of policies $\pi_1, \pi_2, \ldots, \pi_n$ from the the policy space $\Pi$ (the set of all policies). While in each iteration we aim at improving the policy, we are not actually directly searching for a policy in policy space. Instead we are searching the policy in the parameter space $\Theta = \{\theta \in \mathbb{R}^m\}$ with $m$ being the number of parameters in our network. Unfortunately

distances in policy space and in parameter space are not the same with respect to $\theta$. If we take two pairs of parameters $(\theta_1, \theta_2)$ and $(\theta_2, \theta_3)$ with the same distance in the parameter space, their mapped policies in policy space might not have the same distance:

$$d(\theta_1, \theta_2) = d(\theta_2, \theta_3) \not\Rightarrow d(\pi_{\theta_1}, \pi_{\theta_2}) = d(\pi_{\theta_2}, \pi_{\theta_3})$$

The hypersurface on which we are performing our gradient ascent is not equal to the hypersurface we actually care about. Simply adjusting the learning rate $\alpha$ to a small value does not fix this problem. A small learning rate will only slow down learning and raise the probability of getting stuck in a local maximum. On the other hand we cannot accept occasional learning collapses, since learning collapses mean that we will immediately and dramatically reduce the quality of the training data. Recovering from a performance collapse can therefore take a long time. So how can we prevent them, while still learning in parameter space? The answer to this are *trust regions*. The idea is to estimate the change in policy space by looking at the difference of the probability distributions between the old and new policy. We then define a maximal distance "around" our old policy in policy space - called trust region - which is the maximum distance the new policy is allowed to deviate from the old policy. Moreover we want to guarantee, that the performance of the new policy is always better than the performance of the old policy.

In 2015 Schulman et al. were able to provide mathematical foundations which made the computation of trust regions and *guaranteed monotonic improvement* of the policy possible. They also proposed an algorithm called *Trust Region Policy Optimization* (TRPO) [71]. Unfortunately even though their algorithm works, it is hard to implement and computationally very expensive, since it relies on second order derivates. To overcome these problems, they extended their work in 2017, proposing a simplified algorithm called *Proximal Policy Optimization* (PPO) which even outperforms TRPO in many environments [73]. PPO is computationally as expensive as vanilla actor-critic variants and thus PPO is one of the most used reinforcement learning algorithms today. In the remainder of this Section, we want to take a look at how PPO achieves its goal of monotonic policy improvement, while staying computationally inexpensive.

**Monotonic Improvement.** Instead of trying to directly maximize the expected return $J(\pi)$, we can instead maximize a different objective. To achieve monotonic improvement, we formulate a new objective based on the *relative policy performance identity*. If we have some policy $\pi$ and update that policy to get $\pi'$, we define their relative policy performance identity as

$$J(\pi') - J(\pi) = E_{\tau \sim \pi'} \left[ \sum_{t \geq 0} A^{\pi}(s_t, a_t) \right] \tag{4.6}$$

which is their difference in expected return. With this objective our optimization problem becomes $\max'_{\pi}(J(\pi') - J(\pi))$ which ensures, that $J(\pi') - J(\pi) \geq 0$. The worst case is not changing the policy, such that $\pi' = \pi$.

The problem with Equation 4.6 is that it uses an expectation over a trajectory sampled from $\pi'$ - which we obviously do not have access to at that point. We therefore need a way to estimate trajectories from $\pi'$. If we assume that two successive policies $\pi$ and $\pi'$ are relatively close, we can also assume that the resulting state distributions we observe when interacting with the environment are also similar. Therefore we can use a method known as *importance sampling* which estimates an unknown distribution from data sampled from a known distribution. This is done by computing weights which increase rewards which are more likely under $\pi'$ and decrease rewards which are less likely. These weights are calculated by the ratio of action probabilities between $\pi$ and $\pi'$: $\frac{\pi'(a_t,s_t)}{\pi(a_t,s_t)}$. We then can estimate the relative policy performance identity by

$$J(\pi') - J(\pi) = E_{\tau\sim\pi}\left[\sum_{t\geq 0} A^{\pi}(s_t,a_t)\frac{\pi'(a_t,s_t)}{\pi(a_t,s_t)}\right] = J_{\pi}^{CPI}(\pi') \tag{4.7}$$

thus being only dependent on samples generated by the current policy. The new objective with the name *conservative policy iteration* (CPI) $J_{\pi}^{CPI}(\pi')$ is also called the *surrogate objective*, since it indirectly optimizes the original objective. It can be proven that the gradient with respect to $\theta$ of the surrogate objective is equal to the gradient of the original objective. We omit the proof here, but it can be found in [42].

If the gradient of the original and the surrogate objective is the same, one could ask the question why we even need the new objective. Also we only approximate $J(\pi') - J(\pi)$, so how do we guarantee that $J(\pi') - J(\pi) \geq 0$? The reason behind our new objective is that it allows us to bound the error. Achim et al. [2] showed that - using the surrogate objective - we can bound the error in terms of Kullback-Leibler (KL) divergence between two successive policies to

$$|(J(\pi') - J(\pi)) - J_{\pi}^{CPI}(\pi')| \leq C \cdot \sqrt{E_t\left[KL\left(\pi'(a_t,s_t)||\pi(a_t,s_t)\right)\right]}$$

From this, we can easily derive a form which allows us to determine if $J(\pi') - J(\pi) \geq 0$:

$$J(\pi') - J(\pi) \geq J_{\pi}^{CPI}(\pi') - C \cdot \sqrt{E_t\left[KL\left(\pi'(a_t,s_t)||\pi(a_t,s_t)\right)\right]} \tag{4.8}$$

Looking at Equation 4.8 we can see, that $J(\pi') - J(\pi) \geq 0$ only holds, if $J_{\pi}^{CPI}(\pi')$ is greater than the maximum error. If we only update our policy if this condition holds, we can guarantee policy improvement even if we only estimated $J(\pi') - J(\pi)$. In a worst case scenario the best new policy will be the old policy, since if $J_{\pi}^{CPI}(\pi') = 0$ and $KL(\pi||\pi) = 0$ we can still accept the old policy as new policy. Using Equation 4.8 we can finally formulate an updated version of the optimization problem:

$$\arg\max_{\pi'}(J_{\pi}^{CPI}(\pi') - C \cdot \sqrt{E_t\left[KL\left(\pi'(a_t,s_t)||\pi(a_t,s_t)\right)\right]})$$

The remaining question now is how to implement this optimization problem in practice. Usually the error term is directly bounded to some constant $\delta$ which directly limits how large the KL divergence can be:

$$E_t[KL(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))] \leq \delta$$

Of course $\delta$ is a hyperparameter that must be tuned for each individual RL problem. Since we are usually learning from a single step (or a small batch), the expectation is also written as an expectation over a single step. Since we are maximizing the objective in parameter space we are now writing our equation as dependent on these parameters $\theta$, with $\pi' = \pi_\theta$ and $\pi = \pi_{\theta_{old}}$. Therefore we arrive at the final surrogate objective:

$$\begin{aligned} &\max_\theta E_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t^{\pi_{\theta_{old}}} \right] \\ &\text{subject to } E_t[KL(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))] \leq \delta \end{aligned}$$

When optimizing this surrogate objective we are maximizing a linear approximation of $J(\pi') - J(\pi)$ with an imposed trust region, limiting the difference between the old and the new policy and thus avoiding performance collapse. Our new objective also now guarantees monotonic improvement.

**The PPO Algorithm.** The *Proximal Policy Optimization* (PPO) algorithm was proposed by Schulman et al. in 2017 [73]. It uses the idea of trust regions with a minimum of additional computation. It also replaces the hyperparameter $\delta$ with a new hyperparameter which is easier to tune. In their original paper Schulman et al. proposed two different variants for the trust region constraint: An adaptive KL penalty - which uses a variable $\delta$ - and a clipped objective. In their tests, both variants proved to have good performance, but since the clipped objective is easier to tune, easier to implement and at least as good as the version with the KL penalty we will omit further details on adaptive KL penalty.

To simplify our equations and add a little more clarity we will only write $A_t$ for $A_t^{\pi_{\theta_{old}}}$ in the following. We also define our probability ratio as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. This way we can rewrite our surrogate objective as

$$J^{CPI}(\theta) = E_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t^{\pi_{\theta_{old}}} \right] = E_t[r_t(\theta) A_t]$$

We will now take a look at the *clipped surrogate objective*, which adds a clipping term to $J^{CPI}$:

$$J^{CLIP}(\theta) = E_t[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

The interesting thing is, that we can use this objective without adding a constraint for the KL divergence. If we look at one step of training and some $A_t > 0$ one option to maximize $J^{CPI}$ would be to just perform a large policy update - thus maximizing $r_t(\theta)$. This is not what we want, because we already know that we are optimizing in parameter space and large policy updates are risky. In $J^{CLIP}$ we therefore clip the value of $r_t(\theta)$ to an $\epsilon$-neighborhood between $(1-\epsilon)A_t$ and $(1+\epsilon)A_t$. As long as $r_t(\theta)$ is in the interval

$[1-\epsilon, 1+\epsilon]$, $J^{CLIP}$ is equal to $J^{CPI}$, but as soon as the new action probability diverges too much from the old policy the value is clipped. Hence, policy updates under the clipped objective function are always within an $\epsilon$ neighborhood and we do not need an additional constraint.

**Algorithm 4.4.3:** The PPO algorithm with clipped surrogate objective as an extension to synchronous parallelized A2C. (Adapted from [42].)

**Data:** Actor learning rate $\alpha_A$, Critic Learning Rate $\alpha_C$ Entropy coefficient $\beta \geq 0$, Clipping variable $\epsilon \geq 0$, Number of Epochs $K$, Number of Actors $N$, Time Horizon $T$

**Result:** Optimized parameters for the actor $\theta_A$ and critic $\theta_C$

$M \leftarrow NT$
Initialize all the weights $\theta$ randomly
Initialize the old actor network $\theta_{A_{old}}$
**for** $i = 1, 2, \ldots$ **do**
  $\theta_{A_{old}} \leftarrow \theta_A$
  **for** $actor = 1, 2, \ldots, N$ **do**
    Sample trajectory $\tau = ((s_0, a_0, r_0), \ldots, (s_T, a_T, r_T))$ from the environment with policy using $\theta_{A_{old}}$.
    Calculate $V_{\theta_C}(s_0), \ldots V_{\theta_C}(s_T)$ using the value network
    Compute advantages $A_0, \ldots, A_T$ using $\theta_{A_{old}}$
    Calculate $V_{tar}(s_0), \ldots V_{tar}(s_T)$ using the advantage and the value network $\theta_C$.
  **end**
  Construct a *batch* with size $M$ from trajectories, advantages and target V-values.
  **for** $epoch = 1, 2, \ldots, K$ **do**
    **foreach** *minibatch m in batch* **do**
      /* The following are computed over the whole batch */
      Calculate $r_m(\theta_A)$
      Calculate $J_m^{CLIP}(\theta_A)$ using advantages $A_m$ and $r_m(\theta_A)$
      Calculate entropies $H_m$ using actor network $\theta_A$
      /* Calculate loss */
      $\mathcal{L}_{val}(\theta_C) = \frac{1}{M}\sum_{t=0}^{M}(V_{\theta_C}(s_t) - V_{tar}(s_t))^2$ // Value loss using MSE
      $\mathcal{L}_{pol}(\theta_A) = \frac{1}{M}\sum_{t=0}^{M}\left(J_t^{CLIP}(\theta_A) - \beta H_t\right)$ // Policy loss
      /* Update parameters with SGD optimizer */
      $\theta_C \leftarrow \theta_C - \alpha_C \nabla_{\theta_C}\mathcal{L}_{val}(\theta_C)$ // Update Critic
      $\theta_A \leftarrow \theta_A - \alpha_A \nabla_{\theta_A}\mathcal{L}_{pol}(\theta_A)$ // Update Actor
    **end**
  **end**
**end**

Removing the KL divergence constraint makes the clipped objective function computa-

tionally inexpensive. The only computations that are done are the probability ratio $r_t(\theta)$ and the advantage $A_t$. Both of them are the minimal computational cost we have to invest for any policy update optimizing the surrogate objective. Algorithm 4.4.3 shows PPO with a clipped objective function as an extension of the actor-critic algorithm from Section 4.4.1. We also use synchronous parallelization.

The algorithm shown has some major differences with the original A2C algorithm from Algorithm 4.4.2 so lets go over them:

- We got a few new hyperparameters. The number of actors defines the number of parallel agents. We will talk about the number of epochs and the time horizon later on.
- *Line 4*: Our Main loop changed from episodes to an infinite loop. Usually this outer loop will train for a fixed number of steps.
- *Lines 6-11*: Each actor trains for a set amount of steps in its own copy of the environment. We call this amount of steps the *Time Horizon* $T$. Since we are using the synchronous variant of A2C, each actor uses the same network weights. After sampling the trajectory we are computing state values and estimating advantages via GAE for each step. We are also computing the V-target values.
- *Lines 12-14*: After we have generated the trajectories and computed state-values and advantages, we create a batch for the network updates. Since the objective function only needs samples from the old policy $\pi_{\theta_{old}}$, we can use data from the trajectory multiple times. This increases sample efficiency. We create random samples from of our batch of training data with size $T$. These smaller batches are called *minibatches* and we compute a parameter update for each of them. We will repeat this for $K$ epochs.
- *Lines 15-21*: We calculate the probability ratio $r_m(\theta_A)$ our objective $J_m^{CLIP}(\theta_A)$ and the entropy $H_m$ based on the current minibatch. We then calculate the loss for the actor and the critic and apply the resulting gradients with the SGD optimizer of choice.

### 4.4.3 Alternative Actor-Critic Algorithms

We already presented one of the most well-known members of the actor-critic family - the PPO algorithm - in Section 4.4.2. In this section we want to give a short introduction into two additional members of the actor-critic family: ACER which aims at improving sample-efficiency by making it possible to learn from past experiences and ACKTR, which proposes an alternative trust region algorithm.

**ACER.** *Actor-Critic with Experience Replay* (ACER) was developed in 2016 by Wang et al. [82]. The goal of ACER was to create an actor-critic successor which could handle off-policy training and thus being more sample efficient. ACER also implements trust-region policy optimization.

At its base, ACER changes the actor-critic architecture to learn Q-Values instead of state-values. This is achieved by using the Retrace algorithm by Munos et al. [57] which estimates target Q-Values with minimal bias and variance and has been proven to converge:

$$Q^{ret}(s_t, a_t) = r_t + \gamma \bar{r}_{t+1}(\theta) \left[ Q^{ret}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_{t+1}, a_{t+1}) \right] + \gamma V^{\pi}(s_{t+1})$$
$$\text{with } \bar{r}_t(\theta) = \min(c, r_t(\theta))$$

After sampling a trajectory we can compute the values for $Q^{ret}$ from the last timestep backwards, since $Q^{ret}(s_T, a_T) = 0$. We can then use $Q^{ret}$ as target value and compute the mean square error loss for backpropagation.

For the training of the actor, ACER also uses the target Q-Values. Similar to PPO, sample returns will be weighted by the probability ratio between the old and the new policy $r_t(\theta)$. Learning from old trajectories requires to save the probability distribution over actions from the current policy in the replay buffer. The problem for off-policy training is that the importance weights may become very large over time (since the current and some old policy may produce vastly different probability distributions for some state) causing training to become unstable. To prevent this, the importance weights are clipped to a fixed value $c$. To ensure an unbiased estimation they also add a correction term which is called the *truncation with bias correction trick*

$$\begin{aligned} \mathcal{L}^{acer}_{pol}(\theta_A) =& \bar{r}_t(\theta) \log \pi_A(a_t|s_t)[Q^{ret}(s_t, a_t) - V_{\theta_C}(s_t)] \\ &+ E_{a\sim\pi}\left( \left[ \frac{r_t(\theta, a) - c}{r_t(\theta, a)} \right]_+ \log \pi_{\theta_A}(a|s_t)[Q_{\theta_C}(s_t, a) - V_{\theta_C}(s_t)] \right) \end{aligned}$$

$$\text{with } r_t(\theta, a) = \frac{\pi_\theta(a|s_t)}{\pi_{\theta_{old}}(a|s_t)}$$
$$\text{and } [x]_+ = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

The above expression also uses a baseline $V_{\theta_C}(s_t)$ which is subtracted to reduce variance. The first term is the standard loss, weighted by the clipped importance weights and the second term is the correction term.

We already saw that actor-critic variants may have problems when updating the policy, since updates take place in the parameter space. Similar to PPO, ACER uses a trust region when performing updates. However ACER uses a different method: Instead of constraining the policy to not deviate too much from its predecessor, ACER maintains an *average policy network*. This network represents a running average of past policy networks. When updating, ACER computes the KL divergence between the new and the average policy network and thus bounds the policy from deviating too far.

**ACKTR.** *Actor Critic using Kronecker-Factored Trust Region* (ACKTR) has been developed in 2017 by Wu et al. [86]. ACKTR combines an improved optimization method called *natural gradient descent* with yet another idea of how to compute trust regions for actor-critic updates. ACKTR has proven to have higher sample efficiency and learn better results in less time than A2C or TRPO.

Today most RL algorithms use stochastic gradient descent to calculate gradients for the network. SGD optimizers use first order derivates on the loss function, meaning they compute the slope of the loss function at a given point. First order derivates are fast and easy to compute and combined with extensions like momentum provide reasonably fast results. However by only looking at the first derivate, SGD is very sensitive to the learning rate. By calculating gradient updates based on the descent at the current point only, we do not take the change of descent into account. This means, that the direction of the gradients is not optimal and does not point directly into the direction of the optimum. Also we still have to face the problem we explored earlier: Changes to the weights are computed in parameter space assuming an equal change to the objective (in policy space). We saw, that it is possible to limit the change in policy space by setting an upper bound for the KL divergence - ensuring our new policy would be "close enough" to the old policy. But what if we were able to optimize our policy in policy space and compute the gradients in parameter space from there? This would not only solve the problem of diverging from the optimum, but also speed up learning, as our gradient descent would point directly to the optimum of the cost function.

In 1998, Shun-Ichi Amari presented his approach of using a *natural gradient* instead of standard SGD [7]. The idea is that we build a connection between each parameter from our model (from parameter space) to the policy space which tells us how great of an impact the change of each parameter will have in policy space. This way we can compute the natural gradient as the largest change in objective per unit change in our model parameters. By performing these weighted gradient descent steps in parameter space the resulting changes in policy space will have equal distance.

If we look at gradient descent and some loss $\mathcal{L}(\theta)$, the steepest descent direction can be defined as the vector $d$ which minimizes $\mathcal{L}(\theta + d)$ under the constraint that the squared length of $|d|^2 \leq \epsilon$. This length of $d$ is $\sum_i (d_i)^2$ in an euclidean space and $\sum_{i,j} g_{i,j} d_i d_j$ in any arbitrary coordinate system. The values $g_{i,j}$ are given by a positive semi-definite matrix $G$. The second formula for the distance is equal to the first one if $G$ is the identity matrix $I$ (which is true for any euclidean space). Amari showed, that we can define the direction of the steepest gradient descent in any space as

$$\tilde{\nabla}\mathcal{L}(\theta) = G^{-1}\nabla\mathcal{L}(\theta)$$

The matrix $G$ can be computed by using the *Fisher Information Matrix* (FIM) as a replacement. The FIM defines how much information an observable random variable $X$ carries about the unknown parameters $\theta$ of a probability distribution $p$ on which $X$ depends. The FIM is given by

$$F = E\left[\nabla_\theta \log p_\theta(y|x) \nabla_\theta \log p_\theta(y|x)^T\right]$$
$$\approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log p_\theta(y|x_i) \nabla_\theta \log p_\theta(y|x_i)^T$$

with an expectation over the data distribution of the training data. The probability distribution is our neural network (e.g. the actor policy $\pi_{\theta_A}$). The interesting property of the FIM is that it is equal to the Hessian (the second order derivate) of the KL-divergence. Therefore the FIM defines the local curvature in policy space. The natural gradient is therefore defined as:

$$\tilde{\nabla}_\theta \mathcal{L}(\theta) = F^{-1} \nabla_\theta \mathcal{L}(\theta)$$

The computation for natural gradient descent follows Algorithm 4.4.4 and is very similar to SGD with the exception of the computation of the FIM.

**Algorithm 4.4.4:** The Natural Gradient Descent Algorithm.

**Data:** Learning rate $\alpha$
**Result:** Optimal parameters $\theta$ minimizing the loss function $\mathcal{L}(\theta)$
**while** *Not Converged* **do**
  Compute the loss $\mathcal{L}(\theta)$ like before.
  Compute the gradient for the loss $\nabla_\theta \mathcal{L}(\theta)$
  Compute the FIM from the trajectory.
  Compute the natural gradient $\tilde{\nabla}_\theta \mathcal{L}(\theta) \leftarrow F^{-1} \nabla_\theta \mathcal{L}(\theta)$
  $\theta \leftarrow \theta - \alpha \tilde{\nabla}_\theta \mathcal{L}(\theta)$
**end**

In 2002 Sham Kakade was able to show that the natural gradient can be applied to policy gradient methods and converges faster than optimization with SGD in an RL setting [39]. Unfortunately despite this proof of concept, the computation of the FIM is computationally very expensive. Even worse, for the size of nowadays neural networks with millions of parameters, the computation of the FIM would exceed main memory capacity. The only way we are able to use the FIM is to find a reasonably good approximation.

In 2015 James Martens and Roger Grosse presented their novel method called *Kronecker-Factored Approximate Curvature* (KFAC) [50] which is specifically designed to efficiently approximate FIM matrices for neural networks. The idea is to calculate approximations for the FIM layer-wise by assuming, that second-order statistics of the activations in layer $l$ and the backpropagated derivates are uncorrelated. While this assumption is not exact, in reality it still yields good results. KFAC can be used to train both the actor and the critic if we define the output of the critic as a Gaussian distribution - for which we can then can calculate the Fisher matrix.

Since we are only working on an approximation of the Fisher matrix and it was observed, that in the context of deep RL unbounded updates can still lead to large policy updates, ACKTR also uses a trust region method. Ba et al. showed, that since the Fisher matrix has a direct connection to the KL-divergence, it can be used as a replacement when formulating the trust region [10]. This can be done by setting the effective step size $\eta$ for an update of the parameters to

$$\eta = \min\left(\eta_{max}, \sqrt{\frac{2\delta}{\mathbf{v}^T \hat{F} \mathbf{v}}}\right)$$

where $\delta$ is the trust region radius, $\mathbf{v} = \Delta\theta$ is a vector representing the parameter change and $\hat{F}$ is the Fisher matrix approximation. ACKTR has shown exceptional good performance in comparison with A2C and TRPO in both maximum performance and wall-clock time.

### 4.4.4 Learning with Curiosity

So far all our agents tried to maximize one thing: The total reward they get from the environment. This behavior produces multiple problems:

- The agents behavior is directly dependent on the reward function, so correctly shaping the reward is crucially important for every new environment.
- If rewards are only given in a very sparse setting, agents may fail at exploring the environment to the point where they actually get rewarded. This also happens because the agent only learns from loss and the loss is only dependent on the reward. If our agent is not able to get any reward, it will be unable to learn anything.
- Even if the agent is able to learn, it may fail at deeply exploring the environment further, because every change in policy may result in less reward for the moment.

If we look at these problems, they are all related to a problem of exploring the environment. In the past, we explored the environment by random chance, but that does not seem to be sufficient. As a human, how do we get motivated to do something? Do we only learn or explore if we get rewarded for it by someone? The answer is no. We get reward, but it is not given to us, it is intrinsic to ourselves: We are curious about our surroundings and by satisfying that curiosity we generate "reward". Curiosity drives us to learn things that do not serve any need at the moment, but might become handy in the future. This idea motivates us to extend the basic reward function for our agent to include a term for curiosity.

Modeling curiosity for RL agents has been mainly explored in two different approaches: Encouraging the agent to explore novel states or encouraging the agent to perform actions which outcome cannot be predicted well. Measuring novelty has been shown to be difficult, since it requires to model the unknown statistical distribution of environmental states. In 2017 Pathak et al. proposed an *Intrinsic Curiosity Module* (ICM) which generates intrinsic

reward $r_t^i$ additional to the extrinsic reward $r_t^e$ from the environment [61]. They showed, that it is possible to train agents using purely intrinsic reward and that agents trained with both intrinsic and extrinsic reward, outperform agents trained purely with extrinsic reward.

Let us take a look at how the ICM works. The idea of the ICM is to use prediction error as an indicator for curiosity reward. Intuitively this makes sense, since prediction error can be seen as the agent being surprised by a state. But we have to be slightly more precise when it comes to this error. Let us look at an example where the agent gets an observation from a camera. Under low light conditions the image will get more and more noisy, making situations with low light more interesting. Similar things are happening with various other things, like a TV outputting static noise, or leafs moving in the wind. In all these situations the observation (especially observations which are images) will change unpredictably and thus be surprising to the agent. We are therefore interested in a feature representation, which is only dependent on things the agent has influence over or which directly influence our agent in the environment. Everything else is just noise and should be ignored.

To calculate the reward we will use an embedding network which transforms the observation of the current state into a representation $\phi(s)$. We will then use a *forward dynamics network* which should predict the representation of the next state $\hat{\phi}(s_{t+1})$ from the current state and action the agent took in state $s_t$. This network can be expressed as $p(\phi(s_{t+1})|s_t, a_t)$. For a given transition tuple $(s_t, a_t, s_{t+1})$ the intrinsic reward is then given by the negative log likelihood $-\log p(\phi(s_{t+1})|s_t, a_t)$ and usually calculated using the mean square error of the forward dynamics network:

$$r_t^i = \frac{\eta}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||^2$$

with a scaling factor $\eta > 0$. The prediction error for a state will naturally be higher if the state or very similar states have not been often explored in the past, thus encouraging the agent to do so in the future.

Burda et al. investigated the performance of this intrinsic reward in combination with several feature spaces for forward dynamics [19]. Two different approaches were found which both yield good results:

1. **Random Features (RF).** Random features are generated by a convolutional neural network. All parameters of that network will be fixed after the inital random initialization. This procedure results in stable features, but the features itself are not guaranteed to neither filter out any unnecessary information nor contain all necessary ones. Surprisingly random features still yield good results. Random features are easy to implement and do not require extra training time.

2. **Inverse Dynamics Features (IDF).** Inverse dynamics features were initally proposed for the ICM and are generated by extending the embedding network with a second head. Given two states $s_t, s_{t+1}$ the network should predict the action $a_t$ the agent

choses in state $s_t$. The networks first layers are used as a common neural network $\phi$ to embedd the observations. The idea is, that by training this network to also predict the action, the feature representation will be trained to ignore all aspects of the observation which are not under the agents immediate control. However this representation will not show aspects of the environment which will influence the agent but are not under its control.

While both approaches have shown to work well with the ICM, they both failed at reliably eliminating the influence of stochastic transitions in the environment. Even with IDF features, if the agent has the possibility to influence the appearance of stochastic features (like randomly changing static noise on a TV) it would tend to only explore these stochastic states after some time. Also when using IDF we are dealing with stochasticity induced by sampling an action from our policy which cannot be predicted by our deterministic predictor network. Burda et al. therefore proposed a new approach which they called *Random Network Distillation* (RND) in another paper [20].

RND is not an embedding for the observation and instead replaces the whole ICM. The intrinsic reward in RND is generated by a random prediction problem. This problem is build by two neural networks: The first network is called the *target* network which is randomly initialized and fixed - similar to the random feature network from before. This target network takes the observation and transforms it into a random feature representation $f : \mathcal{O} \to \mathbb{R}^k$. We then construct a second network with the same architecture called the *predictor* network which also takes the observation as input $\hat{f} : \mathcal{O} \to \mathbb{R}^k$. The predictor network is trained to minimize the mean square error between its own output and the target network $||\hat{f}(s) - f(x)||^2$. The idea is, that the prediction error will be larger for observations which are dissimilar to observation seen in the past. It could be shown, that gradient descent is unable to mimic the target network perfectly for any input and therefore the error can always be used to generate the intrinsic reward signal.

The authors showed, that for actor-critic variants it can be beneficial to also modify the architecture to predict two state-values instead of one. The first value head predicts the state value for expected environment return (like before) and the second one predicts the value for the expected intrinsic reward. They showed that this method can improve learning further, if rewards are not clipped at the end of an episode (restart is just the next state).

Curiosity - especially with RND - has shown to vastly improve the performance of PPO especially for environments with sparse reward. It also has been proven to be beneficial for environments with dense reward - leading to agents which learn both faster and reach a higher final reward. Curiosity can be combined with any RL algorithm, but only has been extensively studied in combination with PPO so far.

# 5 Implementation

In this chapter we will take a look at different parts of the implementation. We will begin by discussing some general implementation notes in Chapter 5.1 and then continue with an overview of our experimentation environment called *baselines lab* in Section 5.2. Finally, we will talk about the particle simulation environment, including reward shaping, extended environment models and instance generation in Section 5.3.

## 5.1 General Implementation Notes

Before dive into the actual implementation we want to talk a little bit about our general design choices. As a programming language we chose to use Python 3 [80, 27]. Python is an interpreted high-level programming language. Even though it is interpreted and thus may be slower than a direct implementation in C or C++, the easy integration of high-performance libraries which also scale into distributed systems make applications developed in python very fast. In the past years many ML library have been developed for python making Python an ideal choice for data-science and machine learning projects.

To date, we have the choice between several machine learning libraries which all support the creation and training of artificial neural networks. Modern libraries also especially leverage the computing power of the GPU to accelerate computation. Many of the libraries support more or less the same functionality, but may have a very different style of how certain things can be achieved. Currently the two most popular libraries for machine learning (with Python) are PyTorch [60] and Tensorflow [1]. For this project we decided to use Tensorflow in version 1.14. This decision was made in conjunction with our second big backend library which extends Tensorflow for reinforcement learning: Stable-Baselines [32]. Stable baselines offers high-quality implementations of many RL algorithms which are partially forked from the OpenAI Baselines project [24]. While Google offers their own reinforcement library as a Tensorflow extension named TF-Agents [74] we found that it currently is in a too early development state and thus not stable enough.

Since Python offers the easy integration of additional libraries we also make use of a number of additional libraries, most notably:

- OpenAI Gym [18] for a standardized environment creation as well as the integration of the Atari environments for testing purposes.
- OpenCV [17] for image preprocessing.
- Numpy [58] for fast matrix computations.
- Optuna [4] for automated hyperparameter search.

We provide a complete list of all libraries and their respective version which were used in this project in Appendix TODO.

## 5.2 Baselines Lab

To test various combinations of reinforcement learning algorithms, their settings with different environments and environment settings, we need a flexible and easy configurable program which also gives us the possibility to monitor the performance and create evaluations of trained models.

While there already exist some projects which offer some of these features, we did not find a project which was fully suited for our needs. The stable-baselines co-project baselines zoo [64] had to few options for the configuration of algorithms and while other libraries like slm-lab [41] did fit our needs for configurability, they did not offer enough integrated reinforcement learning algorithms. Therefore we decided to build our own lab environment on top of stable-baselines which we therefore call *Baselines Lab*.

### 5.2.1 Basic Features

In this section, we want to go over the basic features of Baselines Lab. When starting the lab, we have to choose between one of the three different lab modes:

1. The *train* lab mode can be used to train a new model.
2. The *enjoy* lab mode can be used to watch a model in action and/or evaluate its performance.
3. The *search* lab mode can be used to perform an automated hyperparametersearch. We will talk more about the search mode in Section 5.2.3.

When starting a session, we also need to specify a *lab configuration file* which can be written in JSON or YAML. Let us take a look at the easiest form of a config file which specifies all mandatory parameters:

We can see that the basic lab configuration only requires a handful of parameters which define which RL algorithm should be used in conjunction with which neural network and which environment. Additionally we have to specify how long the agent should be trained.

In general there are three basic categories for the parameters defined by the three main keywords:

1. The *algorithm* keyword specifies all parameters related to the reinforcement learning algorithm. The parameters which can be set are algorithm dependent and directly correspond to the parameters of the stable-baselines documentation [33]. The only exception to this is the *policy* keyword, which defines the used neural network. We can use any registered policy type by using the *name* keyword. Additional policy arguments can be directly specified as children of the *policy* keyword.

```yaml
algorithm:
  name: "ppo2"
  policy:
    name: "MlpPolicy"

env:
  name: "CartPole-v0"

meta:
  n_timesteps: 10000
```

Figure 5.1: Basic lab configuration file in YAML.

2. The *env* keyword specifies all parameters which are directly related to the environment. The only mandatory parameter is the name of the environment corresponding to its registered gym entry. The env keyword allows for a lot of parameters regarding observation preprocessing like frame stacking or normalization. We included a full list in Appendix TODO.

3. The *meta* keyword describes all additional parameters related to the training process. For example we can define how often the model should be saved or evaluated, which random seed should be used or how many times the experiment should be repeated. For a full list of available arguments, see Appendix TODO.

The same configuration file can be used for all lab modes. However some keywords may only take effect in a certain lab mode. By specifying a log directory, we can monitor the training process and periodically save the model during training. The latest or best models can then be inspected in enjoy mode by starting the lab with the same configuration file.

**Automated Saving and Loading.** Given a log directory, Baselines Lab will automatically save models during training. Saving a model is complicated, since the model may need more than the network parameters to be correctly loaded in the future - either for inspection or extended training. For example if we normalized the observations of the environment using a running mean, we need to save that running mean, otherwise our model would not be able to perform later on. A single checkpoint therefore may contain multiple files depending on the current configuration.

If a log directory location is specified the lab will create a log directory with a current timestamp. Models (and additional components) will be periodically saved during training to a subdirectory called *checkpoints*. On default, the lab keeps the last 5 checkpoints *and* the model which had the best performance during training. This behavior can also be changed (see Appendix TODO).

The checkpoints created during training can be used in enjoy mode or if we want to continue training in the future. To continue training, we just add a `trained_agent`

keyword to the algorithm configuration and specify weather we want to load the *last* or the *best* checkpoint. In enjoy mode the best checkpoint is selected on default. To change this behavior we can specify a `--type` command line argument and set it to *last*.

**Automated Evaluation** Models are automatically evaluated during training. This evaluation is done periodically to determine the current model performance. Evaluation can be done in three different modes depending on the desired tradeoff between evaluation accuracy and speed:

1. **Fast.** In fast evaluation mode, the lab is not performing a dedicated evaluation run. Instead the current results are computed by looking at the average reward and episode length over the last 100 training episodes. This procedure is very fast because the results are already computed during training, but may be inaccurate and also include the training behavior of the agent, which might be substantially different from its test behavior depending on the used RL algorithm (e.g. for DQN).
2. **Normal.** In normal evaluation mode, the lab uses a dedicated vectorized evaluation environment with 32 parallel instances. Because each instance is reset independently this evaluation type still might produce a small bias towards short episodes, which slightly improves results if short episodes are good and slightly worsens results if short episodes are bad. Nevertheless we found that the results do not deviate too much from the results when executing the same model on a single environment and thus the results can be used to compare different models.
3. **Slow.** In slow evaluation mode, the lab uses a dedicated evaluation environment with a single instance. This might take a lot of time during training, but produces the most accurate evaluation results.

Evaluation during training is used to determine the current best model and for monitoring purposes. To evaluate models outside of the train lab mode, we can use the enjoy lab mode. By specifying the `--evaluate` command line argument, the lab creates a file at the log directory location containing detailed evaluation results. The argument expects a number of episodes and uses the normal evaluation mode on default. If we are interested in completely accurate results, we can additionally specify the `--strict` option.

**Monitoring** Monitoring the training process is very important for machine learning. By logging all kinds of values, we are able to determine errors in the implementation, select good hyperparameters and overall control the quality of the training process. Luckily, Tensorflow comes with an integrated monitoring system called Tensorboard which allows us to inspect the neural network and log values during training. These values can then be plotted in real time by starting the Tensorboard web server application. Figure 5.2 shows an example of the Tensorboard visualization, which can be accessed with a web browser.

Stable Baselines already integrates Tensorboard for extended logging and visualization. Dependent on the chosen RL algorithm, Stable Baselines will log internal values like the

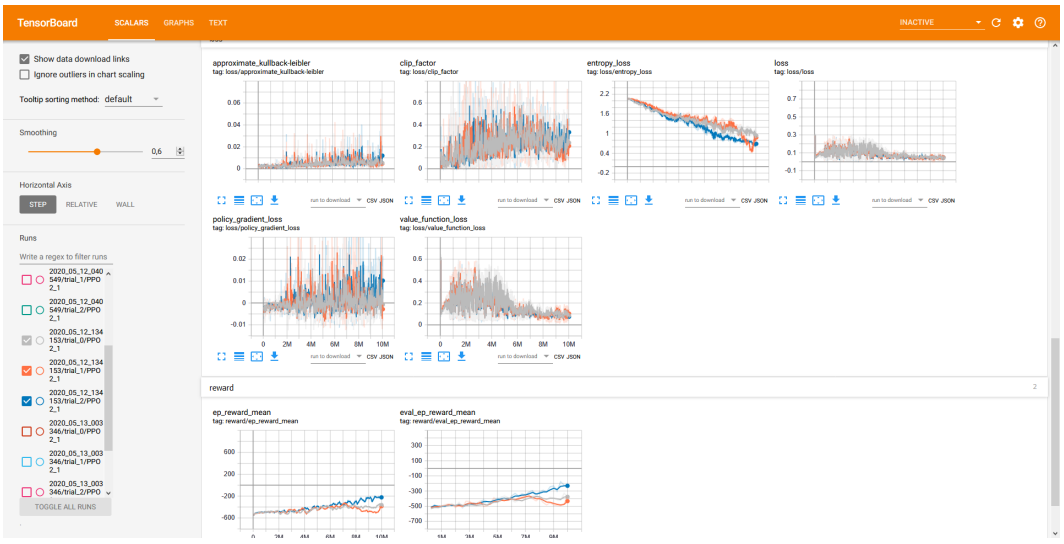



Figure 5.2: Example for a Tensorboard visualization.

average action advantage or the discounted reward. With Baselines Lab, we extend these logging capabilities and now also include several new values:

- Smoothed episode length and episode reward for both training and evaluation.
- Special values - e.g. intrinsic and extrinsic reward when using the RND wrapper.
- The lab configuration as text. This is handy in situations, where Tensorboard visualizes multiple runs at once.
- Periodic images showing the state of the environment or the observations. This makes it easy to see what strategies the agent has learned without executing the agent in train lab mode.
- Frames per second to monitor training performance.

Tensorboard logs are created automatically at the given log directory location if the `tensorboard_log` parameter of the algorithm is set to true. Some values like periodic images require additional opt-in since they may require a lot of disc space. See Appendix TODO for more information.

**Extension of Stable Baselines Functions.** Additional to the extended monitoring, we also extended some other functions of Stable Baselines:

- *Learning Rate Schedules.* Stable Baselines includes an inconsistent system of learning rate schedules, which are only available for certain algorithms. In Baselines Lab we therefore included linear and piecewise learning rate schedules for the PPO, SAC and TD3 implementations. For information on how these schedules can be used we included an example in Appendix TODO.

- *Flexible Neural Network Creation.* Stable Baselines only allows to flexibly create dense neural networks. We extended the network base classes to also include a fully configurable network class which can be defined via config files. For an example see Appendix TODO.

### 5.2.2 Random Network Distillation Module

Stable Baselines does not offer an implementation for any form of curiosity reward. Since Huang et al. had shown, that curiosity reward can significantly improve training, we needed to implement our own version of intrinsic reward. In all previous experiments the RND curiosity reward produced superior results in comparison with the original idea of an ICM module. We therefore decided to implement the RND reward.

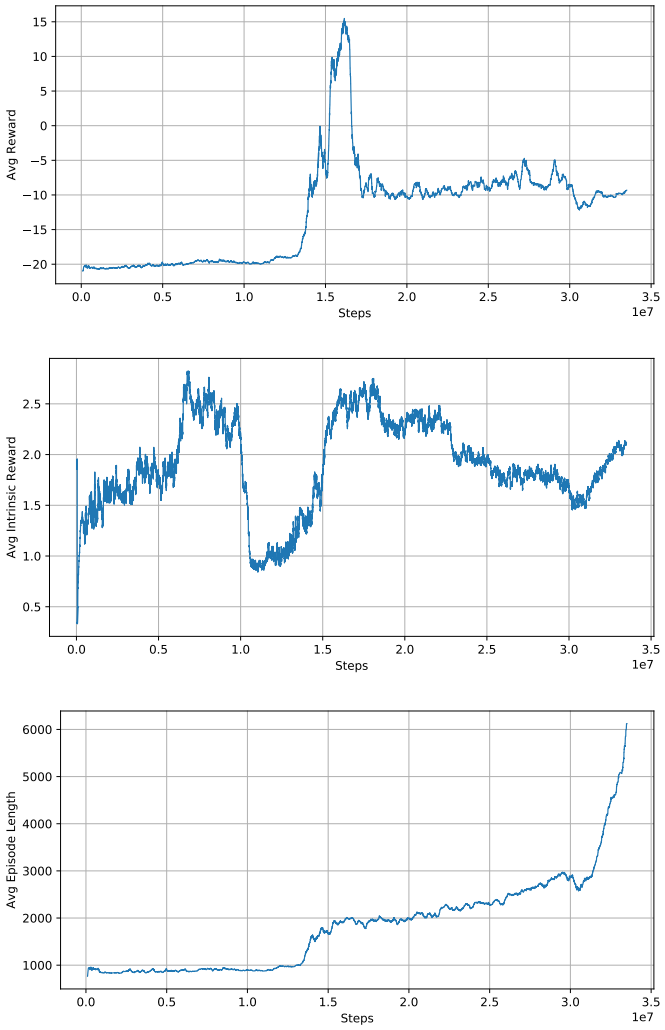



Figure 5.3: Learning curves for the verification experiment of the curiosity reward wrapper on the Atari game Pong.

Instead of implementing the curiosity reward as an extension of PPO (like in the original implementation) we decided to implement RND as a wrapper for the environment. This wrapper contains an internal replay buffer which contains samples from the past four training periods. On default, we train the predictor network with the same frequency as the actor network. For example, if we train 256 steps on 16 environments between each training period of the actor, the replay buffer will have a size of $256 \times 16 \times 4 = 16384$ samples. When training the predictor, we randomly draw batches of $0.25 * \text{size(buffer)}$ from the buffer. On default, we perform four optimization epochs. The networks used for the target and predictor network have the same structure as in the original implementation.

The implementation of the RND module as a wrapper allows us to use the curiosity reward signal independent of the RL algorithm. We verified our implementation by training an PPO agent to play the Atari game "Pong" using curiosity reward only and no end-of-episode signal. Figure 5.3 shows the learning curves for intrinsic and extrinsic reward, as well as the episode length. We can see, that the agent does not optimize for extrinsic reward - this is expected, since we removed that reward signal - and instead continuously improves the intrinsic reward by increasing the episode length. The agent therefore learns to play with its enemy in Pong instead of against him. This behavior was also observed by Burda et al.[20].

The development of the curiosity wrapper was done in collaboration with the development team of Stable Baselines, and the implementation has been staged for a future release of Stable Baselines 3 [65].

### 5.2.3 Automated Hyperparametersearch

The choice of optimal hyperparameters is very important in any machine learning setting. As we saw in Chapter 4, most RL algorithms introduce multiple new parameters, with some of them having great influence on the training performance. These parameter must be tuned additional to the already existing hyperparameters like the learning rate or other optimizer specific values, an observation preprocessing pipeline and eventual environment parameters (e.g. for reward generation). Tuning all these parameters together is especially hard, because every training run may take hours before we can determine if the current configuration is better or worse than any other configuration. To support our decisions when tuning these parameters, we therefore need the possibility of an automated hyperparametersearch.

To efficiently search for hyperparameters in an ML context, a number of algorithms have been developed in the past which all have their individual strengths and weaknesses. Modern optimizer frameworks therefore often allow to choose between a number of different algorithms. For Baselines Lab, we choose to integrate Optuna [4] as it combines a number of up-to-date algorithms with great configurability. In Optuna there are mainly two classes of important algorithms: Samplers and Pruners. Samplers are used to choose from a set of hyperparameters and pruners are used to decide which trial should be canceled early (pruned). While both classes can be used with simple algorithms (e.g. a random

```yaml
search:
  n_trials: 40
  sampler: "tpe"
  pruner:
    method: "halving"
    min_resource: 4000
  algorithm:
    learning_rate:
      method: "loguniform"
      low: 1.0e-4
      high: 0.1
  env:
    n_envs:
      method: "categorical"
      choices: [4, 8]
```

Figure 5.4: Basic search lab configuration file.

value sampler with a median pruner), in Optuna we also can choose from more advanced algorithms. For example sampling can be done using the tree-structured parzen estimator algorithm [15] or pruning can be done via successive halving [40].

With Baselines Lab we can now configure the hyperparametersearch via config files. In Figure 5.4 we included an example showing an excerpt of a lab configuration. The shown section for the *search* keyword can be appended to the lab configuration from Figure 5.1 and is only used if the file is started with the `search` lab mode. We can see that we can define how many different configurations should be tested via the `n_trials` keyword and that we can define a sampler and a pruner method. Each pruner can receive its individual parameters by specifying them as children of the `pruner` keyword. For example the *halving* pruner gets configured such that each trial will run for at least 4000 timesteps before it can be pruned.

Depending on the chosen RL algorithm, Baselines Lab comes with a predefined set of parameter ranges which will automatically be sampled. Parameters which are explicitly given via the normal algorithm or env keyword will not be changed during the parametersearch. If we want to include additional parameters, we can define them under search/algorithm or search/env with a sampler method and the according choices. More details can be found in the Optuna documentation [3] and in Appendix TODO.

### 5.2.4 Additional Features

Baselines Lab offers a bunch of additional functions which are designed to help monitoring the training process, find problems with the learning algorithm and help with result presentation:

- *Plots.* Baselines Lab can automatically create plots of the tensorboard training data. To create the plots either specify `plot: true` under the `meta` keyword or use the `--plot` command line argument in enjoy mode. The default plots contain train and evaluation data for reward and episode length, but it is possible to plot additional data by specifying their tensorboard tags. For more information see Appendix TODO.

- *Video Creation.* Baselines Lab can automatically create videos of the environment. The environment can either be rendered like it is displayed in enjoy mode or the environment can render the observations which are created for the agent. Both options are only available in enjoy mode and can be enabled with the command line arguments `--video` or `--obs-video` respectively.

- *Email Notification.* Baselines Lab can automatically send email notifications to send updates about the training progress. This feature is only available on Linux and requires *mailx* to be installed. To send email notifications we have to specify a mail address via the `--mail` command line argument.

## 5.3 The Particle Maze Environment

Since most RL algorithms require millions of steps of training before they can solve a problem, we need a very efficient implementation for our environment. At the same time we are interested in a modular design, since we want to experiment with rewards, random goal positions, random mazes (polyominos) or even different particle models. In Section 5.3.1 we present our implementation of the basic maze environment and its modular components. In the following sections we then explain in detail how these components work, beginning with Section 5.3.2 where we will talk about reward generation. We then talk about extensions of the original environment to bring the simulations closer to reality with the addition of physical particles or unprecise observations in Section 5.3.3. Finally we will talk about random instance generation in Section 5.3.4.

### 5.3.1 Basic Implementation

We designed the implementation for the maze environment around two key ideas: First, the environment should run as fast as possible, because the agent will need to play hundreds or thousands of episodes. Designing a lightweight environment will therefore save a lot of time during training. Second, we need an environment which is configurable with our lab configuration file system in its main aspects. This means having the possibility to dynamically load instances, define rewards and even particle behavior. For the implementation we therefore decided to build the maze environment as a modular system, which allows easy configuration and extension of the existing architecture.

Figure 5.5 shows a simplified overview of the main components of the environment. The master class *MazeBase* handles basic functions like rendering of the environment

and provides an interface to the outside world. The internal operations however rely on the used components. The modules which can be integrated into the MazeBase class all inherit from three interfaces and therefore can be divided into three groups:

1. *Instance Generators.* Instance generators load or create instances. Depending on the generator, an instance can be just created once and then returned for every future episode (e.g. if we want to load a specific instance) or the instance can be randomly generated. We will talk more about random instance creating in Section 5.3.4.

2. *Reward Generator.* Reward generators create rewards after each step and may also end an episode. We will talk in detail about reward shaping in Section 5.3.2.

3. *Step Modifiers.* Step modifiers define the behavior of the particles. This behavior might be very simple like in the original maze implementation of Huang et al. where each particle moves exactly one step into the direction given by the action, but can be more complex. We will talk about advanced particle behavior in Section 5.3.3.

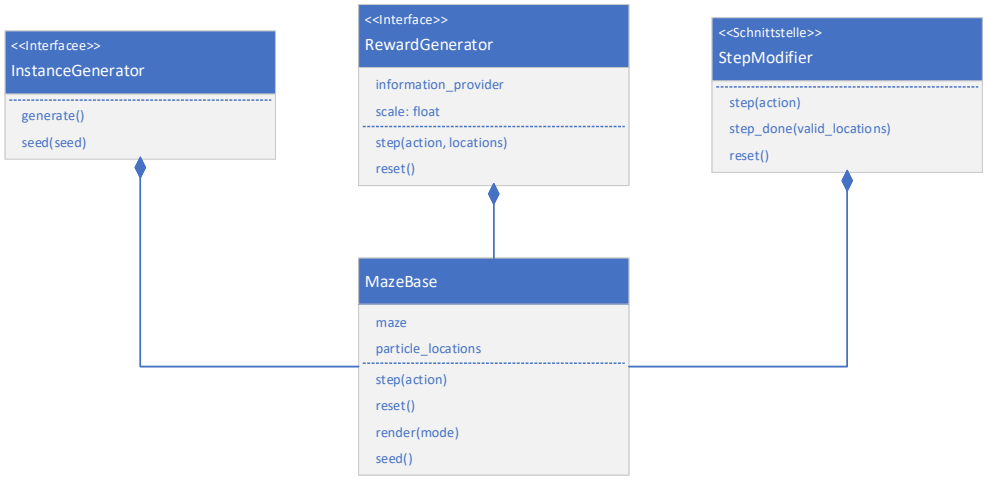



Figure 5.5: Implementation Design for the Maze Environment.

All components of the maze environment are generally designed with performance in mind. When implementing particle motions and reward generations, we minimized the use of python loop statements and instead solved most of the computations by utilizing NumPy [58] functions.

## 5.3.2 Reward Shaping

Rewards are a fundamental component of reinforcement learning. For RL algorithms the difficulty of a problem can largely depend on how much information the reward signal provides during training. Problems become especially complicated when dealing with very sparse rewards (e.g. just a single reward at the end of an episode). When designing

a reward signal for a problem, it is therefore desirable to include as much information as possible into the reward signal to obtain faster and better learning results. Providing much information with the reward signal can include a risk though: The reward must be unbiased and should not limit the solution space the algorithm can come up with. If we reward specific strategies, the RL algorithm will just replicate that strategy and not learn anything new.

Keeping in mind that we want to create an unbiased reward signal, we can use the simple distance to the goal position as a measure for the reward signal. Since pixel perfect navigation is hard for RL algorithms, we want to allow slight inaccuracy and define a small area around the exact goal position in which we accept the solution. We then can measure the distance to the goal position for each particle and create two distinct reward signals from this measure:

- *Total cost reward* is generated based on the sum of distances $d_{total}$ of all particles to the goal position. A positive total cost reward therefore indicates overall improvement. This reward may also be generated by calculating the average cost $d_{avg}$ instead of the total cost.
- *Max cost reward* is generated based on the particle with the greatest distance to the goal position $d_{max}$ only. This reward may differ significantly from the total cost reward, since it strongly punishes leaving behind single particles.

As we mentioned in Section 2.4, Huang et al. used both, the maximum and the total cost reward signal. Instead of giving a continuous reward signal, they defined 4-8 subgoals for both these values and only rewarded the agent if it was able to reduce the distance for the average or maximum particle below one of the subgoal thresholds. An example for this method is shown in Table 5.1.

| Max Cost | Reward | Completed |
|---|---|---|
| <10 | 8 | False |
| <20 | 8 | False |
| <40 | 4 | False |
| <80 | 4 | True |
| <120 | 2 | True |
| ⋮ | ⋮ | ⋮ |

Table 5.1: Example for the original reward system. By reaching a certain goal for a reward category (e.g. max cost) the agent is rewarded with a certain reward. Each reward can only be obtained a single time [36].

The sparseness of this reward signal may significantly increase the complexity of the problem. Also this reward is not flexible, since the reward subgoals were handcrafted for each individual instance. In order to investigate the impact of different rewards, we test

a number of combinations of our reward components and measure their impact on the performance in Section TODO. In the following we want to explain what the goal behind each reward component is and how we calculate them. Figure 5.6 provides an overview of the different components of our flexible reward system.

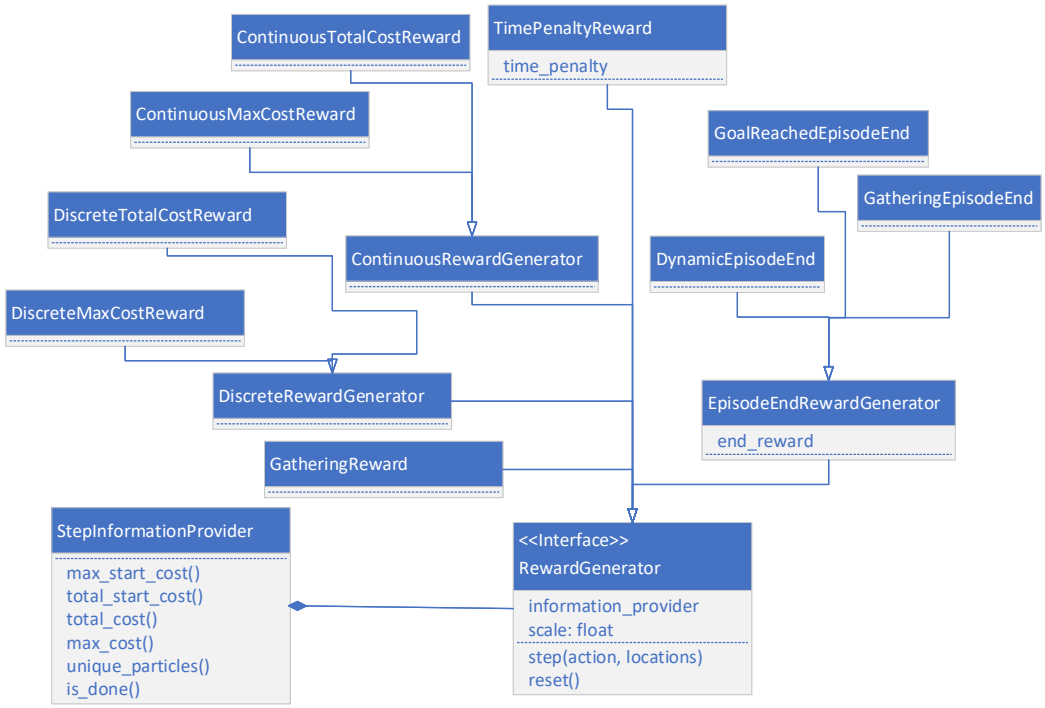



Figure 5.6: Implementation design for the reward generators. To improve efficiency, all reward components share an object called of the type *InformationStepProvider* which ensures that the same calculations are not repeated multiple times.

**Discrete Reward.** We implemented a reward which works similar to the original reward, but with a dynamic number of subgoals. We also drastically increased the number of subgoals, to make the reward less sparse. Instead of rewarding the agent at 4-8 subgoals (which in the original implementation were all given to the agent if its solution was already close to the goal position) we dynamically compute a number of subgoals depending on the size of the maze and distribute these subgoals evenly. This way the agent already receives reward at early stages of the training process. We estimate the number of subgoals by halving the maximum distance any particle can be away from the goal position. Subgoals which are easier to reach (the first subgoals) generate a smaller reward than later ones.

**Continuous Reward.** To reduce sparseness to a minimum, we also implemented continuous reward. The continuous reward does not define any subgoals and instead directly rewards the agent by the change in distances for both, the maximum and the total cost

metric. In this case it is important to either use the average cost metric or scale both the total distance metric and the maximum cost metric to $[0, 1]$, otherwise the weight of the total distance metric is much larger than the weight of the maximum cost metric. There is a second aspect to using scaled total distance reward though and that is reward consistency: Larger instances would automatically generate more reward if the reward is directly dependent on the distance between particles and the goal position. If we want to explore agents which can solve randomly generated instances, we need a reward which always has the same total sum over an episode. We call the the continuous reward normalized, if both the total cost and the maximum cost metric are scaled to $[0, 1]$ (so the reward is always bounded to $[0, 2]$).

One problem when creating the continuous reward signal only based on the change in the particle distance is, that the same observation does not always create the same amount of reward. Remember when we talked about the idea of a baseline in Section 4.3.2? The problem was, that in most situations a high or a low reward does not tell us if the agents performance is good, because the starting situation for the agent might be better or worse than average just by chance. Therefore modern RL algorithms already account for these fluctuations by estimating how good the current state is. For optimal performance, we therefore must design a reward signal, which is consistent through multiple executions for the same environment state - rewarding the agent even if it has just been lucky.

To achieve this, we need an estimate for the starting cost for both the maximum and total (or average) cost metric, which is independent of the actual particle positions. For the total cost metric, we can use the total distance of each possible particle position to the goal position and for the maximum cost metric we use the maximum distance a single particle can be away from the goal position. For each step of training, the continuous reward is then calculated by subtracting the current value of each metric from the corresponding value after the last step.

Generating totally continuous reward may discourage the agent to gather particles which got stuck at some point, if it has to move other particles away from the goal position, because it may receive large amounts of negative reward in the process. We therefore propose an extension to the continuous reward signal which cuts off negative rewards and returns zero reward for all steps which would otherwise result in a negative reward.

**Time Penalty.** While the basic reward does encourage the agent to bring all particles to the goal position, there is no pressure on the agent to use a minimum amount of time. It will receive the same amount of reward independent of the total time. To add a time constraint, we can add a time penalty to the reward signal, which is given out at each step. The original implementation used a static time penalty of $-0.1$ reward per step. With our dynamic reward system this static time penalty does not work optimally for two reasons: First, the ratio between time penalty and the rest of the reward signal may not be the same for all environments, since larger environments now provide more reward due to particles being further away (except for normalized continuous reward). Second, larger environments may

need more time to solve and - like before - we are interested in a constant total value for the time penalty. The problem here is that we do not know beforehand what the optimal episode length will be. We therefore propose to estimate the optimal episode length by

$$ep\_len \approx 0.75 * d_{max} * \log(d_{avg} * k)$$

Here $d_{max}$ is an estimate for the maximum distance between any two points of the maze, by using the maximum distance between a random extreme position and any other point in the maze. $d_{avg}$ is the average distance between any point and the goal position and $k$ is the number of convex corners. We experimentally found, that the constant factor 0.75 works best to estimate the number of steps needed to solve the environment across our test instances.

The time penalty is given by $ep\_len^{-1}$. For discrete or unnormalized continuous rewards, we scale this time penalty to match the sum of all rewards.

**Gathering Reward.** Bringing the particles to the goal position requires to gather the particles in a small area or even in a single pixel. We therefore create an additional reward signal which is generated by calculating the number of unique particle positions in the current step. This reward signal can also be normalized to $[0, 1]$ by dividing it by the initial number of particles. Note that gathering reward might also be more complex by calculating the total pairwise particle distance, but this calculation is extremely costly and we therefore did not use this option.

**Dynamic Episode Length.** When training an agent on the maze environment, we usually set an upper time limit to the maximum episode length. When training the agent it usually receives more and more reward over time until it is able to solve the environment and the episode length decreases. With the addition of curiosity reward, this produces a problem: Longer episodes are usually more interesting. This was observed in the RND paper [20] and during our verification experiment (see Section 5.2.2). This means, that adding a curiosity reward to the agent may encourage the agent to not find a solution which brings all particles on the shortest path to the goal position. This problem can be solved to some extend, by scaling down the curiosity reward, but the problem itself is still present.

There is also a second aspect when talking about the time limit. A high time limit may lead to training situations, where the agent gets stuck at some point and then is not able how to figure out how it should get out of the current situation. By resetting the environment the agent may find a strategy which completely avoids the situation from before. A high time limit therefore might impact the training performance negatively, if the agent gets stuck in a state and then has to wait until the episode is over and it can try again. On the other hand a very short time limit can also negatively impact training. The limit may be shorter than the shortest possible solution and a short time limit also decreases the possibility for exploration. We therefore need to estimate an ideal episode length.

When we talked about the time penalty, we already proposed a way to estimate the optimal episode length. Using this value as an upper limit would make sense, but we could also go one step further: Since we are dealing with non-sparse rewards, we could monitor the reward during an episode to detect situations in which the agent got stuck. We then dynamically cancel episodes which are not promising. To achieve this, we divide the estimated episode length by the number of expected positive rewards. In the case of discrete rewards, this is the number of subgoals. We then increase the time limit each time the agent is able to gain a positive reward. We also include a small bias, which gives the agent a little bit more time early on and a little bit less time the longer the episode has already been.

Using a dynamic episode length in this form may cause problems in combination with a time penalty. Since longer episodes result in more time penalty (meaning less total reward), the agent might try to end episodes as fast as possible. In the case of dynamic episode lengths a short episode means not receiving any "normal" reward at all. We therefore add two constraints which should resolve this problem: First, if the agent is not able to reach the goal, it will receive a negative reward equal to the time penalty it would receive if the episode had not been canceled early. Additionally, if the agent is able to finish the episode earlier than the estimated episode length, it receives a positive reward which is proportional to the number of steps its solution was faster than expected.

### 5.3.3 Extended Models

Previous work only looked at the particle navigation problem from a very theoretical point of view. Each particle could be perfectly observed meaning we always had information about its pixel perfect position. We also were able to perfectly apply the same uniform transformation to each particle and this transformation was a direct change of the position instead of a "real" movement. To investigate the performance of RL models under more real-life conditions, we therefore implemented two extensions which we evaluated one-by-one and combined in Section TODO.

**Introducing Error.** When it comes to real world applications, we always will have to deal with multiple sources of error. Sensors will always provide a certain amount of noise and actions might not be executed perfectly. To simulate various error sources we therefore introduce fuzzy observations and fuzzy actions.

In combination with noise, detecting the exact location of microscopic particles will always be inaccurate as particles may not be recognized correctly. We therefore generate fuzzy observations by using three possible sources of error:

- *Noise.* To simulate standard detection error, we add gaussian noise to the observation.
- *False Negative Detection.* At each step, each particle has a small chance to be not be included in the observation at all.

- *False Positive Detection.* At each step, particles with random positions may be added to the observation.

While particles in the theoretical model are directed using a uniform force, in reality particle movement will never be completely uniform. Slight changes in the magnetic field, collisions with other particles or fluctuations in the blood flow, may interfere with the navigation at any time. Also the generation of the magnetic fields may work with a slight delay and that delay might change over time. We therefore generate fuzzy actions using three sources of error.

- *Sticky Actions.* Each action has the probability to be executed again for the next step.
- *Noisy Actions.* Each action may only affect a certain percentage of the particles.
- *Random Actions.* Each particle may move randomly at each step.

We test all these sources of error and their influence on the performance in Section TODO.

**Physical Particles.** While error in actions and observations adds a lot to the realism of the simulation, we are still working with particles which directly change from one position to another. Instead we want to have particles which behave more like real particles. Instead of directly changing the particle position, we want to accelerate particles. Particles themselves then have an internal speed vector which also influences their position for the following steps of the simulation. To simulate friction we divide this speed vector by a constant value of 1.1 for each step and set the speed to zero if a particles' speed drops below a threshold. The particle position is rounded to match an integer position for the observation, but kept at floating point precision for internal calculations.

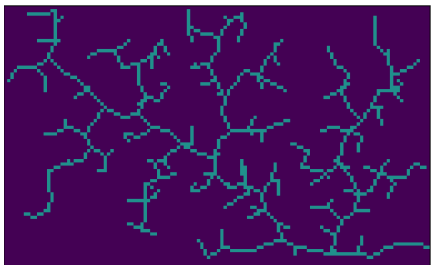
One problem when simulating physical particles is that the computations for the environment get more complicated and training needs more time. This is especially problematic when calculating collisions with the maze. For the non-physical particles, each particle could at most move a single pixel in any direction. If the particle was blocked by the maze in that direction it would remain at its current position. However with physical particles, particles may actually move more than one pixel per step. In case of a collision, we would need to calculate the exact point of contact between the particle and the maze. Instead we approximate the collision by halving the speed if the particle would cause a collision at full speed. The particle then does not move for the current step, but will keep half of its velocity for the next step, getting closer and closer to the wall. Note that this procedure might allow particles to move through walls if their velocity is large enough to cross the wall in a single step. However the size of our test instances does not allow the agent to accelerate particles to this point.

We investigate the performance of RL algorithms when dealing with physical particles in Section TODO.

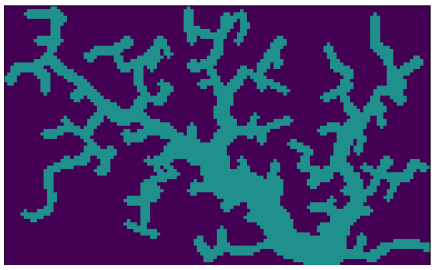
### 5.3.4 Random Instance Generation

To train agents to solve randomized mazes, we need a generator which produces blood vessel like structures. To generate these structures we used an algorithm called *rapidly-exploring random tree* (RRT) [45]. The algorithm was originally designed to efficiently search high-dimensional spaces by randomly generating a space-filling tree. The tree is generated by randomly proposing new points in the search space and connecting these new points to the closest point of the search tree. If the random point is to far away from the closest point of the search tree, a new node is inserted at a predefined maximum length in the direction of the random point. Using the RRT algorithm, we create blood-vessel like instances in four steps:

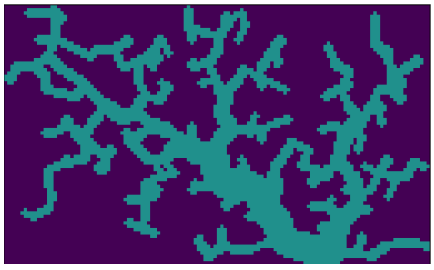
1. Generate a new RRT. A low bound on the maximum length for each new segment ensures a river-like structure of the tree. Figure 5.7 (a) shows an example RRT tree with 250 nodes.
2. We calculate the *flow* to the starting node of the tree. Each leaf node generates a flow of one and propagates that flow back to the root node of the tree (see Figure 5.7 (b)).
3. Create random loops in the tree, by generating a set amount of random points and connecting them to the two closest nodes of the tree. (see Figure 5.7 (c))
4. Draw the tree, using the square root of the flow value as width for each segment.



(a) A simple RRT tree.



(b) Flow-based branch scaling.



(c) Adding random non-intersecting loops.

Figure 5.7: Creating vessel-like random instances with the RRT algorithm.

When training with randomly created instances a goal position is randomly chosen from the set of empty pixels. Additionally we buffer the randomly created instances and choose

an already created instance with a probability of 95% and only generate a new instance with a probability of 5%. This avoids expensive recomputation at every environment reset.

### 5.3.5 Integration of Algorithmic Approaches

To allow easy evaluation, we integrated an adapter which allows algorithmic strategies, which precompute all particle movements, to interact with our step-based environment and replay their decisions accordingly. By adapting code from [12] we have access to the SSP, DSP and MTE algorithms as shown in Section 2. Evaluation results comparing the RL approach against the algorithmic approaches can be found in Section TODO.

# Bibliography


[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] J. Achiam, D. Held, A. Tamar, and P. Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 22–31. JMLR. org, 2017.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna documentation. `https://optuna.readthedocs.io/en/stable/tutorial/index.html`, 2018.

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[5] K. E. Albinali, M. M. Zagho, Y. Deng, and A. A. Elzatahry. A perspective on magnetic core–shell carriers for responsive and targeted drug delivery systems. *International journal of nanomedicine*, 14:1707, 2019.

[6] S. Alpern and S. Gal. *The theory of search games and rendezvous*, volume 55. Springer Science & Business Media, 2006.

[7] S.-I. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[8] E. J. Anderson and S. P. Fekete. Two dimensional rendezvous search. *Operations Research*, 49(1):107–118, 2001.

[9] K. Arulkumaran, A. Cully, and J. Togelius. Alphastar: An evolutionary computation perspective, 2019.

[10] J. Ba, R. Grosse, and J. Martens. Distributed second-order optimization using kronecker-factored approximations. 2016.

[11] A. T. Becker, S. P. Fekete, P. Keldenich, D. Krupke, C. Rieck, C. Scheffer, and A. Schmidt. Tilt assembly: Algorithms for micro-factories that build objects with uniform external forces. *Algorithmica*, pages 1–23, 2018.

[12] A. T. Becker, S. P. Fekete, L. Huang, P. Keldenich, L. Kleist, D. Krupke, C. Rieck, and A. Schmidt. Targeted Drug Delivery: Advanced Algorithmic Methods for Collecting a Swarm of Particles with Uniform, External Forces. 2020.

[13] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

[14] R. Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[15] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[16] S. Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 15, 2016.

[17] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[18] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[19] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*, 2018.

[20] Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[21] R. Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[22] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[23] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[24] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[25] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[26] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[27] P. S. Foundation. Python language reference. `www.python.org/`, 2020.

[28] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[29] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2nd edition, September 2019.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[32] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[33] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines documentation. `https://stable-baselines.readthedocs.io`, 2020.

[34] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[35] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[36] L. Huang. *Towards Microrobot Swarm Path Planning*. PhD thesis, University of Housten, 2019.

[37] D. H. Hubel. Single unit activity in striate cortex of unrestrained cats. *The Journal of physiology*, 147(2):226–238, 1959.

[38] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[39] S. M. Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[40] Z. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.

[41] W. L. Keng and L. Graesser. Slm lab. `https://github.com/kengz/SLM-Lab`, 2017.

[42] W. L. Keng and L. Graesser. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Professional, December 2019.

[43] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[45] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[47] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[48] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[49] A. V. Mahadev, D. Krupke, J.-M. Reinhardt, S. P. Fekete, and A. T. Becker. Collecting a swarm in a grid environment using shared, global inputs. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1231–1236. IEEE, 2016.

[50] J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417, 2015.

[51] J.-B. Mathieu and S. Martel. Magnetic microparticle steering within the constraints of an mri system: proof of concept of a novel targeting approach. *Biomedical microdevices*, 9(6):801–808, 2007.

[52] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[53] L. Mellal, D. Folio, K. Belharet, and A. Ferreira. Magnetic microbot design framework for antiangiogenic tumor therapy. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1397–1402. IEEE, 2015.

[54] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[55] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[56] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[57] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1054–1062, 2016.

[58] T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[59] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[61] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.

[62] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter space noise for exploration, 2017.

[63] P. Pouponneau, J.-C. Leroux, and S. Martel. Magnetic nanoparticles encapsulated into biodegradable microparticles steered with an upgraded magnetic resonance imaging system for tumor chemoembolization. *Biomaterials*, 30(31):6327–6332, 2009.

[64] A. Raffin. Rl baselines zoo. `https://github.com/araffin/rl-baselines-zoo`, 2018.

[65] A. Raffin, M. Konitzny, A. Hill, and A. Kanervisto. Stable baselines issue 309 intrinsic reward vecenvwrapper. `https://github.com/hill-a/stable-baselines/issues/309`, 2020.

[66] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[67] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[68] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[69] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[70] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015.

[71] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[72] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[73] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[74] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. [Online; accessed 25-June-2019].

[75] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[76] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[77] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[78] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[79] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[80] G. Van Rossum and F. L. Drake. *The python language reference manual*. Network Theory Ltd., 2011.

[81] T. Vangijzegem, D. Stanicki, and S. Laurent. Magnetic iron oxide nanoparticles for drug delivery: applications and characteristics. *Expert opinion on drug delivery*, 16(1):69–78, 2019.

[82] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[83] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[84] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[85] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[86] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288, 2017.

[87] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.