Titel der Arbeit

Bachelorarbeit der Fakultät für Physik
der
Ludwig-Maximilians-Universität München

vorgelegt von
Jonas C. Märtens
geboren in Hamburg

München, den 01.02.2024

Contents

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Chapter 1

Introduction

1.1 Artificial Intelligence

Artificial Intelligence (AI) is a broad subject of study that can be defined in different ways [1, chapter 1]. John McCarthy, often called the “father of AI” [2, 3, 4], defines it as “the science and engineering of making intelligent machines” [5], where intelligence means “the computational part of the ability to achieve goals in the world” [5]. AI is sometimes mistakenly used interchangeably with Machine Learning. Machine learning is a subset of AI concerned with enabling AI systems to learn from experience [1, chapter 1]. Machine learning enables the development of large-scale AI systems as they are used today.
The study of artificial intelligence was first proposed by McCarthy et al. in late 1955 [6]. It went through two major hype cycles in the sixties and the eighties [7, 2, 8] followed by phases of “AI winter”. The current (as of early 2024) AI boom, sometimes also called “AI spring” [9] was started by groundbreaking advances in speech recognition [10] and image classification [11] in 2012 [12, 13] and reached the public at the latest in late 2022, following the release of ChatGPT [14], a multipurpose AI-chatbot, open to everyone [15].
These breakthroughs are made possible mainly by advancements in the field of machine learning, enabling AI systems to learn from huge amounts of data. In addition, the exponential increase in computation and storage capabilities as predicted by Moore’s Law [16], algorithms like backpropagation [17] allowed incorporating large amounts of data into machine learning models in realistic amounts of time.
Today, AI systems are indispensable in many areas such as web search engines [18], recommendation systems [19], human speech recognition and generation [20, 10], image recognition and generation [21, 11] and personal assistants [14] and surpasses humans in high level strategy games like go and chess [22, 23] as well as other video games [24].

1.2 Intelligent Matter, Machine Learning and Physics

Classical statistical physics is mostly concerned with systems in thermodynamic equilibrium. It studies ensembles of equilibrium systems to derive macroscopic properties from microscopic interactions. Non-equilibrium statistical mechanics studies systems that are not in thermodynamic equilibrium but are instead driven by imbalances [25]. Simple examples include the physical description of chemical reactions or systems with a steady flow of heat or particles between two baths. In nature, most systems are far from equilibrium [26]. These systems cannot be described by classical statistical mechanics and require new theoretical frameworks [26]. Almost all living systems fall under this category: From microscopic objects such as molecular motors, living cells and bacterial colonies to macroscopic objects such as flocks of birds or schools of fish [27, 26], all living systems consist of many interacting parts that draw energy from their local environment to perform mechanical work [28]. These components can be simple on their own, but when they interact with each other, they can form complex structures and patterns and often exhibit highly intelligent behavior [28, 29].

The study of these systems is called *active matter* research [29]. Active matter describes any kind of matter that exhibits dynamic behavior and can adapt to external stimuli [29]. It can be further distinguished from *intelligent matter*, which is a subset of active matter that learns from and interacts with its environment [29]. Active matter research is a relatively new field of study, with the popularity of the subject beginning to rise in the mid-1990s [28].

Modeling and replicating such systems would be of great scientific interest, as it could lead to a quantitative, mathematical theory of still-mysterious biological processes [28], but it also has great potential for real-world applications in fields such as robotics, material science and computational research. Groups of intelligent robots such as the "Kilobots" introduced by Rubenstein et al. [30] can work together to complete tasks that would be impossible for a single robot, similar to how a colony of ants builds nests and transports food [30]. Intelligent matter could also be used to create adaptive artificial skins [29] or smart clothing that changes its insulation properties or transparency based on the environment [31]
Another interesting aspect of active matter systems that was also one of the initial motivations for research in this field is the computational efficiency of these systems [29]. The human brain manages to perform complex cognitive tasks while consuming only a small amount of energy compared to modern computers [32]. Intelligent matter research aims to replicate nature's way of information processing to find new, more efficient ways of computing [29].

Machine learning can be used in different ways to study active matter systems. For one, it can be used to analyze data from active matter experiments. Applications include tracking objects in active matter data [27], analyzing time series data [27] or finding connections between genetic code and emergent bacterial behavior [27].
Another interesting application of machine learning in active matter research is the use of machine learning to simulate new forms of active matter. Oftentimes, the rules that govern the behavior of the individual components of an active matter system are very simple, but the interactions between these components can lead to complex behavior that is hard to predict. Machine learning can be used to explore a new form of active matter where the large number of components is combined with intelligent behavior already at the microscopic level. While systems with a large number of simple components, such as the Kilobots or DNA molecules, have been studied previously, and the same is true for systems with a small number of very intelligent components, such as self-driving cars or Boston Dynamics' "Spot" [33, 34], this combination of a large number of intelligent components is a fairly new field of study in active matter research and especially in physics.

While many aspects of collective human behavior are studied by the field of sociology, and studies exist on the collective behavior of animals such as the flocking of birds, the schooling of fish and krill and herding of mammals [35], the topic is still relatively unexplored in physics. A straightforward way of studying collective behavior in active matter systems is to use agent-based modeling (ABM). In ABM, the individual units of a system (agents) are modeled and simulated to capture the emergent phenomena arising from their interaction when the simulation is run [36]. This is often easy to implement, while enabling a rich description of the system [36]. For many systems, ABM is the most natural way of modeling them, as it recreates complex behavior from the bottom up. When modeling traffic jams or supermarket shoppers for example, it is straightforward to model the individual cars or shoppers and their interactions, but it would be very difficult to model the top-level dynamics of the system directly [36].

A popular example of systems where highly complex behavior emerges from simple rules are cellular automata like Conway's Game of Life [37, 38], which has even been shown to be Turing-complete [39]. These systems typically consist of a grid of cells that can be in a finite number of states (e.g. occupied or unoccupied) and a set of rules that determine the state of each cell in the next time step based on the state of the cell and its neighbors in the current time step. Conways Game of Life for example has only four basic rules, but new patterns and structures are still being discovered today, more than 50 years after its invention [40].
This thesis will use a similar grid-based system, but instead of defining the rules of the system manually, agents in the system will learn an optimal set of rules using deep reinforcement learning with neural networks, also allowing different agents to learn different rules. This is a fairly new approach in the study of active matter systems.

1.3 Stochastic Processes and Markov Chains

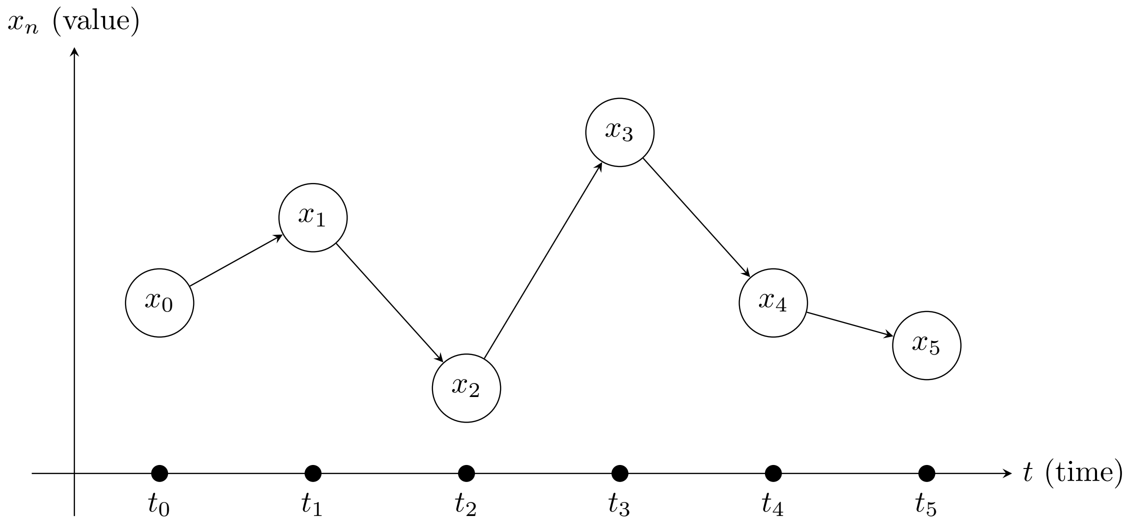


Figure 1.1: A stochastic process is an ordered collection of random variables x_t indexed by time t.

This section will briefly lay out the mathematical framework of stochastic processes and Markov chains. Section 3.1.3 will later build on this framework to explain the mathematical foundations of reinforcement learning using Markov decision processes. The derivations in this chapter closely follow [41, chapter 2.2].

A *stochastic process* is an ordered sequence of values $x_n = x(t_n)$ of a random variable $x \in \mathcal{S}$ indexed by (possibly discrete) time $t_n \in \mathcal{T}$ (Fig. 1.1). The probability of observing such a sequence of n values is given by the *n-point probability distribution*

$$P_n : \mathcal{S}^n \times \mathcal{T}^n \to [0,1] \tag{1.1}$$

$$(x_1, t_1; x_2, t_2; \ldots; x_n, t_n) \mapsto P_n(x_1, t_1; x_2, t_2; \ldots; x_n, t_n), \tag{1.2}$$

which is normalized:

$$\int dx_1 \cdots \int dx_n P_n(x_1, t_1; x_2, t_2; \ldots; x_n, t_n) = 1, \tag{1.3}$$

and obeys the *hierarchy rule*

$$\int dx_n P_n(x_1, t_1; x_2, t_2; \ldots; x_n, t_n) = P_{n-1}(x_1, t_1; x_2, t_2; \ldots; x_{n-1}, t_{n-1}). \tag{1.4}$$

Time-averages, moments and correlations are defined in the usual way:

$$\langle x(t)^n \rangle = \int dx x^n P_1(x, t), \tag{1.5}$$

$$\langle x(t_1) \ldots x(t_n) \rangle = \int dx_1 \cdots \int dx_n x_1 \ldots x_n P_n(x_1, t_1; \ldots; x_n, t_n). \tag{1.6}$$

The same goes for conditional probabilities:

$$P_{m|k}(x_{k+1}, t_{k+1}; \ldots; x_{k+m}, t_{k+m} | x_1, t_1; \ldots; x_k, t_k) = \frac{P_{m+k}(x_1, t_1; \ldots; x_{k+m}, t_{k+m})}{P_k(x_1, t_1; \ldots; x_k, t_k)}. \tag{1.7}$$

A stochastic process is called *stationary* if its n-point probability distribution is invariant under time translations:

$$P_n(x_1, t_1; \ldots; x_n, t_n) = P_n(x_1, t_1 + \tau; \ldots; x_n, t_n + \tau). \tag{1.8}$$

A sequence $x_{k+1}, x_{k+2}, \ldots, x_{k+n}$ can be *statistically independent* of the preceding sequence $x_1, x_2, \ldots, x_k$:

$$P_{n|k}(x_{k+1}, t_{k+1}; \ldots; x_{k+n}, t_{k+n} | x_1, t_1; \ldots; x_k, t_k) = P_n(x_{k+1}, t_{k+1}; \ldots; x_{k+n}, t_{k+n}). \tag{1.9}$$

In that case, the n-point probability distribution factorizes:

$$P_n(x_1, t_1; \ldots; x_{k+n}, t_{k+n}) = P_k(x_1, t_1; \ldots; x_k, t_k) P_n(x_{k+1}, t_{k+1}; \ldots; x_{k+n}, t_{k+n}). \tag{1.10}$$

In the special case where the value of the random variable x at time t_{k+1} only depends on the value of x at time t_k, the process is called *Markovian* and the process is called a *Markov process* or *Markov chain*. In that case, the conditional probability distribution simplifies to:

$$P_{1|n-1}(x_n, t_n | x_1, t_1; \ldots; x_{n-1}, t_{n-1}) = P_{1|1}(x_n, t_n | x_{n-1}, t_{n-1}). \tag{1.11}$$

Markov Chains are fully characterized by their initial probability distribution $P_1(x_1, t_1)$ and their *transition probabilities* $P_{1|1}(x_{j+1}, t_{j+1} | x_j, t_j)$. Using the defintion of conditional probability (eq. 1.7) and the Markov property (eq. 1.11), one can derive the *Chapman-Kolmogorov equation* [41, page 59]:

$$P_{1|1}(x', t' | x, t) = \int d\bar{x} P_{1|1}(x', t' | \bar{x}, t') P_{1|1}(\bar{x}, t' | x, t), \tag{1.12}$$

which says that the transition from x to x' can be split into an arbitrary number of intermediate steps that only depend on the last step.
For continous-time Markov chains, one can derive the *master equation* [41, page 60]:

$$\frac{\partial}{\partial t} P_1(x, t) = \int d\bar{x} \left[P_1(\bar{x}, t) W(\bar{x} \to x, t) - P_1(x, t) W(x \to \bar{x}, t) \right], \tag{1.13}$$

where we introduced the *transition rate* $W(x \to \bar{x}, t)$ from x to $\bar{x}$ at time t. When we interpret the values of x as the states of a system (as we will do in section 3.1.3, calling them *states* s), the master equation can be interpreted as the balance of *gain* and *loss* of the probability to be in state x. Probability is gained when the system transitions from any other state $\bar{x}$ to x and lost when the system transitions from x to any other state $\bar{x}$.

One could continue building this powerful framework of stochastic processes and Markov chains by showing that all important properties of the probability distribution are preserved under time evolution and by introducing relevant quantitative measures such as the time-dependent *entropy* and their respective evolutions as is done in [41, chapter 2.2]. However, this is beyond the scope of this thesis as the reinforcement learning problem treated in this thesis builds on the framework of discrete-time Markov decision processes, which will be introduced later in section 3.1.3.

Chapter 2

Neural Networks

2.1 Overview

At the heart of almost all the technologies mentioned in section 1.1 are deep artificial neural networks. The next section will outline the mathematical details of how these systems work and learn. Although the comparison of artificial neural networks, from now on just called "neural networks", to their biological counterpart can be criticized as oversimplifying the inner workings of biological brains [42, chapter 1.1], the architecture of neural networks is heavily inspired by how decision-making and learning work in the human brain [42, 43, chapter 1.1]. I will illustrate the basic principles of neural networks at the example of a network that detects the gender of a person by looking at pictures.

A neural network consists of a number of layers of artificial neurons, called *nodes*. In a *fully connected* network, each node is connected to every node in the next layer. The connection strengths are called *weights*. An image of a person can be fed into the network by setting the *activation* values of the first layer of the network, the *input layer*, to the individual pixel values of the image. This information is then fed forward through the layers of the network until the *output layer* is reached. If a network has at least one layer in between the input and output layer, it is called a *deep neural network*. These intermediate layers are called *hidden layers*. If the outputs of each layer are only connected to the inputs of the next layer, the network is called a *feedforward neural network*. If *feedback* connections are allowed, the network is called a *recurrent neural network*.
In our example, the output layer should consist of only two nodes. If the activation of the first node is larger than the activation of the second node, the network thinks that the person in the picture is a male. If on the other hand the second node has a larger activation, the network classifies this person as female [42, chapter 1.2].

In order to make accurate predictions, a reasonable set of network parameters (i.e.the weights) has to be found. This is done by training the network with pre-classified images. After an image has been processed by the network, the output is compared to the correct classification and the network parameters are updated in a way that would improve the networks output if the same image was to be processed again [42, 44, chapter 1.2].
This is similar to how humans learn from experience. If we were to misclassify a persons gender, the unpleasant social experience that may come with that mistake would cause us to update our internal model of what different genders look like to not make the same mistake again.

One of the main strengths of neural networks is their ability to *generalize* [45]. When a network was trained on a large enough set of examples, it gains the ability to generalize this knowledge to examples that were previously unseen. The gender classification network from our example doesn't just memorize the genders of the people it has seen, but instead learns about the features that help to identify the gender of a random person.

The problem of image classification is a rather complex one. One wouldn't typically think of it as finding a function that maps the values of each input pixel to the classification output. But even very complex problems can be modeled by equally complex functions. The universal approximation theorem states, that a feedforward neural network with at least one hidden layer with appropriate activation functions (see 2.2.3 for details) can approximate any continuous function if given enough nodes [46, chapter 6.4.1]. That's why training a neural network can be thought of as fitting the network to the training data.

2.2 Mathematical Details

The following section will outline the mathematical details of how neural networks work. The definitions and derivations are based on [42, chapter 1.2-1.3], [46, chapter 5-6], [44] as well as [47, chapter 4.4].

The most complete treatment of the mathematical details of neural networks is arguably given by the framework of computational graphs [48]. In this framework, a neural network is treated as single that maps a set of input values to a set of output values. This function is composed of individual mathematical operations and can be represented as a directed graph [47, section 1.4]. I will not rigorously define the framework of computational graphs here, as this is beyond the scope of this thesis. Instead, I will explain the principles of neural networks starting from a single neuron and then build up to a fully connected neural network.

2.2.1 Notation

The following notation will be used throughout this section:

$\boldsymbol{x}$	:	input vector of the neural network
$\boldsymbol{a}$	:	activation vector of a layer
$\boldsymbol{z}$	:	pre-activation vector of a layer
$\boldsymbol{y}$	:	output vector of the neural network
$\boldsymbol{b}$	:	bias vector of a layer
x_i, a_i, z_i, y_i, b_i	:	individual elements of the respective vectors, for individual nodes
$\boldsymbol{w_i}$	:	weight vector of all weights connected to neuron i
w_{ij}	:	weight of the connection from neuron i of a layer to neuron j of the previous layer
W	:	weight matrix of a layer. Contains rows $\boldsymbol{w_i}$
$^{[J]}$	:	superscript denoting the layer of a variable

2.2.2 Single Node

Before we can build a neural network out of nodes, we have to define how a single node works. Each node receives the activations a_i from the nodes of the previous layer as inputs. Each connection is assigned a weight w_i, stored in the node's weight vector $\boldsymbol{w}$.
Additionally, each node has a so-called *bias b*. The bias shifts the net input of the node by a constant value. It is needed to model certain problems where part of the prediction is independent of the input [42, p. 6]. Examples include all problems where the output should not be zero even if all inputs are zero.

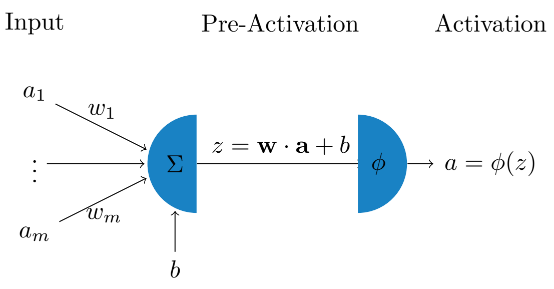


Figure 2.1: A single node of a neural network. To get the activation a of the node, the pre-activation z is calculated from the inputs a_i and the bias b and is passed through an activation function ϕ.

The net input, called *pre-activation value* z of a node is the weighted sum of all inputs plus the bias:

$$z = \sum_{i=1}^{m} w_i a_i + b = \boldsymbol{w} \cdot \boldsymbol{a} + b. \tag{2.1}$$

The pre-activation value is then passed through an *activation function* ϕ to get the activation a of the node, which is then passed on to the next layer, where the process is repeated:

$$a = \phi(z). \tag{2.2}$$

The whole process is illustrated in figure 2.1.

2.2.3 Activation Functions

The activation function ϕ is used to introduce non-linearity into the network and thus increasing its modeling power [42, section 1.2.1.3]. Some activation functions are also referred to as *squashing function* [47, p. 10], as they map the unbounded pre-activation value z to a bounded activation value a. The choice of activation function has a large impact on the performance of the network in terms of both accuracy and speed [49]. The type of function heavily influences the way that information is processed by the network and the complexity of the function naturally has a large impact on the computational cost of the network. The best choice therefore depends on the problem that is being solved and the architecture of the network. Typically, the same activation function is used for all nodes in a layer and is applied to the pre-activation value of each node individually, but different layers can use different activation functions depending on their purpose [46, p. 174]. For a long time, the most popular activation functions were ([46, chapter 6.3], [42, section 1.2.1.3]) the sigmoid function:

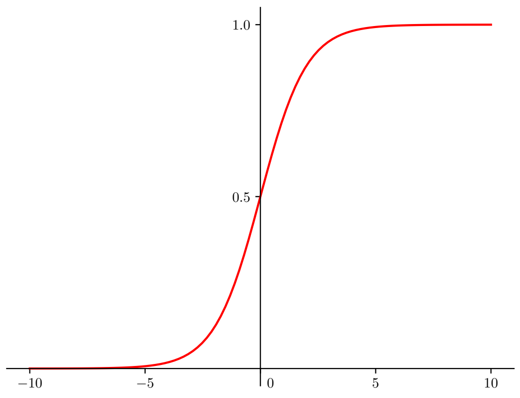
$$\phi(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

and the hyperbolic tangent function:

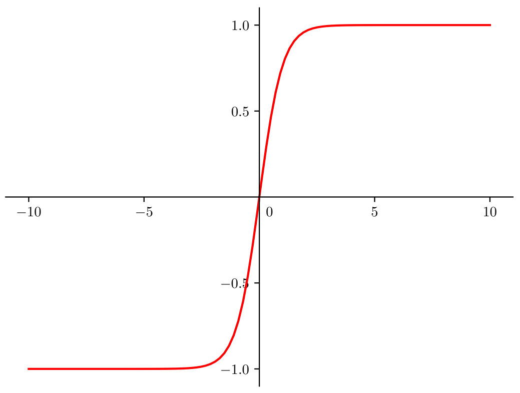
$$\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{2.4}$$

as well as the sign function:

$$\phi(z) = \operatorname{sign}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0. \end{cases} \tag{2.5}$$



(a) The sigmoid activation function.



(b) The hyperbolic tangent activation function.

Figure 2.2: The most popular activation functions before the rise of ReLU.

The sign function can map neural network output to a binary classification, but it is not suitable for backpropagation (see section 2.2.6) due to its derivative being zero almost everywhere. The sigmoid function and the hyperbolic tangent function are both differentiable and limit the output to the range $(-1, 1)$ and $(0, 1)$ respectively. They are however more computationally expensive than other activation functions and suffer from the *vanishing gradient problem*. The vanishing gradient problem is caused by the fact that the derivative of the sigmoid function approaches zero for large absolute values of z. This leads to the weights of the nodes in the first layers of the network being updated very slowly, as the gradient of the loss function with respect to the weights of these nodes is very small (see section 2.2.6)[42, section 1.4.2][49]. The sigmoid function and the hyperbolic tangent function can be seen in figure 2.2a and 2.2b.

In recent years, the *rectified linear unit* (ReLU) and similar stepwise linear functions have become the go-to activation functions for deep neural networks [46, chapter 6.3.2][49]. ReLU is defined as:

$$\phi(z) = \max(0, z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0. \end{cases} \tag{2.6}$$

Its main advantage is its very low computational cost, as it consists of only a single comparison. Although it is not as prone to the vanishing gradient problem as the sigmoid and the hyperbolic tangent, the problem still exists for negative values of z. This has been addressed by variations like *Leaky ReLU*, introducing a small but non-zero slope for negative values [49]:

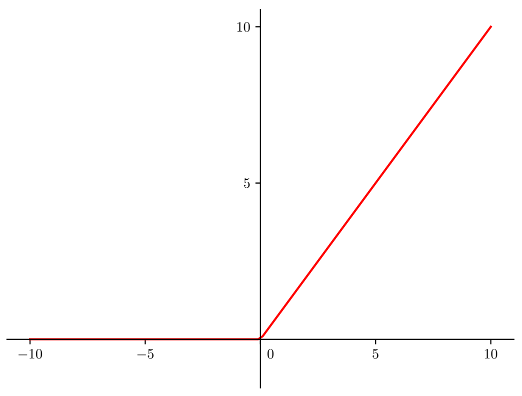
$$\phi(z) = \max(0.01z, z) = \begin{cases} 0.01z & \text{if } z \leq 0 \\ z & \text{if } z > 0. \end{cases} \tag{2.7}$$

The ReLU function and its leaky version can be seen in figures 2.3a and 2.3b.

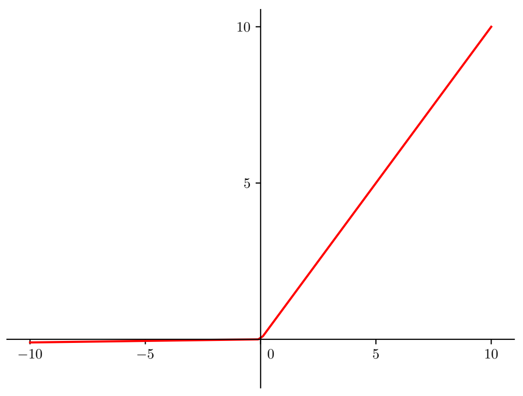
2.2.4 Forward-Propagation in Feedforward Neural Networks

Now that the workings of the nodes are defined, we can build a fully connected neural network out of these building blocks. I will illustrate the process at the example of a feedforward neural network as defined in section 2.1. An example of such a network can be seen in figure 2.4.

The process of feeding an input vector $\boldsymbol{x}$ through the network to get the output vector $\boldsymbol{y}$ is called *forward-propagation*, as the information propagates through the network layer. The



(a) The rectified linear unit (ReLU) activation function.



(b) The leaky rectified linear unit (Leaky ReLU) activation function.

Figure 2.3: Modern, stepwise linear activation functions.

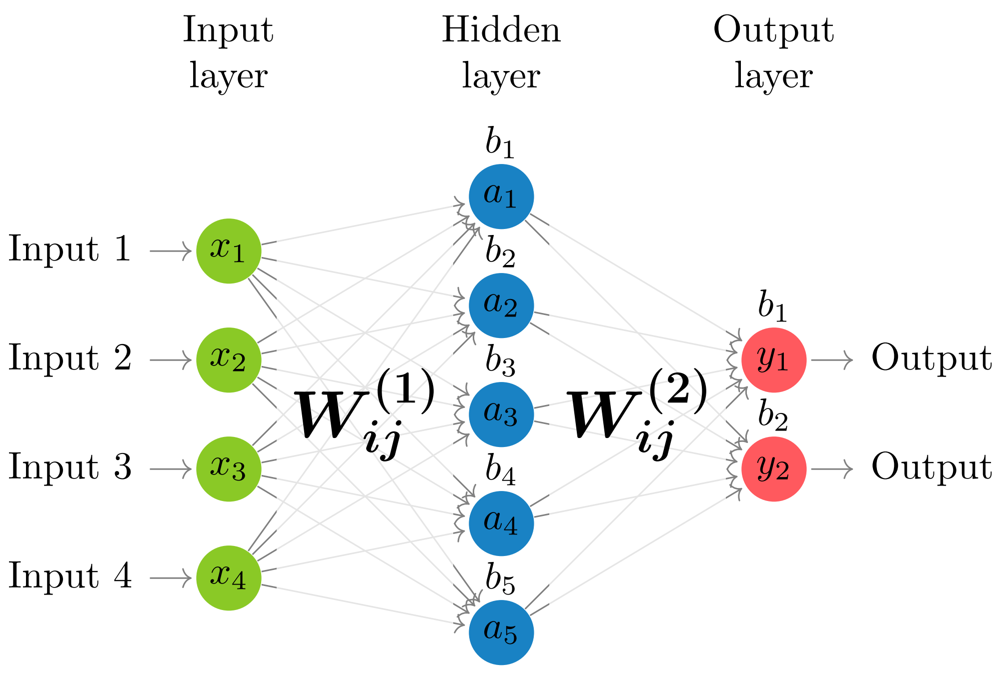


Figure 2.4: A feedforward neural network with one hidden layer.

activation of any layer $\boldsymbol{a}^{[J]}$ can be calculated from the activation of the previous layer $\boldsymbol{a}^{[J-1]}$ by generalizing equation 2.1 and 2.2 to the vector case:

$$z_k^{[J]} = \sum_{i=1}^{m} w_{ki}^{[J]} a_i^{[J-1]} + b_k^{[J]} \implies \boldsymbol{z}^{[J]} = W^{[J]}\boldsymbol{a}^{[J-1]} + \boldsymbol{b}^{[J]}, \tag{2.8}$$

$$a_k^{[J]} = \phi(z_k^{[J]}) \implies \boldsymbol{a}^{[J]} = \phi(\boldsymbol{z}^{[J]}), \tag{2.9}$$

where the activation function $\phi : \mathbb{R}^m \to \mathbb{R}^m$ is applied element-wise. The activation of the input layer $\boldsymbol{a}^{[0]}$ is simply the input vector $\boldsymbol{x}$ and the activation of the output layer $\boldsymbol{a}^{[L]}$ is the output vector $\boldsymbol{y}$.

Equations 2.8 and 2.9 can be applied recursively to calculate the activation of each layer from the input layer to the output layer:

$$\begin{aligned}
\boldsymbol{y} = \boldsymbol{a}^{[L]} &= \phi(W^{[L]}\boldsymbol{a}^{[L-1]} + \boldsymbol{b}^{[L]}) && (2.10)\\
&= \phi(W^{[L]}\phi(W^{[L-1]}\boldsymbol{a}^{[L-2]} + \boldsymbol{b}^{[L-1]}) + \boldsymbol{b}^{[L]}) && (2.11)\\
&= \phi(W^{[L]}\phi(W^{[L-1]}\phi(\dots\phi(W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]})\dots) + \boldsymbol{b}^{[L-1]}) + \boldsymbol{b}^{[L]}). && (2.12)
\end{aligned}$$

This equation also shows the significance of the activation function. Without it, the whole network would be equivalent to a single layer and could be replaced by a single matrix multiplication. It would therefore only be able to model linear functions. To show this, let's set $\phi(z)$ to the identity in equation 2.12. This yields:

$$\begin{aligned}
\boldsymbol{y} &= W^{[L]}W^{[L-1]}\dots W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[L]} + W^{[L]}\boldsymbol{b}^{[L-1]} + \dots + W^{[L]}W^{[L-1]}\dots W^{[2]}\boldsymbol{b}^{[1]} && (2.13)\\
&= \widetilde{W}\boldsymbol{x} + \tilde{\boldsymbol{b}}. && (2.14)
\end{aligned}$$

2.2.5 Loss Functions and Gradient Descent

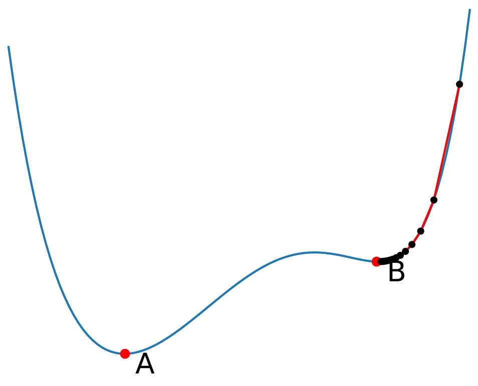
In order to produce meaningful output, the network's weights and biases have to be adjusted to minimize the error of the network's output. This process is called *training* the network. The error of the network is measured by a *loss function* $\lambda(\boldsymbol{y}, \hat{\boldsymbol{y}})$, where $\hat{\boldsymbol{y}}$ is some target output vector. The loss function is a measure of how far the network's output $\boldsymbol{y}$ is from the target output $\hat{\boldsymbol{y}}$. As we want to minimize the network's error, the training process is essentially an optimization problem [46, chapter 4.3]. Let's step back from neural networks for a moment and look at a method to minimize a function $f(\boldsymbol{x})$ with respect to its parameters $\boldsymbol{x}$. The most common method to do this in machine learning is *gradient descent* [46, chapter 4.3].

If we imagine the function $f(\boldsymbol{x})$ as a landscape, the goal of gradient descent is to find the lowest point of the landscape. To do this, the algorithm starts at some point $\boldsymbol{x}_0$ and in each iteration, it takes a step in the direction of the steepest descent. The size of the step is determined by the *learning rate* η. Choosing an appropriate learning rate is crucial for the algorithm to converge. Figure 2.5 shows gradient descent with different learning rates and the resulting paths through the landscape.

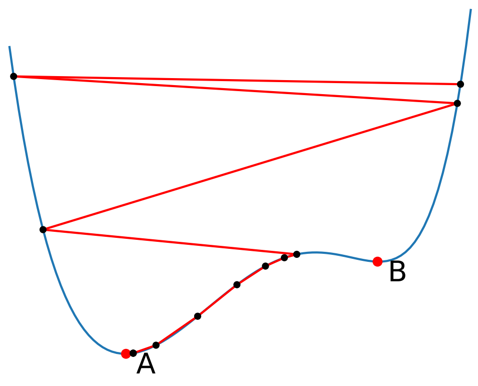
As is known from multivariable calculus, the direction of the steepest descent of a function $f(\boldsymbol{x})$ is given by the negative gradient $\nabla f(\boldsymbol{x})$. We can therefore update the parameters $\boldsymbol{x}$ in each iteration by:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta\nabla f(\boldsymbol{x}_n). \tag{2.15}$$

After a sufficient number of iterations, the algorithm will converge to a local minimum of the function. In the region around the minimum, the gradient is close to zero and the algorithm



(a) Gradient descent with a small learning rate.



(b) Gradient descent with a large learning rate.

Figure 2.5: Gradient descent with different learning rates. Choosing the learning rate too small can lead to slow convergence and to being stuck in local minima. Choosing the learning rate too large can lead to *overshooting* and to the algorithm diverging. Here, the algorithm still converges, but the large oscillations slow down the convergence.

Algorithm 1 Gradient Descent

```
Require:
    $f(\boldsymbol{x})$: Function to minimize
    $\eta$: Learning rate
    $\epsilon$: Threshold
    $\boldsymbol{x}_0$: Initial parameters
Output: $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$: Parameters that minimize $f(\boldsymbol{x})$

1: $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$                          ▷ Initialize parameters
2: while $\|\nabla f(\boldsymbol{x}_n)\| > \epsilon$ do                ▷ Until convergence
3:     $\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x})$            ▷ Update parameters
4: return $\boldsymbol{x}$                                ▷ Return parameters
```

will not change the parameters significantly anymore. We therefore define a threshold ϵ and stop the algorithm if the norm of the gradient falls below this threshold. The gradient descent algorithm is summarized in algorithm 1.

Going back to neural networks, the function that we want to minimize is the loss function $\lambda(\boldsymbol{y}, \hat{\boldsymbol{y}})$. As the loss function depends on the network's output $\boldsymbol{y}$, which in turn depends on the network's parameters $\boldsymbol{w}$ and $\boldsymbol{b}$, the loss function is a function of the network's parameters $\lambda(\boldsymbol{w}, \boldsymbol{b})$.

For single layer networks, the calculation of the gradients is straightforward. For multi-layer networks however, the loss function depends on the parameters of earlier layers in a non-trivial way. The next section will outline an algorithm, that allows the efficient calculation of the gradients of multi-layer networks.

2.2.6 The Backpropagation Algorithm

The calculation of the gradients is the most computationally expensive part of training a neural network. The invention of the *backpropagation* algorithm by Rumelhart et al. in 1986 [17] was a major breakthrough in the field of neural networks, as it allowed very efficient gradient calculation and thus enabled the training of large neural networks.
The idea behind backpropagation is to propagate the error back through the network after each forward-propagation step using *local gradients* δ. During the forward-propagation step, the activations of each layer are stored in memory and used to calculate the derivatives need for the backpropagation step. This is a form of *automatic differentiation* [50, 51] and allows the calculation of the gradients with the same time complexity as the forward-propagation step. This is sometimes called the *cheap gradient principle* [52]. The following mathematical derivation of the backpropagation algorithm is based on [47, chapter 4.4] as well as [46, chapter 6.5].

To calculate the update of a weight $w_{ij}^{[K]}$, we have to calculate the derivative of the loss function λ with respect to the weight $w_{ij}^{[K]}$:

$$\Delta w_{ij}^{[K]} = \eta \cdot \frac{\partial \lambda}{\partial w_{ij}^{[K]}}. \tag{2.16}$$

This derivative can be evaluated using the chain rule:

$$\frac{\partial \lambda}{\partial w_{ij}^{[K]}} = \frac{\partial \lambda}{\partial a_i^{[K]}} \frac{\partial a_i^{[K]}}{\partial z_i^{[K]}} \frac{\partial z_i^{[K]}}{\partial w_{ij}^{[K]}} = \delta_i^{[K]} a_j^{[K-1]} \tag{2.17}$$

where we used equation 2.8 to simplify the last derivative and defined the local gradient $\delta_j^{[K]}$ as:

$$\delta_i^{[K]} = \frac{\partial \lambda}{\partial a_i^{[K]}} \frac{\partial a_i^{[K]}}{\partial z_i^{[K]}}. \tag{2.18}$$

Let's take a closer look at the local gradient $\delta_i^{[K]}$. The second factor in equation 2.18 is the derivative of the activation function ϕ with respect to the pre-activation value $z_i^{[K]}$. This term is straightforward to calculate. Remembering that the activation function is applied element-wise, using equation 2.9 we get:

$$\frac{\partial a_i^{[K]}}{\partial z_i^{[K]}} = \frac{\partial \phi(z_i^{[K]})}{\partial z_i^{[K]}} = \phi'(z_i^{[K]}). \tag{2.19}$$

The first factor in equation 2.18 is the derivative of the loss function λ with respect to the activation $a_i^{[K]}$. If layer K is the output layer, this derivative is simply the derivative of the loss function with respect to the output value y_i:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \frac{\partial \lambda}{\partial y_i}. \tag{2.20}$$

If layer K is not the output layer, the derivative is slightly more complicated. We can use the chain rule trick again, re-introducing the activation values $a_n^{[K+1]}$ of the next layer, which all depend on $a_i^{[K]}$:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \frac{\partial \lambda}{\partial a_n^{[K+1]}} \frac{\partial a_n^{[K+1]}}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \frac{\partial \lambda}{\partial a_n^{[K+1]}} \frac{\partial a_n^{[K+1]}}{\partial z_n^{[K+1]}} \frac{\partial z_n^{[K+1]}}{\partial a_i^{[K]}}. \tag{2.21}$$

Similar to equation 2.17, we can use equation 2.8 to simplify the last derivative and identify the local gradients $\delta_n^{[K+1]}$:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \delta_n^{[K+1]} w_{ni}^{[K+1]}. \tag{2.22}$$

Plugging equations 2.19 and 2.20 or 2.22 back into equation 2.18 yields the final form of the local gradient:

$$\delta_i^{[K]} = \phi'(z_i^{[K]}) \cdot \begin{cases} \dfrac{\partial \lambda}{\partial y_i} & \text{if layer } K \text{ is the output layer} \\ \displaystyle\sum_{n=1}^{m_{K+1}} \delta_n^{[K+1]} w_{ni}^{[K+1]} & \text{otherwise.} \end{cases} \tag{2.23}$$

This equation can be applied recursively to calculate the local gradients of all layers from the output layer to the input layer. The weight update $\Delta w_{ij}^{[K]}$ can then be calculated using equation 2.16 and 2.23:

$$\Delta w_{ij}^{[K]} = \eta \cdot \delta_i^{[K]} a_j^{[K-1]}. \tag{2.24}$$

When introducing the backpropagation algorithm, I summarized it as a method of *propagating the error back through the network*. The recursive usage of equation 2.23 is exactly that: We start at the output layer and calculate the local gradients of all nodes in the output layer using equation 2.23. Then we use these local gradients to calculate the local gradients of the previous layer using equation 2.23 again and so on until we reach the input layer. Note that we need to store the activations of each layer in memory during the *forward-pass* to calculate the updates during the *backward-pass* as mentioned in the beginning of this section.

The derivations for the bias updates are analogous to the weight updates and are therefore omitted here. The only difference is that the last term in equation 2.17 is replaced by 1 as the bias is not connected to any previous layer. The algorithm for a whole training step (i.e. one forward-pass and one backward-pass) is summarized in algorithm 2.

2.2.7 Gradient Descent Optimization Algorithms

The gradient descent algorithm outlined in section 2.2.5 can be implemented in different ways, depending on how the training examples are used.
Batch gradient descent uses the whole training set to calculate the gradient in each iteration [53], making it very accurate, but also computationally expensive. *Stochastic gradient descent* (SGD) on the other hand uses only a single training example to calculate the gradient in each iteration [53]. This allows for online learning, where the network is trained efficiently on the

Algorithm 2 One training step of a neural network: Forward- and Backpropagation

```
Require:
    $\boldsymbol{x}$: Input vector
    $\hat{\boldsymbol{y}}$: Target output vector
    $\boldsymbol{w}$: Weight array containing all weight matrices $\boldsymbol{w}^{[K]}$
    $\boldsymbol{b}$: Bias array containing all bias vectors $\boldsymbol{b}^{[K]}$
    $\lambda$: Loss function
    $\phi$: Activation function
    $\eta$: Learning rate
Output: Updates the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ of the network in place.

 1: $\boldsymbol{a}^{[0]} \leftarrow \boldsymbol{x}$                                                  ▷ Initialize activations
 2: for $K = 1, \dots, L$ do                                                      ▷ Forward-pass
 3:     $\boldsymbol{z}^{[K]} \leftarrow \boldsymbol{w}^{[K]}\boldsymbol{a}^{[K-1]} + \boldsymbol{b}^{[K]}$              ▷ Calculate pre-activations
 4:     $\boldsymbol{a}^{[K]} \leftarrow \phi(\boldsymbol{z}^{[K]})$                                   ▷ Calculate activations
 5: $\delta^{[L]} \leftarrow \phi'(\boldsymbol{z}^{[L]}) \cdot \frac{\partial \lambda(\boldsymbol{a}^{[L]}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{a}^{[L]}}$          ▷ Initialize local gradients
 6: for $K = L, \dots, 1$ do                                                      ▷ Backward-pass
 7:     $\Delta\boldsymbol{w}^{[K]} \leftarrow \eta \cdot \delta^{[K]}\boldsymbol{a}^{[K-1]}$                    ▷ Calculate weight updates
 8:     $\Delta\boldsymbol{b}^{[K]} \leftarrow \eta \cdot \delta^{[K]}$                                  ▷ Calculate bias updates
 9:     $\delta^{[K-1]} \leftarrow \phi'(\boldsymbol{z}^{[K-1]}) \cdot \boldsymbol{w}^{[K]T}\delta^{[K]}$       ▷ Calculate local gradients
10: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \Delta\boldsymbol{w}$                                             ▷ Update weights
11: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \Delta\boldsymbol{b}$                                              ▷ Update biases
```

fly while it is being used, but it also introduces a lot of noise into the gradient calculation, overshooting the minima.
Mini-batch gradient descent is a compromise between the two, using a small batch of training examples to calculate the gradient in each iteration [53]. This combines the efficiency of SGD with the accuracy of batch gradient descent, but still has some downsides like getting stuck in saddle points, where the gradient is close to zero, but the function is not at a minimum [53]. Furthermore, the learning rate has to be chosen carefully and the constant learning rate can be problematic for datasets where some features occur more frequently than others [53].

To address these problems, many optimization algorithms have been developed over the years, two of which are important for this thesis:
The *Adam* algorithm [54] keeps track of the first and second moments of the gradient (an exponentially decaying average of the past gradients and their squares) and uses them to adapt the learning rate for each parameter individually [54, 53]. This can be seen in analogy to the physical momentum of a ball rolling along the hills and valleys of the loss function landscape. Using momentum dampens the sharp oscillations that can arise when using the pure gradients ("only the gravitational forces").
The *AdamW* algorithm [55] is a variation of Adam that introduces slight modifications in order to improve its generalization performance. To prevent overfitting, Adam uses an L2-regularization term that adds a small fraction of the old parameter values to the gradient before updating the parameters. AdamW decouples this weight decay from the gradient calculation and adds it directly to the parameter update, which, according to the authors, improves the generalization performance of the network [55].

2.2.8 Summary

In this chapter, the foundations of neural networks were outlined. Section 2.2.2 started by defining the building blocks of neural networks: The *nodes*. Section 2.2.4 combined *Layers* of these nodes into a *fully connected neural network*. The specific *architecture* of the network, that is the number of layers, the number of nodes per layer and the *activation functions*, depends on the problem that is being solved. Later, we will see that in our case of Deep-Q-Learning, the size of the *input vector* is determined by the number of cells that are visible to the agent and the size of the *output vector* is determined by the number of possible actions that the agent can take.

On this basis, the *forward-propagation* step could be analyzed in detail and the significance of the activation function was explained. Next, section 2.2.5 introduced the *loss function* as a measure of the network's error and the *gradient descent* algorithm was outlined as a method to minimize this error. Finally, the *backpropagation* algorithm treated in section 2.2.6 provides a way to efficiently compute the gradients needed for gradient descent and enables the *training* of large neural networks. The whole process of forward-propagation, followed by backpropagation and network parameter updates is summarized in pseudocode in algorithm 2.

Chapter 3

Deep Q-Learning

3.1 Reinforcement Learning

Out of the three main branches of machine learning, *supervised learning*, *unsupervised learning* and *reinforcement learning*, reinforcement learning is the most similar to the way humans learn.
While supervised learning is based on the idea of learning from examples and unsupervised learning is concerned with finding patterns in data [56], reinforcement learning is based on the idea of learning from experience [57, chapter 1.1]. In reinforcement learning, the learning agent is not told what action to take, but instead has to learn which actions lead to a desired outcome by trial and error.
This framework has led to many successes over the last decades, including computer programs that beat the world's best players in board games like chess, Go, backgammon [22, 23, 58] or even in video games like Atari games [59] and Dota 2 [60], as well as teaching robots how to walk and handle objects [61, 62]. TODO: Add OpenAI Q^* here?

3.1.1 Overview

In a reinforcement learning scenario, the *agent* observes some information about the *environment*, called the *state* of the environment. It then uses its *policy* to map the state to an *action* that it takes in the environment. As a reaction, the environment returns a *reward* to the agent and transitions to a new state [63].
During the learning process, the agent tries to find a policy that maximizes the total reward that it receives from the environment. Depending on the type of problem, the agent can either try to maximize the reward in the short term or in the long term.

The following sections will refine these concepts and introduce the formalism of reinforcement learning.

3.1.2 Notation

Before diving into the details of reinforcement learning, I will introduce the notation that will be used throughout the following sections. It is mostly identical to the notation used in [57]. Some of these definitions might seem a bit abstract at first, but they will become clearer in the following sections, where they are introduced and explained one by one.

- s_t: The state of the environment at time step t.
- a_t: The action taken by the agent at time step t, based on the state s_t.

- r_t: The reward returned by the environment at time step t, based on the state s_t, action a_t and the new state s_{t+1}.
- π: The policy of the agent, mapping states to actions.
- $\pi(a|s)$: The probability of the agent taking action a in state s.
- $p(s_{t+1}|s_t, a_t)$: The probability of the environment transitioning to state s_{t+1} after the agent takes action a_t in state s_t.
- τ: A trajectory, i.e. a sequence of states, actions and rewards $(s_0, a_0, r_0, s_1, a_1, r_1, \dots)$.
- G_t: The return at time step t, i.e. the total discounted reward from time step t onwards.
- γ: The discount factor, determining how much future rewards are worth compared to immediate rewards.
- $v_\pi(s)$: The value of state s under policy π, i.e. the expected return when starting in state s and following policy π.
- $q_\pi(s, a)$: The value of taking action a in state s under policy π, i.e. the expected return when starting in state s, taking action a and then following policy π.
- π_*, v_*, q_*: The optimal policy, value function and action-value function respectively.

3.1.3 Important Concepts

Markov Decision Processes

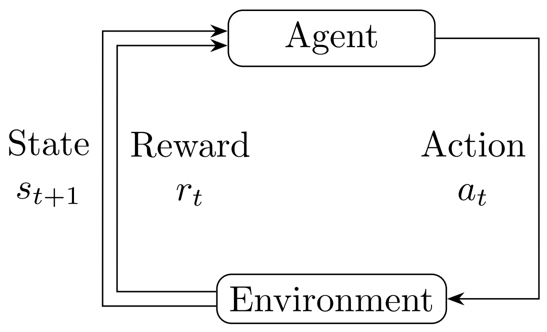


Figure 3.1: The agent-environment interface in a Markov decision process as defined in [57, chapter 3.1]. The agent observes the state s_t of the environment and takes an action a_t. The environment transitions to a new state s_{t+1} and returns a reward r_t to the agent.

Markov decision processes (MDPs) are a mathematical framework for modeling decision-making in situations where outcomes are partly random and only partly under the control of the agent [57, chapter 3]. In an MDP, the agent's actions influence not only the immediate reward, but also the next state of the environment and therefore all future rewards. They are a powerful abstraction that can be used to model a wide range of problems, from simple board games to complex real-world scenarios.
MDPs can be seen as a generalization of discrete-time *Markov chains* (introduced in section 1.3), where the next state of the system is determined only by the current state and not by any previous states. This property is called the *Markov property* and can be expressed in the notation of this chapter as [64]:

$$p(s_{t+1}|s_1, \dots, s_t) = p(s_{t+1}|s_t) = p_{ss'}. \tag{3.1}$$

It is intuitive that the state trajectories in a reinforcement learning scenario should always satisfy this property, as it allows the agent to make decisions based on the current state without having to consider the whole history of states that led to the current state.

MDPs slightly modify the definition of Markov chains by introducing the notion of actions and rewards. As outlined in the previous section, the agent interacts with the environment by taking actions a_i based on the current state of the environment s_i and receives rewards r_i from the environment in return. This *agent-environment interface* is illustrated in figure 3.1. A *trajectory* τ in an MDP is a sequence of states, actions and rewards [57, p. 48]:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \dots). \tag{3.2}$$

The transition probability now depends on the action a_t that the agent takes in state s_t:

$$\begin{aligned} p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} &\to [0,1] \\ s_{t+1}, r_t, s_t, a_t &\mapsto p(s_{t+1}, r_t | s_t, a_t), \end{aligned} \tag{3.3}$$

where $\mathcal{S}$ denotes the set of all possible states and $\mathcal{A}$ denotes the set of all possible actions. This probability function completely determines the dynamics of the reinforcement learning environment. Environments can be entirely deterministic or entirely stochastic, or anything in between. The four-argument transition probability, if known, can of course be used to calculate other properties of the environment, such as the three-argument state-transition probability [57, p. 49]:

$$\begin{aligned} p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} &\to [0,1] \\ s_{t+1}, s_t, a_t &\mapsto p(s_{t+1} | s_t, a_t) = \sum_{r_t \in \mathcal{R}} p(s_{t+1}, r_t | s_t, a_t). \end{aligned} \tag{3.4}$$

where $\mathcal{R}$ denotes the set of all possible rewards.
Furthermore, knowing the four-argument transition probability of an environment allows us to create a simulation of the environment, which can be used to test reinforcement learning algorithms. This is a very useful property, as it allows us to test reinforcement learning algorithms in a controlled environment before deploying them in the real world.

Reward

After each time step t, the agent receives a reward r_t from the environment. The reward is a scalar value that indicates how good or bad the action a_t that the agent took in state s_t was [57, p. 53]. Rewards are the "trainer's" way of telling the agent what it should achieve. In order to use the full potential of reinforcement learning, rewards should be chosen carefully. The reward signal should not tell the agent *how* to achieve the goal, but only *what* the goal is. The agent should then be able to figure out the best way to achieve the goal by itself [57, ch. 3.2]. For example, if the goal is to teach a robot to walk, the agent should receive a reward for moving forward while maintaining a certain balance, but not for moving its legs in a certain way.
In order to be able to learn from these rewards, not just the immediate reward r_t should be taken into account, but also the rewards that the agent will receive in the future. This is done by introducing the notion of *return* The return G_t is a measure of the cumulative reward that the agent will receive from time step t onwards [57, ch. 3.3]. The simplest form of return is just the sum of all future rewards:

$$G_t = r_t + r_{t+1} + r_{t+2} + \dots + r_T = \sum_{i=t}^{T} r_i, \tag{3.5}$$

where T is the final time step in the current *episode*. Episodic tasks are task where a *terminal state* s_T can be reached, after which the episode ends. Examples are board games like chess or Go, where the game ends after one player wins. In contrast, for *continuing tasks*, as is the case for the environment that we will consider in this thesis, there is no terminal state and $T = \infty$. In this case, it's useful to introduce a *discounted* return [57, ch. 3.3]:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=t}^{\infty} \gamma^{i-t} r_i. \tag{3.6}$$

The *discount factor* γ determines how much immediate rewards should be valued compared to future rewards. A discount factor of $\gamma = 0$ means that only the immediate reward is taken into account, while a discount factor of $\gamma = 1$ would value all future rewards equally.
The discounted reward formula can be rewritten recursively as [57, p. 55]:

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots) \\ &= r_t + \gamma G_{t+1}. \end{aligned} \tag{3.7}$$

This recursive formulation is very important, as it allows us to express the return in terms of the return at the next time step. This will be useful in the next section, where we want to calculate the expected return of a state-action pair.

Policies

Reinforcement learning agents are characterized by their *policy* π. The policy maps states s to actions $a = \pi(s)$ and therefore determines the behavior of the agent [57, ch. 3.5]. The policy can be either *deterministic* or *stochastic*. A deterministic policy maps each state to exactly one action, while a stochastic policy maps each state to a probability distribution over actions. If a policy deterministically maps states to what it believes to be the best action, it is called a *greedy* policy [57, p. 64]. If an optimal policy has been found, it can be greedy, as it will always choose the best action. However, during the learning process, stochastic policies should be preferred, as they allow the agent to explore the environment and find better policies. In order to learn, an "inexperienced" agent needs to try actions that it assumes to be suboptimal to find out whether they are really suboptimal. This is called the *exploration-exploitation dilemma* [57, p. 3].
A popular solution which will also be used in this thesis is the *ϵ-greedy* policy [57, p. 100]. ϵ-greedy policies are greedy with probability $1 - \epsilon$ and choose a random action with probability ϵ. During training, ϵ is slowly decreased over time, so that the agent will explore the environment more at the beginning and exploit its knowledge later on. During evaluation, ϵ is set to zero, so that the agent will always choose the best action.

Value Functions

To actually build a policy that maximizes the expected return, the agent needs to know how good each state or state-action pair is. In order to do this, most reinforcement learning algorithms use *(state) value functions* [57, ch. 3.5]. Value functions estimate the expected return of a state s when following policy π [57, p. 58]:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i \middle| s_t = s\right]. \tag{3.8}$$

In the same manner, we can define the *action-value function* [57, p. 58]:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i \middle| s_t = s, a_t = a\right]. \tag{3.9}$$

Optimality and Bellman Equations

Now the recursion formula from equation 3.7 comes in handy to express the value functions in terms of the value functions of the next state:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma G_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_{r_t} p(s', r_t|s,a) \left[r_t + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_{r_t} p(s', r_t|s,a) \left[r_t + \gamma v_\pi(s')\right], \end{aligned} \tag{3.10}$$

where we reintroduced the four-argument transition probability. The summations are over all possible actions a and all possible next states s' and rewards r_t. Equation 3.10 is called the *Bellman equation* for v_π [57, p. 59].
The Bellman equation for q_π is analogous [63]:

$$q_\pi(s,a) = \sum_{s'} \sum_{r_t} p(s', r_t|s_t, a_t) \left[r_t + \gamma \sum_{a_{t+1}} \pi(a_{t+1}|s_{t+1} = s') q_\pi(s', a_{t+1})\right]. \tag{3.11}$$

The Bellman equations heavily simplify the calculation of the value functions, as they allow us to express the value of a state or state-action pair in terms of the values of the next state or state-action pairs. Imagine a game of chess: Even for very fast computers, evaluating every possible trajectory of moves until one player wins is not feasible. However, the Bellman equations allow us to calculate the value of a state or state-action pair without having to consider all possible trajectories. We just have to keep track of the values of individual states or state-action pairs and update them according to the Bellman equations.

We can now use these concepts to define *optimal* policies π_*. A policy is considered "better" than another policy, if it achieves a higher expected return in every state [57, p. 62]:

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}. \tag{3.12}$$

An optimal policy π_* is a policy that is better than or equal to all other policies [57, p. 62]. Associating the value functions v_* and q_* with the optimal policy π_*, we can define the optimal (action-)value functions as [57, ch. 3.6]:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}. \tag{3.13}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{3.14}$$

The optimal value functions satisfy the *Bellman optimality equations* [57, ch. 3.6]. When an agent follows an optimal policy, the sum over all possible actions can be replaced by using the action that maximizes the expected return:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_\pi\left[r_t + \gamma \max_{a'} q_*(s_{t+1}, a') \middle| s_t = s, a_t = a\right] \\ &= \sum_{s'} \sum_{r_t} p(s', r_t|s,a) \left[r_t + \gamma \max_{a'} q_*(s', a')\right]. \end{aligned} \tag{3.15}$$

Solving equation 3.15 yields the optimal action-value function q_*, which can then be used to derive the optimal policy π_*. For most reinforcement learning problems however, it is not feasible to solve the Bellman optimality equations analytically, even if the transition probabilities are known. Therefore, most reinforcement learning algorithms use iterative methods to approximate the optimal value functions [57, ch. 4].

3.1.4 Summary

Before we dive into the details of the Deep Q-Learning algorithm, let's summarize the most important concepts of reinforcement learning that were introduced in this section.

Reinforcement learning is a framework for *learning from experience*. An *agent* interacts with an *environment* by taking *actions* based on the current *state* of the environment. The environment returns a *reward* which the agent uses to keep track of the *return* that it will receive in the future. This in turn allows the agent to learn a *policy* that maximizes the expected the return by learning the *value* of each state or state-action pair. *Optimal* policies can be found by solving the *Bellman optimality equations*.

3.1.5 Algorithms

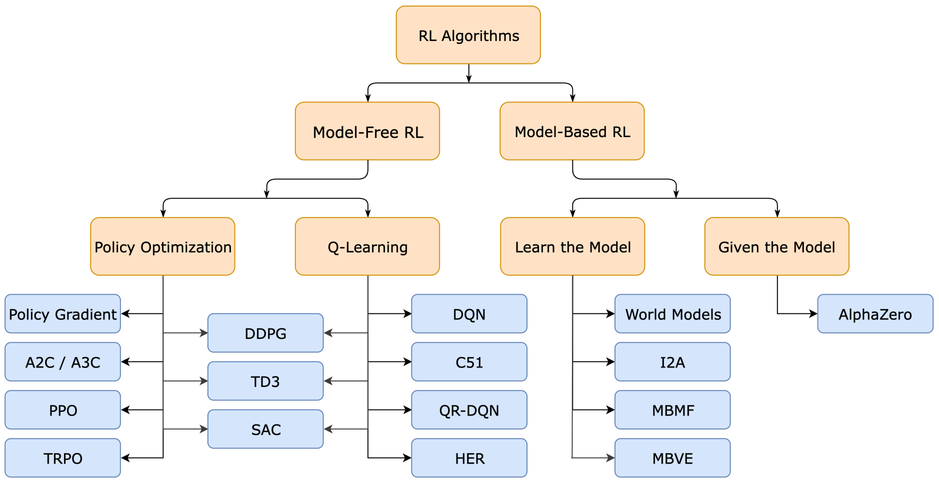


Figure 3.2: Overview of the different classes of reinforcement learning algorithms [65]. The papers that introduced the algorithms are also available at [65].

Figure 3.2 provides a basic overview of the different classes of reinforcement learning algorithms. We will not explain the details of each algorithm here, but instead focus on the general distinctions between the different classes of algorithms.

The first distinction is between *model-based* and *model-free* algorithms. Model-based algorithms have an internal model of the environment, which is either learned or provided by the user [65]. This allows them to plan ahead and simulate the environments dynamics to find the best action to take. Model-free algorithms on the other hand only implicitly learn the dynamics of the environment by interacting with it and learning value functions, as explained in the previous sections. Model-free algorithms are more flexible, as they can be applied to any environment, but they are also less sample-efficient, as they have to learn the dynamics of the environment by trial and error. Model-based algorithms on the other hand can be more sample-efficient, but they are harder to implement and fine-tune to specific problems [65]. Probably the most famous model-based reinforcement learning algorithm is AlphaZero [23], a program that taught itself to play chess, Go and Shogi at superhuman levels.

Model-free algorithms can be distinguished further by *what* is learned. In the last section, we introduced the concept of value functions. Algorithms that learn value functions are called *value-based* algorithms. If they learn the action-value function q_π, they are called *Q-learning* algorithms. Most of these algorithms use *off-policy* learning, which means that they learn the value of the optimal policy π_* while following a different policy π [65]. This allows them to learn from data that was collected at a previous stage of the training process, making them very sample-efficient. Probably the most famous Q-learning algorithm is Deep Q-Learning, which we will discuss in detail in the next section.
The other class of model-free algorithms are *policy-based* algorithms. These algorithms directly learn the optimal policy π_*, instead of learning value functions. Most of these algorithms use *on-policy* learning, restricting them to only learn from data that was collected while following the current policy π [65]. Because of that, they are less sample-efficient than Q-learning algorithms, but they also tend be more stable, as they directly learn the policy, instead of indirectly learning the policy by learning value functions [65].

As we can see in figure 3.2, there are also algorithms that use ideas from different classes of algorithms. In general, it is hard to draw clear distinctions between the different classes of algorithms, as their modular nature allows them to be combined in many different ways [65].

Choosing the best algorithm for a specific problem is one of the most important steps in applying reinforcement learning to a real-world problem. In this thesis, we will use Deep-Q-Learning, as its use of deep neural networks allows it to scale and generalize well. It is also a model free, off-policy algorithm, which makes it relatively easy to implement and modify. The sampling efficiency of off-policy learning is also a big advantage, as it allows training the network quickly on normal hardware.

3.2 Q-Learning

Before diving into the details of the Deep Q-Learning algorithm, we will introduce the general concepts of Q-Learning, building on the concepts of reinforcement learning that were introduced in the previous section.

Q-Learning was introduced by Watkins in 1989 [66]. It is a model-free, off-policy algorithm that learns an action-value function $Q(s, a)$ which is guaranteed to converge to the optimal action-value function q_* [67] [57, ch. 6.5]
The algorithm works by storing a table of action-values $Q(s, a)$ for each state-action pair (s, a) that the agent has encountered. Such a *Q-table* can be seen in figure 3.3. In the beginning, the Q-table is initialized randomly. Then, the agent interacts with the environment by taking actions and receiving rewards. The policy used for this interaction is derived from the Q-table for example by choosing the action with the highest action-value in an ϵ-greedy manner. For each sample (state s, action a, reward r, next state s') that the agent encounters, the Q-table is updated according to the following formula [57, ch. 6.5]:

$$Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{3.16}$$

where η is the learning rate and γ is the discount factor. Comparing with equation 3.15, we can see that this update rule is an iterative approximation of the Bellman optimality equation for q_*. The new approximation of the optimal action-value is a weighted sum of the old approximation, and the new information that was gained from the sample, i.e. the immediate reward r and the discounted value of the next action. The learning rate η determines how much the new information is weighted compared to the old approximation. The discount factor again γ determines how much future rewards are valued compared to immediate rewards.

We see that while time complexity stays constant when using lookup tables, the space

	a_1	a_2	$\dots$	a_n
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$\dots$	$Q(s_1, a_n)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$\dots$	$Q(s_2, a_n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_m	$Q(s_m, a_1)$	$Q(s_m, a_2)$	$\dots$	$Q(s_m, a_n)$

Figure 3.3: A Q-table for an environment with m states and n actions.

complexity grows linearly with the number of states and actions. This makes Q-learning impractical for environments with large state spaces. Also, environments that have continuous state or action spaces have to be discretized, which can lead to a loss of information and therefore suboptimal policies. Even for seemingly simple environments such as a grid, where each grid cell can either be occupied or unoccupied, the number of possible states grows exponentially with the size of the grid. Therefore, we need a more efficient way to represent the action-values. This is where Deep Q-Learning comes in.

3.3 Deep Q-Learning

To solve the problem of large state spaces, we look back at chapter 2, where we learned, that deep neural networks are universal function approximators. This makes them a good candidate for approximating the action-values $Q(s, a)$. When we replace the Q-table with a neural network, we supply the current state s as input and get the action-values $Q(s, a)$ as output. For our grid environment example from the last section, this would mean that we supply the current grid as input. This eliminates the exponential space complexity of the Q-table, as the input vector of the network now grows linearly with the number of cells. Furthermore, the discretization problem is also solved, as the network can now take continuous inputs. Ideally, the network would store the same information as the table, but in a more compact way by learning the patterns in the data.

This approach is called *Deep Q-Learning* (DQN) and was introduced by Mnih et al. in 2013 [59]. DQN also introduces a few other concepts apart from the Q-network, which we will discuss in the following sections.

3.3.1 Q-Network

The Q-network, sometimes also called *policy network*, was already discussed in the previous section. It is a neural network that replaces the Q-table and approximates the action-values $Q(s, a)$. During evaluation, the environments state is supplied as input to the network. After a forward pass through the network, as explained in section 2.2.4, the output vector of the network corresponds to the approximate action-values $Q(s, a)$ for each action a. When the network has converged to a good approximation of the optimal action-values q_*, the action with the highest action-value can be chosen as the action that the agent takes in the current state. During training, this action is chosen in an ϵ-greedy manner, as explained in section 3.1.3 to maintain exploration [68].

3.3.2 Target Network

The Q-network also changes the way that the parameters are updated. We can no longer update individual Q-values, as this information is now encoded in the weights and biases of the network. Instead, we have to update the parameters of the network. This is done by using *gradient descent* and the *backpropagation* algorithm, as explained in sections 2.2.5 and 2.2.6.

The loss function that is used for the gradient descent requires a target output vector $\hat{y}$ for each sample. In the case of Q-learning, this would be the optimal action-values $q_*(a)$. As this is what we're optimizing for, we can't use it as the target output vector. Instead, we could use an earlier approximation of the action-values $Q(s, a)$. However, this can lead to oscillations or divergence of the network parameters, as the target network would use the same parameters as the Q-network [68]. To solve this problem, DQN introduces a second network, called the *target network*. The target network is a clone of the Q-network with frozen parameters. Using this separate network to calculate the action-value targets makes the learning process more stable [68].
In the original algorithm, the target network parameters are updated periodically by copying the parameters from the Q-network. This is called *hard target network update* [68]. In 2016, Lillicrap et al. [69] introduced *soft target network updates*, where the target network parameters are updated by slowly blending the parameters of the Q-network into the target network parameters:

$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{Q}} + (1-\tau)\theta_{\text{target}}, \tag{3.17}$$

where $\tau \ll 1$ is a small number between 0 and 1, θ_{target} are the target network parameters and θ_{Q} are the Q-network parameters.
This makes the learning process even more stable, as the target network parameters are constrained to slow updates [69]. In this thesis, we will also use soft target network updates.

3.3.3 Experience Replay

In the beginning of this section, DQN was introduced as an off-policy algorithm. Mnih et al. introduced a technique called *experience replay* [68], which massively improves the algorithms' quality. The technique is inspired by the way that humans learn [68] and works by storing tuples of states, actions, rewards and next states (s_t, a_t, r_t, s_{t+1}), called *transitions* in a *replay buffer*. During training, the network is updated by sampling a batch of transitions from the replay buffer and performing a gradient descent step on the loss function. This not only allows the network to learn from the same experience multiple times, increasing sample efficiency, but also breaks up the correlation between consecutive samples, which would otherwise decrease learning efficiency and stability [68]. In order to prevent the replay buffer from growing infinitely, the oldest samples are discarded when the buffer is full. This total buffer size and the sampling batch size are hyperparameters that have to be tuned for each problem.

One problem that exists in both on-policy learning and off-policy learning with experience replay is that rare transitions are sampled very infrequently, which can slow down learning. To solve this problem, Schaul et al. introduced *prioritized experience replay* in 2015 [70]. In prioritized experience replay, each transition is assigned a priority based on the magnitude of the loss function [70]. That way, transitions that are "surprising" to the agent are sampled more frequently, which speeds up learning [70].

3.3.4 Summary

In this section, we learned that Deep Q-Learning uses a neural network to map states to action-values. This allows it to scale to large and continuous state spaces. Learning is performed *offline*, by saving encountered transitions to a replay buffer and sampling batches of transitions from the buffer to update the network parameters. A separate target network is used to calculate the action-value targets, which makes the learning process more stable. The target network parameters are updated by slowly blending the parameters of the Q-network into the target network parameters. The complete algorithm is shown in algorithm 3.

Algorithm 3 Deep Q-Learning

```
Initialize replay buffer $\mathcal{D}$ to capacity $N$
Initialize Q-network $Q$ with parameters $\theta$
Initialize target network $\hat{Q}$ with parameters $\theta^{-} = \theta$
Initialize environment to get initial state $s_1$
Initialize $\epsilon$ to $\epsilon_0$
for steps $t = 1, \dots, T$ do
    ▷ Generate new transition
    Perform forward pass through $Q$ to get action-values $Q(s_t, a')$
    With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = \operatorname{argmax}_a Q(s_t, a)$
    Execute action $a_t$ in environment and observe reward $r_t$ and next state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$
    ▷ Update Q-network parameters
    Sample random minibatch of transitions $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s'})$ from $\mathcal{D}$
    Set $\boldsymbol{y} = \boldsymbol{r} + \gamma \max_{a'} \hat{Q}(\boldsymbol{s'}, a')$
    Calculate mean loss $\mathcal{L} = \text{mean}(\mathcal{L}(\boldsymbol{y}, Q(\boldsymbol{s}, \boldsymbol{a})))$
    Perform gradient descent step on $\mathcal{L}$ using backpropagation and update $\theta$
    ▷ Update target network parameters
    Update target network parameters $\theta^{-} \leftarrow \tau\theta + (1 - \tau)\theta^{-}$
    ▷ Update exploration rate
    Decrease $\epsilon$
```

Chapter 4

Physical Model

In this chapter, we will introduce the physical model that that will be used in this thesis. We will start by introducing the 1D TASEP, which is a well studied model for transport processes. Then, we will introduce the 2D TASEP, which is the model that we will use in this thesis. Finally, we will introduce the concept of smarticles, which are smart particles that use reinforcement learning to maximize transport in the 2D TASEP.

4.1 1D TASEP

The TASEP is one of the most well studied models in non-equilibrium statistical physics. It can be used as a stochastic model for transport processes on a 1D lattice and was first introduced by MacDonald, Gibbs and Pipkin in 1968 [71] as a model for protein synthesis, where particles are ribosomes and sites are codons. It was later introduced in mathematics by Spitzer in 1970 [72]. The name TASEP stands for *totally asymmetric simple exclusion process*. *Totally asymmetric* means that particles can only move in one direction. In the more general ASEP, particles have different rates p and q for jumping to the left or right respectively. *Simple* means that particles can only move to the next site. *Exclusion* means that each site can only be occupied by one particle. *Process* means that the model is a stochastic process, specifically a continous-time Markov process on the finite state space

$$\mathcal{S} = \{0,1\}^L, \tag{4.1}$$

where L is the number of sites. Each site can either be occupied by a particle or empty, so the state space consists of all possible configurations of particles on the lattice. It can be treated as a continous-time process, as each particle moves according to its own internal clock, although in this thesis we will discretize time by picking a random particle in each time step.

The 1D TASEP can be treated with periodic boundary conditions, where the first and last site are connected as shown in figure 4.1, or with different rates for insertion and removal of particles, as shown in figure 4.2. Exact solutions are known for the one-dimensional TASEP and different phases can be found, depending on these rates or the density of particles [73, 74].

4.2 2D TASEP

The TASEP can be generalized to higher dimensions. In this thesis, we will use the 2D TASEP, which can be used as a simplified model for traffic flow with multiple lanes or intracellular transport processes, for example the movement of motor proteins on microtubules. In this case, the process is totally asymmetric in one direction and symmetric in the other direction.

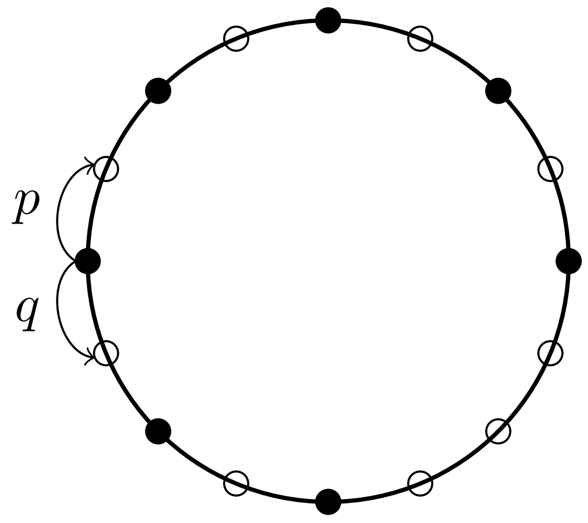


Figure 4.1: A 1D TASEP with periodic boundary conditions. The number of particles is constant. Each particle has a probability p to move clockwise and a probability $1 - p$ to move counterclockwise. If the target site is occupied, the particle stays.

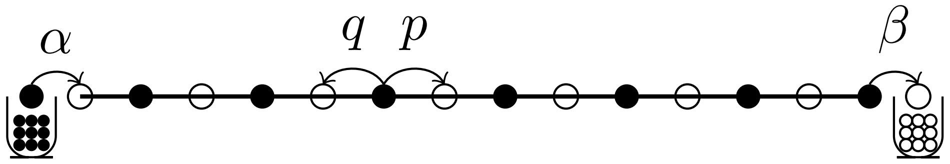


Figure 4.2: A 1D finite TASEP. The reservoir on the left inserts particles with rate α and the reservoir on the right removes particles with rate β.

Particles can only move to the right ("forward"), up or down. We will use periodic boundary conditions in both directions, which means that the number of particles is constant, like on a torus. A small 8x4 version of the 2D TASEP is shown in figure 4.3. The 2D TASEP is not as well studied as the 1D TASEP. No exact solutions are known and only approximations, such as mean-field theory, exist [75].

In this thesis, we will increase the complexity of the 2D TASEP by introducing velocities. The velocity is implemented as a probability of actually performing an attempted jump. For example, if a particle's velocity is 0.5, it will only move in 50% of the cases where a jump is attempted and the target site is empty.

After a basic numerical treatment of the classical 2D TASEP, we will try to optimize the total transport in these systems by introducing *smarticles*.

4.3 Smarticles

Smarticles (**Smart** Part**icles**) are a form of active matter as described in section 1.2. In this thesis we will use smarticles to optimize the transport in the 2D TASEP and observe how global structures can emerge from local interactions. The smarticles will be implemented as particles with a neural network that decides where to move based on the surroundings.

This can be framed as a reinforcement learning problem, where the reward structure is defined by the goal of the system, which is to maximize transport. Local interactions can also be integrated into the reward structure in order to bias the learned behavior towards certain global structures.

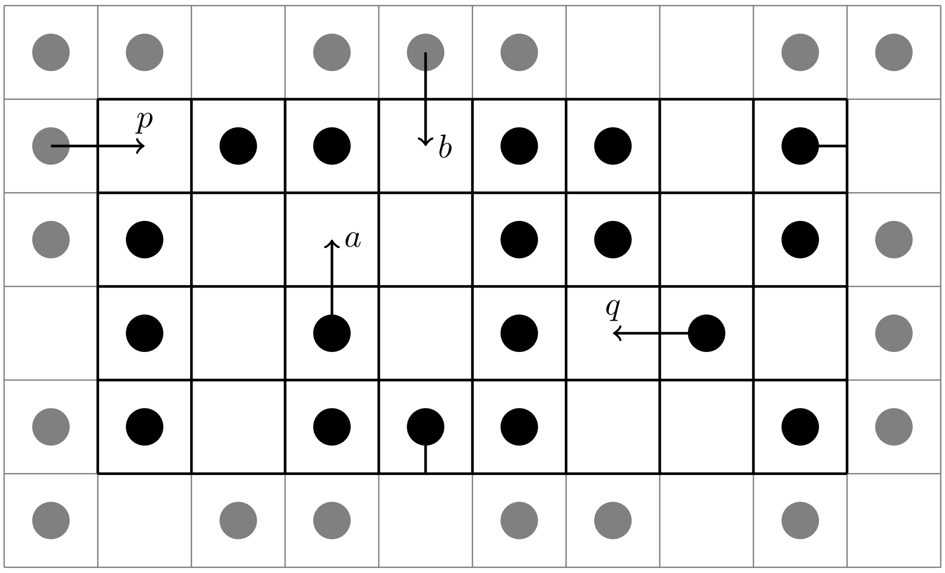


Figure 4.3: A 2D ASEP with periodic boundary conditions. The number of particles is constant. Each particle has a probability p to move forward, a probability q to move backward and probabilities a and b to move up or down respectively. If the target site is occupied, the particle stays. $p + q + a + b = 1$.

Chapter 5

Implementation

Now that the theoretical foundations of the TASEP and Deep Q-Learning have been introduced, we can start to put the pieces together and implement the smarticles. This chapter will start with the implementation of the classical 2D TASEP to set a baseline for the performance of the smarticles. Then, we will implement the smart TASEP and introduce a number of modifications.
All code was written in Python to reduce the amount of boilerplate code and to make the code understandable for a wider audience. Python code is often thought to be slow, but the performance-critical parts of the code were outsourced to low-level approaches such as just-in-time compilation and native low-level code used by third-party libraries. All code is available at [76], with the smart TASEP code being well documented (TODO: and available as pypi package).

5.1 Classical 2D TASEP

The first step is to implement the classical 2D TASEP. This will serve as a baseline for the performance of the smarticles.
The implementation consists of a simple python script with the main loop for the simulation just-in-time-compiled to machine code using the `numba` library [77]. The system is represented as a 2D array of integers, where 0 represents an empty site and 1 represents a site occupied by a particle. The main loop is shown in algorithm 4. In each time step, a random particle is selected and a random direction is chosen. If the target site is empty and the random direction is not backward, the particle moves to the target site and the move counter for the chosen direction is incremented. When different velocities are used, a random number is drawn and the jump is only performed when the random number is smaller than the velocity. In this case, the velocities are floats in the range $(0, 1)$ and stored in the array instead of the ones. Finally, the time step counter is incremented and the loop starts again.

Algorithm 4 Main loop of the 2D TASEP simulation.

```
Initialize system $S$ as empty 2D array
Fill in particles randomly (when using velocities, fill in velocities $v \in (0,1)$ instead of ones)
Initialize time step counter $t = 0$
Initialize move counters $m_{\text{up}} = 0$, $m_{\text{down}} = 0$, $m_{\text{right}} = 0$
while $t < T$ do
    Select random position $\boldsymbol{x} = (x_1, x_2)^T$ in system
    if site $S_{x_1,x_2} \neq 0$ then
        Select random direction $\boldsymbol{d} \in \{\text{up}, \text{down}, \text{right}, \text{left}\}$ with equal probability
        if site $\boldsymbol{x} + \boldsymbol{d} == 0$ and $\boldsymbol{d} \neq$ left and random $r \in (0,1) < S_{x_1,x_2}$ then
            Swap sites $S_{x_1,x_2}$ and $S_{x_1+d_1,x_2+d_2}$ to perform jump
            $m_d \leftarrow m_d + 1$
    $t \leftarrow t + 1$
```

5.2 SmartTASEP Python Package

5.2.1 Overview

The smart TASEP was implemented as a PyPi package available at (TODO). The package is called `SmartTasep` and can be installed using pip. This makes it easy to not only reproduce the results of this thesis, but also to conduct further research using the smart TASEP. The package is well documented and contains a number of examples that show how to use the package. Features include:

- **Training** of DDQN agents in the 2D TASEP
- **Evaluation** of trained agents
  - using **metrics** such as the particle current
  - using a "**playground**" to test the agent's behavior for custom states
  - using the **real-time visualization** of the simulation
- Different **velocities** for particles
- Different **initial conditions**
- Different **reward structures**
- Different **network architectures**
- Different **replay buffer** implementations (uniform, prioritized)
- Different **hyperparameters**, **training schedules and environment parameters**
- Real-time **visualization** of the simulation and metrics
- **Saving** and **loading** of trained agents

The package consists of different modules that are shown in figure 5.1. They will be explained one by one in the following sections. Helper modules and interfaces will not be explained, as they are not relevant for the understanding of the smart TASEP.

5.2.2 GridEnv and EnvParams

The `GridEnv` class implements the 2D TASEP environment for a reinforcement learning agent. It is a subclass of the `gym.Env` class from the OpenAI `gym` library [78]. This allows the environment to be used with any reinforcement learning algorithm that is compatible with

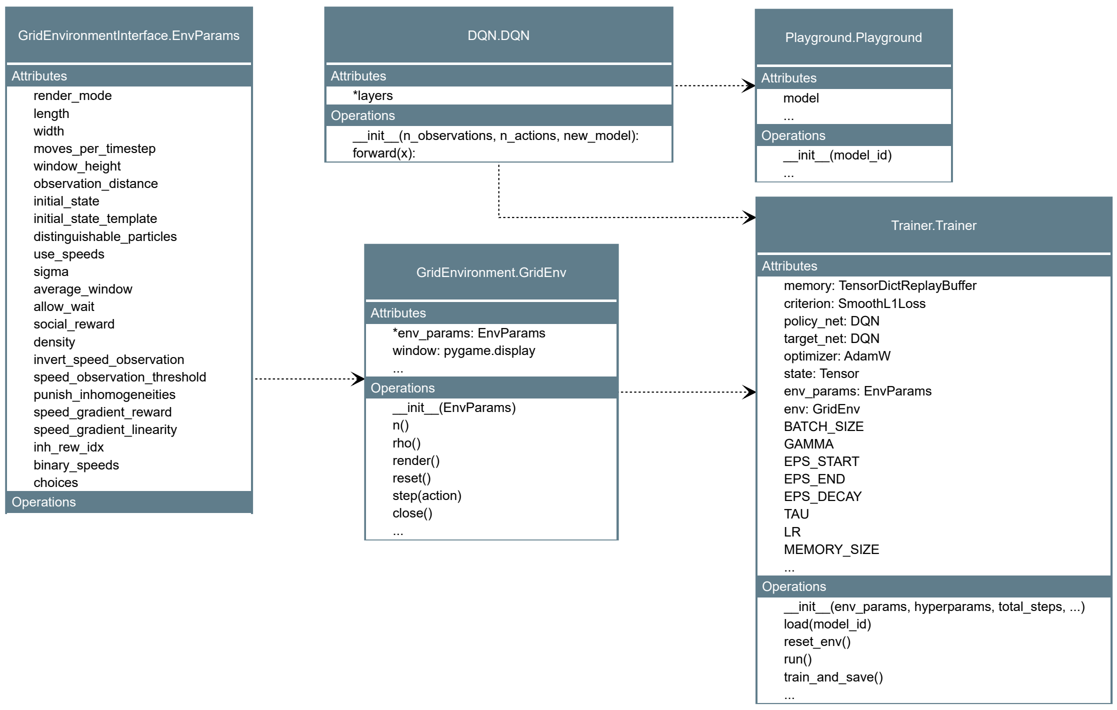


Figure 5.1: UML diagram of the `SmartTasep` python package. Only selected methods and attributes are shown. The full documentation is available at [76].

the OpenAI Gym interface. The environment is initialized with an `EnvParams` object, which may contain a number of parameters to customize the environment size and initial conditions, the reward structure, the visualization and metrics as well as other properties e.g. to control the dynamics of the reinforcement learning algorithm.

Internally, the `GridEnv` class also uses a 2D array to store the state of the system. For indistinguishable particles with equal velocities, this is a binary array, where 0 represents an empty site and 1 represents a site occupied by a particle. Section 5.2.5 will explain that sometimes, particles have to be distinguishable. In that case, the binary array is not suited to store the state. Instead, during initialization, each particle is assigned a unique integer ID, which is stored in the array instead of the ones. If the particles have different velocities, the velocities are stored by adding them to the ID. As the velocities are floats in the range $(0, 1)$, they can be stored in the same array without any loss of information. We can retrieve the ID and velocity v of a particle by using the floor and modulo operators on the stored value s respectively:

$$\text{ID} = \lfloor s \rfloor, \quad v = s \bmod 1. \tag{5.1}$$

This is very memory efficient, as we only need one array to store the state of the system.
In addition to the state array, the `GridEnv` also stores the current particle's position in order to be able to calculate the reward after an agent submits its action.
The following are the two most important methods of the `GridEnv` class:

reset

Resets the environment to its initial state and returns the initial state. Note that in this case, "state" does not refer to the whole system, but only to the selected particle's observation of the system.

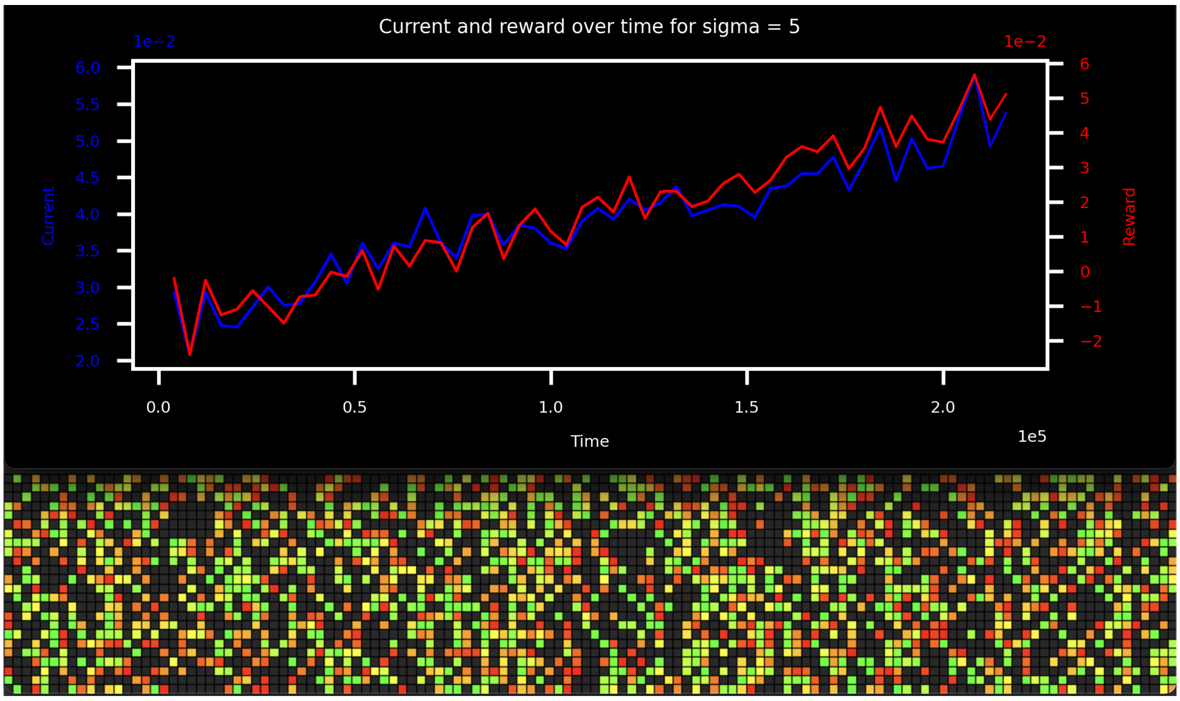


Figure 5.2: Screenshot of the real-time visualization of the smart TASEP, when the `render_mode` parameter is set to "`human`". The visualization shows the system matrix, where particle speeds are represented by colors ranging from red for the slowest particles to green for the fastest particles. Above the system matrix, current and reward are plotted against the number of time steps.

Agents perceive the system as a $d \times d$ grid centered around them, with the observation distance $d/2$ being a parameter. Observations do not include the particle IDs, as this information is irrelevant to the agent and would even make the problem harder, as the agent would have to learn to distinguish between particles.
Particle velocities are included in the observations, as they are relevant to the agent. Optionally, the observed velocities can be converted to shifted inverse velocities:

$$v_{\text{shift, inv}} = 1 - v + v_{\text{shift}}. \tag{5.2}$$

This has two possible advantages. Firstly, slow particles are now represented by high values, because they correspond to "more blockage" in the system. Intuitively, it makes sense to frame the problem in this way, as slow particles are not only worse for the transport, but also more persistent in the observation. Secondly and more importantly, the shift v_{shift} prevents fast particles from being invisible. Without the velocity inversion, particles with very low velocities become indistinguishable from empty sites, which makes it impossible for the agent to learn to avoid them. The inversion alone does not solve this problem, as particles with very high velocities now become invisible.

It is important to note that this problem is less about the absolute values of the observations and more about the distinguishability between observations. At first glance, it seems like this rescaling of velocities is not necessary, as the weights can be negative and the biases can be adjusted to compensate for the shifted values. However, this is not the case, as the rescaling is only applied to the particles, not to the empty sites.

step

Takes an action as an argument and returns the next state and the reward after performing the action. Actions are encoded as integers, which, as we will see, correspond to the index of the output neuron with the highest activation. The form of the next state or states depends on the environment parameters. See section 5.2.5 for more details.

The reward is calculated based on the reward structure that was defined in the `EnvParams` object. Each option can be included or excluded from the reward calculation, which allows for a high degree of customization. Examples include:

- The default structure yields a positive reward for moving forward, a negative reward for trying to move into an occupied site and zero reward for waiting or switching lanes.
- The `social_reward` option adds a negative reward switching lanes into a site with a particle behind it. The magnitude of the reward is proportional to the velocity of the blocked particle. The analogy in the traffic flow interpretation is the car horn that a driver behind you would use if you switched lanes in front of them.
- The `punish_inhomogeneties` option combined with the `inh_rew_idx` parameter can add different complex potential-like rewards to encourage the emergence of global structures. This will be explained in more detail in section TODO

When rendering is enabled, the step method also renders the state of the system in regular intervals. The system matrix is rendered using the `pygame` library, which is fast enough to not bottleneck the simulation. Velocities are encoded as colors using the HSL color space, which makes it easy to convert velocities to hues. The velocity range $(0, 1)$ is mapped to the hue range $(0, 120)$, which corresponds to the red to green range.

5.2.3 DQN

The `DQN` class implements the deep neural network that is used to approximate the action-values $Q(s, a)$. It is a subclass of the `torch.nn.Module` class from the PyTorch library. The network has two hidden layers and uses the `ReLU` activation function introduced in section 2.2.3. The output layer uses the identity activation function, as we want to approximate the action-values directly. During initialization, the size of the flattened observation array and the number of actions are passed as arguments, to determine the size of the input and output layers respectively. An additional parameter switches between two 128-neuron hidden layers and a 24-neuron hidden layer followed by a 12-neuron hidden layer.

As a subclass of the `torch.nn.Module` class, the `DQN` class implements the `forward` method, which performs the forward pass. The `forward` method can be called with a single observation when evaluating the network or with a batch of observations when training the network. Backpropagation is efficiently implemented in PyTorch using automatic differentiation, as explained in section 2.2.6. During a forward pass, all operations (e.g. network layer operations, activation functions, loss function) automatically store the information that is needed for backpropagation in the `torch.Tensor` objects used for the calculations. When the `backward` method is called on the loss tensor at the end of the forward pass, the gradients are automatically calculated and stored in the `grad` attribute of each tensor. These gradients are then used to update the network parameters as will be explained in section 5.2.5.

5.2.4 ReplayBuffer

The replay buffer is also one of the performance-critical parts of the implementation. Instead of a naive custom implementation, the `SmartTasep` package uses the `TensorDictReplayBuffer` or TensorDictPrioritizedReplayBuffer classes together with the LazyTensorStorage class from the torchrl library [79]. The torchrl library is a reinforcement learning library based on PyTorch that implements many very efficient modules for high performance reinforcement learning training using parallel environments, multithreading and supports computation on the GPU. The ReplayBuffer classes implement very efficient sampling, storage and priority update of single transitions or batches.

The torchrl library implements many more features that could be used to improve the performance of the smart TASEP, but due to the limited time of this thesis and the early stage of the library, only the replay buffer was used. The SmartTasep package uses the TensorDictReplayBuffer class for uniform sampling and the TensorDictPrioritized-ReplayBuffer class for prioritized sampling. The LazyTensorStorage class is used internally by these classes to store the transitions.

5.2.5 Trainer

The Trainer class is the heart of the SmartTasep package. It implements the training loop and the evaluation loop as well as some additional functionality such as saving and loading trained agents. The following are the most important methods of the Trainer class:

Constructor

The constructor method initializes the Trainer object. It takes a number of parameters:

- env_params: An EnvParams object that contains the parameters for the environment as explained in section 5.2.2.
- hyperparams: A Hyperparams object that contains the hyperparameters for the reinforcement learning algorithm. The structure of the Hyperparams object is the same as the EnvParams object, but the parameters are different.
- Additional keyword arguments that control the training schedule, visualization and general algorithm choices such as the replay buffer implementation or whether different networks should be used for the individual particles.

After some basic input validation, e.g. checking if any of the parameter choices are incompatible, the constructor method initializes all subcomponents needed for training and evaluation of the agents as well as the visualization in pygame, the plotting in matplotlib and the accumulation of metrics.
A GridEnv object is initialized with the EnvParams object and DQN objects are initialized as Q-networks and policy networks. Depending on the parameter choices, the Q-network and policy network are either shared between all particles or n pairs of separate networks are initialized. In the latter case, the particle velocity range is divided into n equally sized intervals and each pair of networks is responsible for the particles in one of these intervals. This allows different types of particles (e.g. fast, medium and slow for $n = 3$) to learn different policies without the space complexity of a separate network for each particle.
The size of the observation arrays and the action space is determined by the GridEnv object and passed to the DQN objects during initialization. The ReplayBuffer objects are initialized with the corresponding parameters from the Hyperparams object. Furthermore, pytorch's SmoothL1Loss loss function is initialized and pytorch's AdamW optimizers (see section 2.2.7) are initialized for each Q-network.

train

The `train` method implements the training loop. As the TASEP is not an episodic task, the method consists of only one loop running the simulation for a specified number of time steps. Optionally, the training can be performed pseudo-episodically, where the environment is reset in regular intervals, possibly with changing initial conditions, for example to train smarticles in environments with variable density. I refer to this as *pseudo*-episodic, as we are still using a replay buffer with stored transitions from previous episodes.

Depending on the parameters, the `train` method uses slightly different algorithms that differ in the way that transitions are stored in the replay buffer. There are essentially three distinct ways to generate transitions:

1. **Immediate transitions:** The environment returns the observation of the particle that took the action immediately after the action was executed. When this observation is used in transitions, the state-next state pairs contain no information about the environment dynamics, as there is no time between the states for the environment/other particles to react to the action. In this case, the smarticles learn a policy for a static environment, while actually living in a very dynamic environment. Note that in this case, the environment has to also return an additional next state for a new particle in order to move different particles in each time step.
2. **Weakly correlated transitions:** This is the natural way to combine algorithms 3 and 4. The environment returns the next observation after picking a new particle which is then used both as the next state stored in the transition and as the current state for the next action. This method only makes sense when the same networks are shared between all particles, as the next state is not the next state of the particle that took the action. Instead, the network learns a policy for an environment where it is thrown around, each new state seemingly random and only weakly correlated with the previous state. This algorithm is used when the `distinguishable_particles` parameter is set to `False`.
3. **Delayed transitions:** When particles are distinguishable and the current particle's ID is returned by the environment together with the next state, the next state in a transition can be set to the observation of the same particle that took the action at the time step where the particle is picked again and has to choose its next action. This generates transitions from the reference frame of the particles. A particle is presented a state (observation) s, takes an action a and receives an immediate reward r for that action. It then "sleeps" until it is picked again and receives a new state s'. Between these two observations, the environment has changed, as other particles have moved. The transitions now contain additional information, e.g. that slow particles tend to remain stationary while fast particles tend to move forward. This algorithm is used when the `distinguishable_particles` parameter is set to `True`. It requires a temporary storage of the current state and action for each particle for later use when the particle is picked again. A step by step implementation is provided in pseudocode in algorithm 5.

When different speeds and multiple networks are used, algorithm 5 has to be slightly modified. Instead of Q, $\hat{Q}$, $\mathcal{D}$, $\mathcal{L}$ and a single AdamW optimizer for Q's parameters θ, we now have an array of n such objects for each of these modules. The correct objects are chosen in each timestep based on the particle's velocity which is extracted from the state.

save and load

The `save` method all model parameters and collected metrics logs the ID of the newly saved model. The `load` method loads the model parameters and metrics of a previously saved model. This allows for easy saving and loading of trained agents. When no ID is provided during

Algorithm 5 DQN with delayed transitions and shared networks (difference to algorithm 3 highlighted in blue)

```
Initialize replay buffer $\mathcal{D}$ to capacity $N$
Initialize Q-network $Q$ with parameters $\theta$
Initialize target network $\hat{Q}$ with parameters $\theta^- = \theta$
Initialize environment to get initial state $s_1$, initial particle ID $i_1$
Initialize $\epsilon$ to $\epsilon_0$
Initialize temporary storage $\mathcal{M}$ for $(s, a, r)$ tuples indexed by particle ID
for steps $t = 1, \dots, T$ do
    ▷ Generate new transition
    Perform forward pass through $Q$ to get action-values $Q(s_t, a')$
    With probability $\epsilon$ select a random action $a_t$, otherwise select $a_t = \operatorname{argmax}_a Q(s_t, a)$
    Execute action $a_t$ in environment and observe reward $r_t$, next state $s_{t+1}$ and ID $i_{t+1}$
    $\mathcal{M}[i_t] \leftarrow (s_t, a_t, r_t)$
    if $i_{t+1} \in \mathcal{M}$ then
        Store transition $(^*\mathcal{M}[i_{t+1}], s_{t+1})$ in $\mathcal{D}$
    ▷ Update Q-network parameters
    Sample random minibatch of transitions $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s'})$ from $\mathcal{D}$
    Set $\boldsymbol{y} = \boldsymbol{r} + \gamma \max_{a'} \hat{Q}(\boldsymbol{s'}, a')$
    Calculate mean loss $\mathcal{L} = \text{mean}(\mathcal{L}(\boldsymbol{y}, Q(\boldsymbol{s}, \boldsymbol{a})))$
    Perform optimization (AdamW) of $\theta$ using backpropagation of $\mathcal{L}$
    ▷ Update target network parameters
    Update target network parameters $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$
    ▷ Update exploration rate
    Decrease $\epsilon$
```

loading, a table is shown with all available models and their properties. The user can then select a model to load by entering its ID. The `train_and_save` method provides a wrapper for automatically saving the model after training.

run

The `run` method is a stripped down version of the `train` method that only runs the simulation for a specified number of time steps, omitting all operations only necessary for parameter optimization. A greedy policy is used instead of the epsilon-greedy policy used during training. This method is used for evaluation of trained agents.

5.2.6 Playground

The `Playground` class implements a playground for testing the behavior of trained agents. When run, the user chooses a saved model and is presented with a window of two grids of the same size as the model's observation size. The left grid is empty except for the agent in the middle and can be filled by the user. Clicking a grid cell adds or removes an agent and clicking and dragging adds an agent with a velocity proportional to the drag distance. The right grid shows the same state after the agent has taken an action. Figure 5.3 shows a screenshot of the playground.

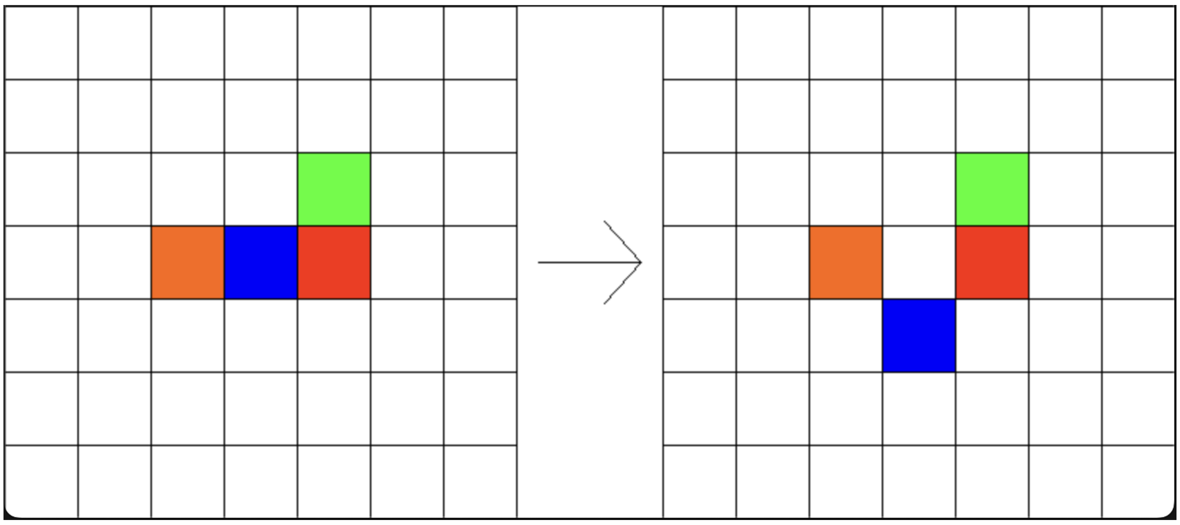


Figure 5.3: Screenshot of the playground. The state contains one slow particle blocking the agent, a fast particle above the slow one and a particle with average velocity behind the agent. The agent has learned to move around the slow particle in the direction where it's not blocked. More complicated states could be tested e.g. to test the agent's ability to make intelligent decisions based on the speeds and positions of the surrounding particles.

5.3 Hyperparameter Optimization

Optimizing the hyperparameters of a reinforcement learning algorithm is a difficult, but important task. Previous research has shown that the choice of hyperparameters can have a significant impact on the performance of the algorithm [80, 81, 82].

The goal of hyperparameter optimization, or hyperparameter tuning, is to find a set of hyperparameters that maximizes the cumulative reward of the agent. This means that in order to evaluate a set of hyperparameters, we have to train a new agent with these hyperparameters and evaluate its performance by running the simulation for a number of time steps.

As the choice of one hyperparameter can impact the optimal values of the other hyperparameters, the hyperparameters have to be optimized simultaneously, for example using a grid search or a random search. These algorithms are embarrassingly parallel, as the evaluation of a set of hyperparameters is independent of the evaluation of other sets, but the number of sets that have to be evaluated grows exponentially with the number of hyperparameters. This makes hyperparameter optimization a computationally expensive task.
Due to time constraints, a less computationally expensive approach was chosen for this thesis. Instead of performing a grid search or similar algorithm, the hyperparameters were first roughly optimized by a combination of reference to the literature, intuition and trial and error. Then, starting from this set of hyperparameters, the hyperparameters were optimized one by one, using the best value of a hyperparameter for all further tuning of other hyperparameters. This approach is not guaranteed to find the optimal hyperparameters, but it is computationally cheap and can yield good results. Furthermore, it shows how sensitive the algorithm is to the individual hyperparameters. This knowledge could be used to identify the most important hyperparameters and perform a more thorough optimization of these hyperparameters.

The following eight hyperparameters were optimized:

- **Learning rate** of the AdamW optimizer.

- **Discount factor** γ.
- **Replay buffer size**.
- **Batch size** for training.
- **Target network update rate** τ.
- **Exploration-exploitation tradeoff** via the decay constant η of the exploration rate $\epsilon(t) = \epsilon_{\text{end}} + (\epsilon_{\text{start}} - \epsilon_{\text{end}})e^{-n_{\text{steps}}/\eta}$.
- **Neural network architecture** (number of hidden layers and neurons per layer).
- **Activation function** of the neural network.

It is important to note, that there isn't a single set of hyperparameters that is optimal for deep Q-learning in general. Instead, the optimal hyperparameters depend on the environment and the task. Various studies have shown that reinforcement learning algorithms are very sensitive to the choice of hyperparameters and even the state-of-the-art RL algorithms often fail to generalize beyond their specific task [81, 80, 83, 84]. This means that the hyperparameters have to be optimized for each environment and task individually. In this section, we will discuss the hyperparameters that were optimized for the smart TASEP with a simple reward structure that includes only the `social_reward` option on top of the default reward (see section 5.2.2). A relatively high particle density of $\rho = 0.5$ was chosen and particle speeds were sampled from a uniform distribution in the range $(0, 1)$.

The starting point for the hyperparameter optimization was the following set of hyperparameters:

- **Learning rate**: 0.005
- **Discount factor** γ: 0.8
- **Replay buffer size**: 10^6 ($\geq$ number of training steps)
- **Batch size**: 64
- **Target network update rate** τ: 0.005
- **Epsilon decay constant**: 10^5
- **Neural network architecture**: 2 hidden layers with 128 neurons each
- **Activation function**: ReLU (output layer uses identity activation)

In order to evaluate the performance of the algorithm, the mean reward collected over 10000 time steps was measured in regular intervals of 2500 training steps. For each new hyperparameter value, 100 such episodes of training followed by evaluation were run, training the agents for a total of 250000 optimization steps. The results were plotted with a moving average over 8 and 30 episodes to smooth out the noise. The plots are shown in figure A.1 and A.2 in appendix A.

The following insights were gained from the hyperparameter optimization:

Epsilon decay

For this specific reward structure, the exploration-exploitation trade off is not very important. In fact, the experiment suggests that the agent performs slightly better without any exploitation at all. The dense, chaotic environment paired with the random initial weights apparently provides enough exploration to find a good policy even when acting greedily. This is not surprising, as the agent is not required to learn a complex policy, but only to move forward when possible and avoid collisions. A more thorough evaluation of the policies in the playground could provide more insight into the agent's behavior. I expect the reward to be only a rough measure of the agent's "intelligence", as it is possible to achieve a high reward by simply waiting for the other particles to move out of the way and moving forward whenever possible. We would expect a more intelligent agent to actively avoid collisions and to treat slow particles differently from fast particles. Examples of such behavior will be analyzed in the next chapter. A decay constant of 10000 was chosen for the following experiments, as it allows for some exploration in the beginning, but quickly converges to a greedy policy.

Discount factor

We see that discount factors in the range $0.5 - 0.9$ perform best. A slightly smaller discount factor of 0.3 still performs reasonably well, but a discount factor of ≤ 0.1 leads to a significant drop in performance. If the discount factor is chosen too large, the performance seems to oscillate around a lower mean reward. A discount factor of 0.9 was chosen for the following experiments. Agents trained with this discount factor seem to learn slightly slower than agents trained with a smaller discount factor, but the performance at the end of training is better. I would also expect these agents to learn more complex policies, as they have to take into account the future rewards of their actions.

Activation function

The ReLU and LeakyReLU activation functions perform best, which is not surprising, as they are the most commonly used activation functions in deep learning. The Softsign and tanh functions perform worse and the Sigmoid function performs worst. It is surprising to see that the Sigmoid function performs so much worse than the hyperbolic tangent, as they are rescaled versions of the same function. The tanh should also saturate slightly faster than the Sigmoid function, as it is steeper around the origin. I would expect it to be more prone to vanishing gradients than the Sigmoid function.
The ReLU function was chosen in favor of the LeakyReLU function, as it seems to perform equally well and is slightly faster to compute.

Learning rate

As expected, large learning rates lead to unstable training and poor performance. They fail to find minima as they keep overshooting, leading to low rewards. Optimal learning rates seem to be smaller than 0.01 with small variance for values in that range. The optimum seems to be around 0.001, which is small enough to be stable, but still large enough to allow for fast learning.

Batch size

The batch size seems to be inversely proportional to the speed of convergence of the algorithm. Larger batch sizes converge significantly slower than smaller batch sizes. This can be explained partly by the fact that agents with larger batch sizes start learning later, as they have to collect more transitions before they can collect the first batch and partly by the fact that for smaller batch sizes, the correlation between the transitions in a batch is smaller. Moreover, the scalar loss, which is propagated back through the network, is calculated as the mean loss over the batch. This means that the magnitude of the gradients does not depend on the batch size. The optimum seems to be around 32. With this batch size, the algorithm converges quickly and reaches the highest mean reward in the end.

Replay buffer size

As expected, larger replay buffers perform significantly better than smaller ones. If the buffer is too small, the algorithm fails to converge to a reasonable policy, as not enough independent transitions are stored in the buffer. Larger buffers perform better, saturating of course at the point where the buffer size becomes as large as the total number of transitions that are collected during training. For future experiments, the buffer should be chosen as large as the total number of training steps, if possible.

Target network update rate

The target network update rate seems to have a threshold somewhere between 10^{-5} and 10^{-6} below which the performance drops significantly. Above this threshold, the performance seems to be only weakly dependent on the update rate, with smaller update rates being more robust. A very large update rate of 0.5 still produces reasonable results, but the performance oscillates a lot during training. A target network update rate of 0.005 was chosen for the following experiments, as it seems to be a good compromise between performance and stability.

Neural network architecture

In order to test the performance of different neural network architecture, networks with 0-3 hidden layers and different numbers of neurons per layer were trained. A purely linear network with no hidden layers performs very poorly, as expected. Networks with only 6 neurons per layer also perform poorly, although of course much better than the linear network. Interestingly, when only using 6-node hidden layers, the performance seems to decrease with the number of layers. This could be due to the fact that the network is not able to learn a good policy with only 6 neurons per layer, but the additional layers increase the number of parameters that have to be optimized. The best overall performance can be seen for networks with 24 neurons per hidden layer, with more layers performing slightly better than fewer layers. These networks reach almost the same performance as the larger networks by the end of the training, but converge much faster, as fewer parameters have to be optimized. A network with 2 hidden layers and 24 neurons per layer appears to be the best overall choice.

Summary

Although a more thorough hyperparameter optimization could be performed, the analysis allows us to gain some meaningful insight into the smart TASEP's sensitivity to different hyperparameters and the ranges of their optimal values. Figure 5.4 compares the average performance of agents trained with the initial set of hyperparameters and the optimized parameters respectively. We can see that the initial set of hyperparameters was already quite good, achieving convergence to the same mean reward as the optimized set of hyperparameters. However, the optimized set of hyperparameters converges much faster, allowing for an earlier stop of the training.

Note that the acquired set of hyperparameters is only optimal for this specific reward structure. As already mentioned, reinforcement learning algorithms are very sensitive to the choice of hyperparameters and the optimal hyperparameters depend on the environment and the task. In the environment used for this experiment, the conditions stay the same throughout the whole simulation. Imagine for example a reward structure that encourages particle clustering or an environment, where the density is not constant over time. In that case, the epsilon decay constant would probably be an important hyperparameter, as the agent would have to exploit more often in order to generate transitions for the later stages of the simulation.

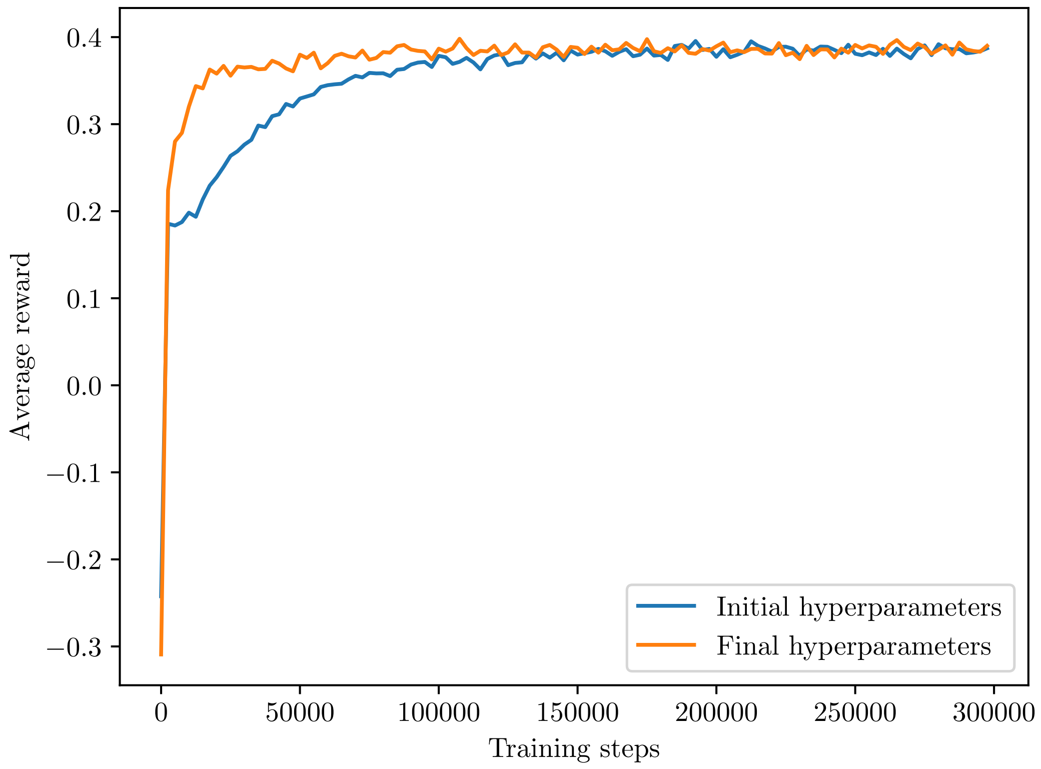


Figure 5.4: Average reward of smart TASEP agents trained with the initial set of hyperparameters and the final set of hyperparameters respectively. The plot is averaged over 10 separate trainings for each set of parameters.

Chapter 6

Results

This chapter will present the results of the smarticle experiments in the TASEP. We will start by setting a baseline with the classical TASEP and analyze how different speed distributions affect this purely stochastic system. Next, we will compare the results with a simple hard-coded policy, before moving on to the smarticle training. We will analyze the results of the smarticle training for different reward structures and compare them to the baseline.

6.1 Notes on measuring time and current

The classical TASEP is defined for continuous time. Each particle moves according to its own internal clock. In the numerical version treated in this thesis, time has to be discretized. The discrete time step should be defined in a way that allows time averages of observables in the discrete version and the continuous version to coincide. We base our time step definition on the fact that in the continuous version, the *average* number of jump attempts per second is constant. We therefore define our time step as a fixed number of *update attempts.* An update attempt is the picking of a random grid cell and the attempt to move the particle in that cell, *if it is occupied.* This means that also in the discrete version, the number of *jump attempts* per second is constant in the time average, while it can fluctuate over short time scales.

In this thesis, the time step is specifically defined as *one* update attempt. This makes it easy to define a particle *current*, which is independent of the system size. The current is defined as the number of actual forward jumps per *update attempt.* This of course has the dimension of a particle current *density*, but it is still called *current* in the lattice gas literature. It is the more useful quantity to work with, as it makes it easy to compare different systems.

Note that this definition of the time step differs from the prevalent definition of the Monte Carlo time step in the literature [85, ch. 3.2], where one time step is defined more naturally as one Monte Carlo sweep. A Monte Carlo sweep is defined as N update attempts, where N is the number of particles in the system.

6.2 Setting a Baseline: Classical TASEP

This section will use the classical 2D (T)ASEP as introduced in section 4.2. We will make it totally asymmetric in the horizontal direction (the *forward* direction) by setting the probability p to jump forward to $1/2$ and the probability q to jump backward to 0. The vertical direction (the *up/down* direction) will be symmetric with probabilities $a = b = 1/4$. The system has periodic boundary conditions in both directions. We will analyze a 128×32 system and a narrower 128×6 system. Both systems will be initialized with a checkerboard pattern of particles and holes, yielding a density of $\rho = 1/2$. This density will be chosen for most experiments, as it is dense enough for the different policies and speed distributions to have a significant effect on the system, but not so dense that the system is always jammed. Speeds will be drawn from a truncated normal distribution with mean 0.5 and different standard deviations σ. The distribution is truncated at 0 and 1, so that speeds are always in that range. Two example speed distributions are shown in figure 6.1.

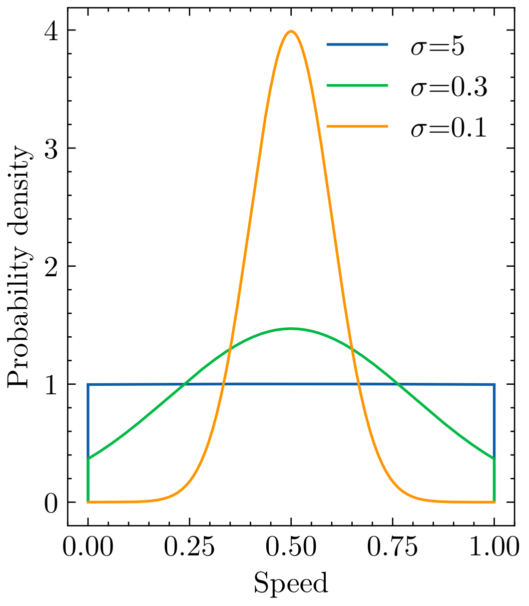


Figure 6.1: Normalized truncated normal distributions with mean $\mu = 0.5$ and different standard deviations σ.

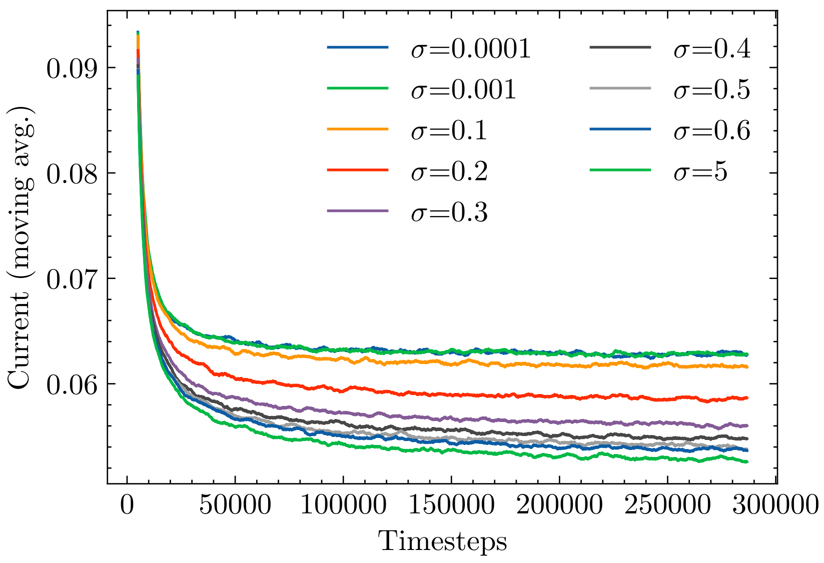
6.2.1 Finding the Steady State

When we want to compare the average current of different ASEP configurations, we have to make sure that the system has reached a steady state. The steady state has been reached when the current fluctuates around a constant mean value without an overall upwards or downwards trend. The time it takes to reach the steady state depends on the system size, the density and the initial conditions. When using a checkerboard setup for example, it's intuitive that the current will be very high in the beginning of the simulation, as every particle has an empty site in front of it and can move forward. As time passes, the probability of being able to move forward decreases (for different reasons that we will examine in the next paragraphs), and the mean current drops, asymptotically approaching the steady state current.

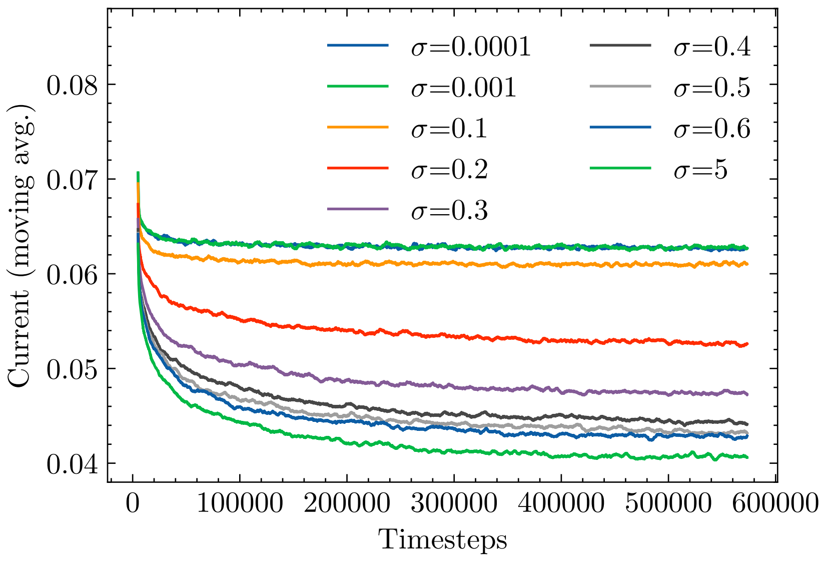
Figures 6.2a and 6.2b show the current as a function of time steps since initialization of the system to the checkerboard pattern for different speed distribution standard deviations σ. We can see that for small σ, when all particles have similar speeds, the current converges quickly and reaches a steady state after about 100,000-150,000 time steps. For larger σ, particles have different speeds and the equilibration phase takes longer, as jams form and dissolve and slow particles move very rarely. For $\sigma = 5$, especially in the narrow system (Fig. 6.2b), we see that the steady state is still not quite reached after 300,000 time steps.

One could expect the equilibration time to be roughly proportional to the number of Monte Carlo sweeps that have passed. This however is not the case, as we can see by comparing figures 6.2a and 6.2b. The narrow system has almost 11 times fewer particles than the wide system and thus performs about 11 times more Monte Carlo sweeps in the same number of time steps. However, the equilibration time is not 11 times shorter, but seems to be about the same, or even longer. We can conclude that the number of time steps is a better measure of the equilibration time than the number of Monte Carlo sweeps.

For the following experiments, the steady state current will be calculated by running the simulation for 600,000 time steps and averaging the current over the last 150,000 time steps. This ensures that the system has reached a steady state even for the configurations that take longer to equilibrate.



(a) System size: 128x32



(b) System size: 128x3

Figure 6.2: Current as a function of time steps since initialization of the system to the checkerboard pattern for different speed distribution standard deviations σ and two different system sizes. The data is averaged over 800 independent runs and plotted with a moving average over 5000 time steps.

6.3 Steady State Current as a Function of System Size and Speed Distribution

Now that we have established a definition of what is to be measured and how this measurement can be done from the simulation data, we can check if the measured steady state current is in accord with our intuition and how it depends on the system parameters.

6.3.1 Theoretical Expectation

In order to derive an expression for the expected value of the current, we closely examine what happens during one time step. This will help us to understand the probability of an actual forward move happening during one time step, which, by our definition, is the current. The following steps are performed during one time step:

1. Pick a random grid cell.
2. If the cell is occupied (probability $p_{occ} = \rho = 0.5$), perform a move attempt as follows:
3. Check the speed: The move attempt is continued with probability $p_{spd} =$ speed.
4. If the move attempt is continued, pick a direction (forward has probability $p = 1/2$).
5. If the target cell is empty (probability $p_{emp} \approx 1 - \rho$), perform the move.

Where the expression $p_{emp} \approx 1-\rho$ only holds for an approximately uniform density distribution throughout the system. The average speed $\bar{v} = /barp_{spd}$ is the expectation value μ of the speed distribution, which is set to 0.5 in all experiments. The total probability of a forward move in one time step is the product of the probabilities of the individual steps:

$$\begin{aligned} \langle J \rangle &:= p_{occ} \cdot p_{spd} \cdot p \cdot p_{emp} \\ &\approx p \cdot \mu \cdot \rho(1-\rho) = \left(\frac{1}{2}\right)^4 \\ &= 0.0625. \end{aligned} \tag{6.1}$$

More thorough derivations of this expression for ASEP systems with equal speeds can be found in the literature, for example in [85, section 2.3.2]. Equation 6.1 also confirms that our choice of the density $\rho = 1/2$ is optimal for maximizing the current in the system, as the maximum of the function $f(\rho) = \rho(1-\rho)$ is at $\rho = 1/2$.

6.3.2 Results of the Simulation

Figure 6.3 shows the steady state current as a function of the speed distribution's standard deviation σ for different system sizes. The steady state current has been obtained as explained previously and the data is averaged over 800 independent runs. For all system sizes, we observe a constant maximum current for $\sigma \ll 1$ that is only very slightly above the expected value of 0.0625. The slight increase is probably due to the fact that the steady state is only reached in the limit of infinite time. Although being very close, after 450,000 time steps, the current is still dropping a tiny amount, as can be seen in figure 6.2b.
As the width of the speed distribution grows to values of $\sigma \approx 1$, the current drops quickly before asymptotically converging to the minimum value, following a horizontally inverted logistic function. The logistic curve's midpoint is at $\sigma_0 \approx 0.2$ and the width of the transition is $\Delta\sigma \approx 1$. This makes sense, as the speed distribution is truncated at 0 and 1, so for values of σ close to 0, all particles have speeds close to 0.5 and the current is close to its maximum value. When σ grows, the speeds in the system get more and more diverse, until σ reaches 1, where the speed distribution is almost uniform and doesn't change much anymore.
The current drops although the mean speed in the system is still the same, because the

probability of a particle moving forward is not only dependent on its own speed, but also indirectly on the speeds of the particles in front of it. The slow particles bottleneck the current as the fast particles have to wait for them to move forward, creating a jam. This effect is more pronounced in narrower systems, as small jams block a higher proportion of the system. In wide systems, particles can avoid jams by moving around them, and even if there's a jam in almost every lane, the probability of these jams to all be in the same column is very low. In narrow systems, this can happen, forming a wall of slow particles, which is hard to penetrate. This is clearly visible in figure 6.3, where the total drop in current is much higher for narrow systems.

Looking back at equation 6.1, we can also try to explain this behavior theoretically. When jams form, the density in the system is not uniform anymore. Instead, there are regions of high density (jams) and regions of low density (empty lanes). This increases the mean density in the neighborhood of a random particle, decreasing $p_{emp} = (1 - \rho)$ and thus the probability of a forward move.

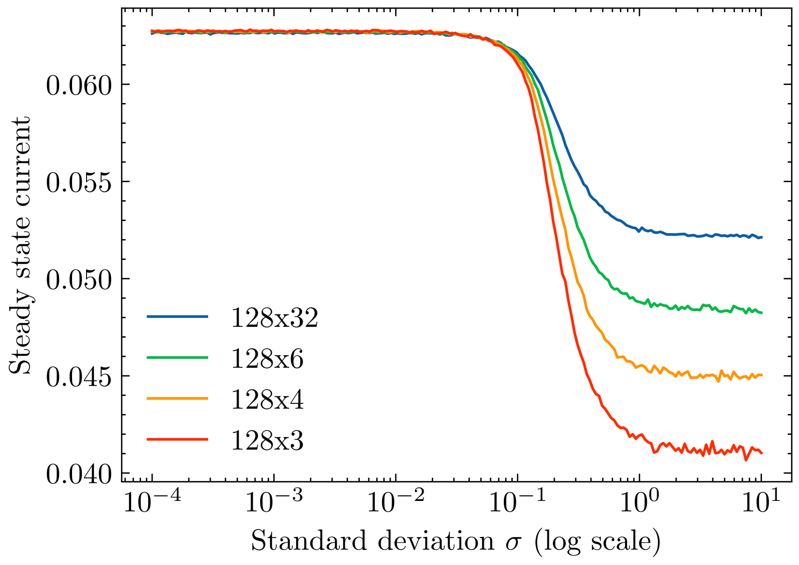


Figure 6.3: Steady state current as a function of the speed distribution's standard deviation σ for different system sizes. Steady state current is calculated as the average current over all remaining time steps after the steady state has been reached (the current has stopped dropping). The data is averaged over 800 independent runs.

6.4 A Naive Policy

We have now established a baseline current for different system sizes and speed distributions. Before we move on to the smarticle training to try to optimize the current with reinforcement learning, let's see if we can improve the current with a simple hard-coded policy.
One of the simplest and yet effective policies is to always move forward if possible. Especially when we choose σ to be very small and all particles have similar speeds, this policy should be very effective, as it keeps the density uniform and avoids jams. Furthermore, it is very easy to implement without writing any new code by just setting the probability p to move forward to 1 and all other probabilities (a, b, q) to 0.

Figure 6.4 compares the steady state current of this policy with the current from the last experiment, again as a function of the width of the truncated normal distribution. We can see that for small σ, our policy drastically improves the current, approximately doubling it. For larger σ, the current drops as before and drops even further, resulting in only about half the current of the baseline for $\sigma \gg 1$.

The reason for the extreme drop in current is that the policy is not able to deal with jams at all. The current in each lane is limited to the speed of the slowest particle in that lane. The resulting measured current then is the average of the currents in all lanes.

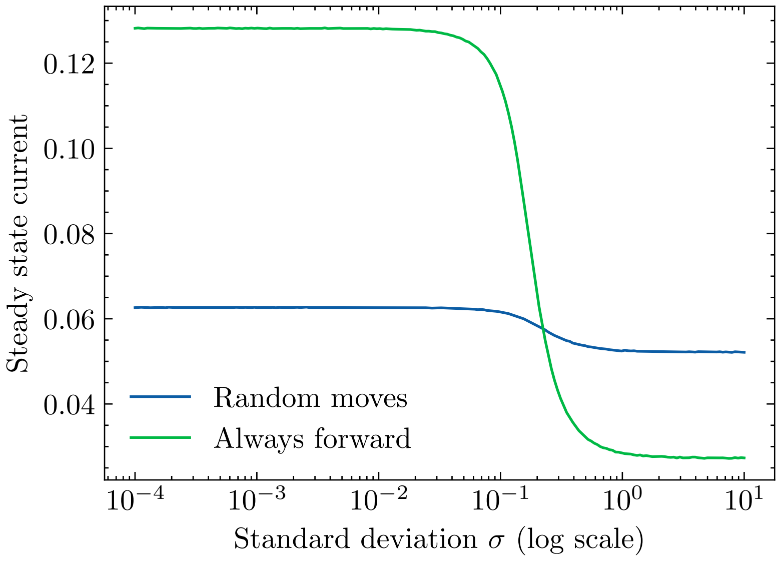


Figure 6.4: Steady state current as a function of the speed distribution's standard deviation σ for two different ASEP configurations. "Random moves" has a probability of $p = 1/2$ to move forward probabilities $a = b = 1/4$ to move up/down. "Always forward" has $p = 1$ and $a = b = 0$. A system of size 128x32 was used. The data is averaged over 800 independent runs.

The current achieved by this policy for small σ is even more interesting. It can be used as a new, very strong baseline for the smarticle training in systems with equal speeds. At first glance, it might even seem like the theoretical maximum current. Finding this maximum analytically is not trivial and goes beyond the scope of this thesis. However, we can try to find an upper bound for the current by looking at equation 6.1 again:

$$\begin{aligned}\max(\langle J \rangle) &= \max(p_{occ} \cdot p_{spd} \cdot p \cdot p_{emp}) \\ &\leq \max(p_{occ}) \cdot \max(p_{spd}) \cdot \max(p) \cdot \max(p_{emp}) \\ &= \rho \cdot \mu \cdot 1 \cdot \max(p_{emp}) \\ &= 0.25 \cdot \max(p_{emp}),\end{aligned}$$

where $\max(x)$ is the maximum expected value of x at a random time step, for a randomly picked particle. It therefore represents the maximum time-averaged value of x that can be achieved by any policy. I have set $p_{occ} = \rho$, $p_{spd} = \mu$, as they are external parameters that cannot be optimized by a policy. The policy cannot influence which particle is picked at a given time step, so it cannot influence p_{occ}. The *average* speed of a random particle will also always be the mean speed μ. The maximum of p that any policy can have is obviously 1 (in general, this will not be a probability when using a policy, but the decision will be based on the particle's observation, but the maximum fraction of forward decisions is still 1).
The last factor, $max(p_{emp})$, is the hardest to approximate. There are configurations where p_{emp} reaches its maximum value of 1, but these configurations cannot last longer than one time step. In the checkerboard configuration for example, every particle has an empty site in

front of it, so $p_{emp} = 1$. However, as soon as one particle moves forward, two particles are immediately next to each other and p_{emp} decreases. We therefore know that $\max(p_{emp}) < 1$. On the other hand, we know that $\max(p_{emp}) \geq 0.5 = (1 - \rho)$, as this upper bound is reached by the "always forward" policy. This places the upper bound for the current at

$$0.25 > \max(\langle J \rangle) \geq 0.25 \cdot (1 - \rho) = 0.125. \tag{6.2}$$

In order to keep p_{emp} above $1 - \rho$, which it would be for a randomized configuration, a policy has to intelligently fight against the disorder inevitably introduced by the random picking of particles. This is especially hard for systems with equal speeds, as the density in these systems has to be kept uniform, while also moving forward as often as possible.
In the regime of high σ, where the "always forward" policy performs very poorly, p_{emp} becomes a very interesting point of attack for a current-optimizing policy. When particles have very different speeds, a policy can organize the system in a way that slow particles are more densely packed than fast particles, increasing p_{emp} for the fast particles while decreasing it for the slow particles. This increases the net current, as the fast particles can fulfill their potential to move forward more often. We will investigate this in more detail in the next sections.

6.5 Smarticles for Current Optimization

This section will present the results of different smarticle trainings for slightly different goals. The main goal is always to maximize the current, but we will different approaches to achieve this goal, utilizing many of the features of the python package introduced in section 5.2. Although the deep Q-learning algorithm is not deterministic e.g. due to the random initialization of the neural network weights, the results presented here have shown to be reproducible. Appendix B contains the code that is needed to reproduce the results with the smart TASEP python package.

6.5.1 First Training: Equal Speeds

As a first experiment, we will try to maximize the current in a system with equal speeds. We will use a system of size 128×24 and a speed distribution with $\sigma = 10^-4$. The reward structure for this first experiment is very simple and includes only the `social_reward` in addition to the default reward. Optimal hyperparameters for this reward structure have already been found in section 5.3. The training was run for 500,000 time steps, with a reset of the environment after every 100,000 time steps.

Figure 6.5 shows a screenshot of the real-time visualization of the training after $\approx 350,000$ time steps. We can see that the training is almost converged at this point, confirming that $500,000$ time steps is enough to reach convergence for this problem. After the training, the simulation was run with a greedy policy for another 1.5 million time steps to measure the steady state current. The results are shown in figure 6.6.

We can see that the steady state is reached fairly quickly compared to the stochastic policies. The current fluctuates around a mean value of 0.152, which is about 143% higher than the baseline and about 22% higher than the "always forward" policy. The amplitude of the fluctuations is about 0.03, which is about 20% of the mean value, but the lowest current measured is still higher than the "always forward" policy's current.

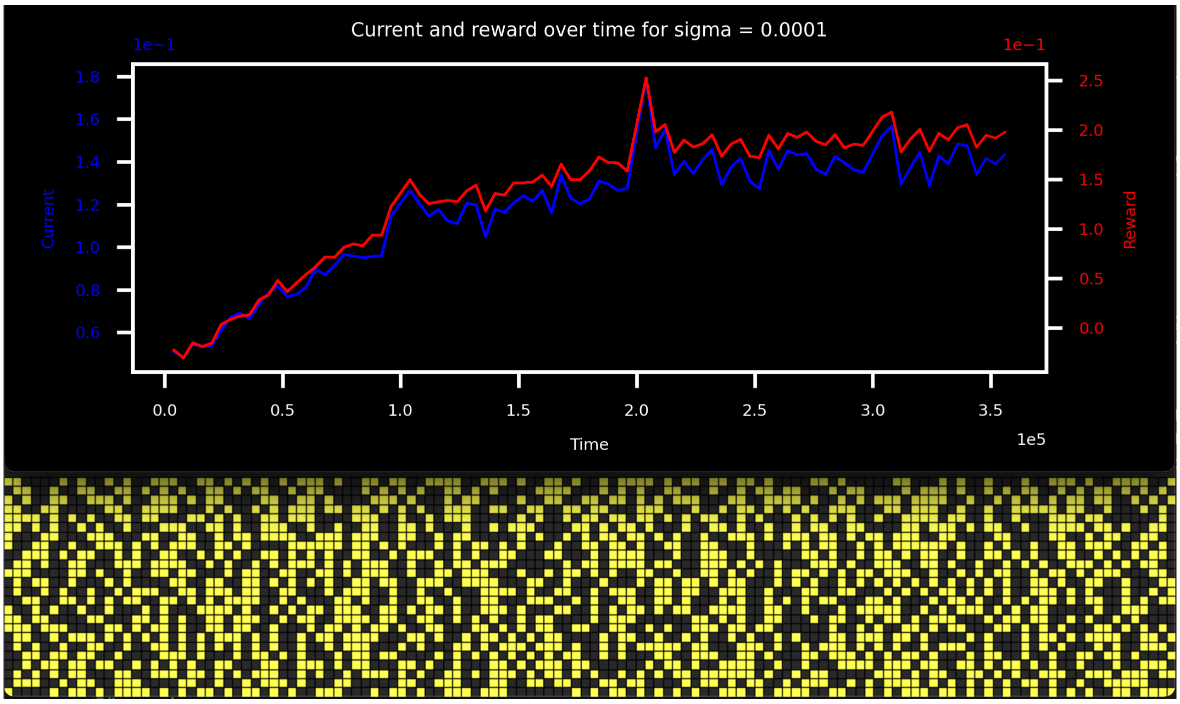


Figure 6.5: A screenshot of the real-time visualization of the first training, taken after $\approx 350,000$ time steps. All particles have the same speed of 0.5, which can also be seen from their yellow color. We can see that the algorithm has almost converged at this point.

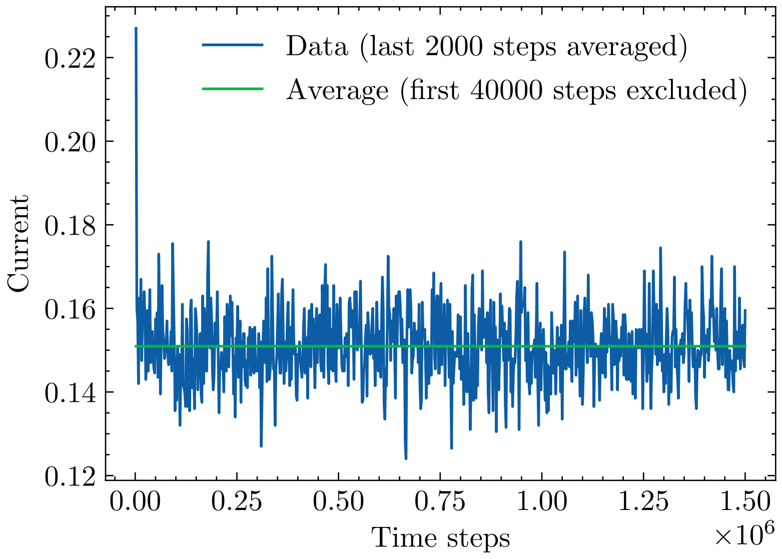


Figure 6.6: Steady state current as a function of the simulation time for the first trained smarticle agent. The green line shows the average current, which is at about 0.152, outperforming not just the baseline by about 143%, but also the “always forward” policy by about 22%.

Bibliography

[1] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Fourth edition. Pearson series in artificial intelligence. Hoboken: Pearson, 2021. ISBN: 978-0-13-461099-3.

[2] *Artificial intelligence.* In: *Wikipedia.* Page Version ID: 1185402635. Nov. 16, 2023. URL: https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=1185402635 (visited on 11/16/2023).

[3] Elaine Woo. *John McCarthy dies at 84; the father of artificial intelligence.* Los Angeles Times. Section: Obituaries. Mar. 20, 2014. URL: https://www.latimes.com/local/obituaries/la-me-john-mccarthy-20111027-story.html (visited on 11/16/2023).

[4] Scott L. Andresen. "John McCarthy: Father of AI". In: *IEEE Intelligent Systems* 17.5 (Sept. 1, 2002). Publisher: IEEE Computer Society, pp. 84–85. ISSN: 1541-1672. DOI: 10.1109/MIS.2002.1039837. URL: https://www.computer.org/csdl/magazine/ex/2002/05/x5084/13rRUxE04ph (visited on 11/16/2023).

[5] *What is AI? / Basic Questions.* URL: http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html (visited on 11/16/2023).

[6] John McCarthy et al. *A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE.* URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html (visited on 11/16/2023).

[7] *Google Books Ngram Viewer.* URL: https://books.google.com/ngrams/graph?content=AI&year_start=1800&year_end=2019&corpus=en-2019&smoothing=3 (visited on 11/16/2023).

[8] SITNFlash. *The History of Artificial Intelligence.* Science in the News. Aug. 28, 2017. URL: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/ (visited on 11/16/2023).

[9] *AI Spring? Four Takeaways from Major Releases in Foundation Models.* Stanford HAI. URL: https://hai.stanford.edu/news/ai-spring-four-takeaways-major-releases-foundation-models (visited on 11/16/2023).

[10] Geoffrey Hinton et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012). Conference Name: IEEE Signal Processing Magazine, pp. 82–97. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2205597. URL: https://ieeexplore.ieee.org/document/6296526 (visited on 11/16/2023).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems.* Vol. 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (visited on 11/16/2023).

[12] *A decade in deep learning, and what's next.* Google. Nov. 18, 2021. URL: https://blog.google/technology/ai/decade-deep-learning-and-whats-next/ (visited on 11/16/2023).

[13] Bryan House. *2012: A Breakthrough Year for Deep Learning.* Deep Sparse. July 17, 2019. URL: https://medium.com/neuralmagic/2012-a-breakthrough-year-for-deep-learning-2a31a6796e73 (visited on 11/16/2023).

[14] *Introducing ChatGPT.* URL: https://openai.com/blog/chatgpt (visited on 11/16/2023).

[15] *ChatGPT.* URL: https://chat.openai.com (visited on 11/16/2023).

[16] *What Is Moore's Law and Is It Still True?* Investopedia. URL: https://www.investopedia.com/terms/m/mooreslaw.asp (visited on 11/16/2023).

[17] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986). Number: 6088 Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 11/16/2023).

[18] *9 ways we use AI in our products.* Google. Jan. 19, 2023. URL: https://blog.google/technology/ai/9-ways-we-use-ai-in-our-products/ (visited on 11/16/2023).

[19] Robin Burke, Alexander Felfernig, and Mehmet H. Göker. "Recommender Systems: An Overview". In: *AI Magazine* 32.3 (June 5, 2011). Number: 3, pp. 13–18. ISSN: 2371-9621. DOI: 10.1609/aimag.v32i3.2361. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2361 (visited on 11/16/2023).

[20] *ElevenLabs: AI Voice Generator & Text to Speech.* URL: https://elevenlabs.io/ (visited on 11/16/2023).

[21] *Midjourney.* Midjourney. URL: https://www.midjourney.com/home?callbackUrl=%2Fexplore (visited on 11/16/2023).

[22] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016). Number: 7587 Publisher: Nature Publishing Group, pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://www.nature.com/articles/nature16961 (visited on 11/16/2023).

[23] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* Dec. 5, 2017. DOI: 10.48550/arXiv.1712.01815. arXiv: 1712.01815[cs]. URL: http://arxiv.org/abs/1712.01815 (visited on 11/16/2023).

[24] Kelsey Piper. *AI triumphs against the world's top pro team in strategy game Dota 2.* Vox. Apr. 13, 2019. URL: https://www.vox.com/2019/4/13/18309418/open-ai-dota-triumph-og (visited on 11/16/2023).

[25] *Statistical mechanics.* In: *Wikipedia.* Page Version ID: 1194581857. Jan. 9, 2024. URL: https://en.wikipedia.org/w/index.php?title=Statistical_mechanics&oldid=1194581857 (visited on 01/13/2024).

[26] Étienne Fodor et al. "How far from equilibrium is active matter?" In: *Physical Review Letters* 117.3 (July 13, 2016), p. 038103. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.117.038103. arXiv: 1604.00953[cond-mat]. URL: http://arxiv.org/abs/1604.00953 (visited on 01/13/2024).

[27] Frank Cichos et al. "Machine learning for active matter". In: *Nature Machine Intelligence* 2.2 (Feb. 2020). Number: 2 Publisher: Nature Publishing Group, pp. 94–103. ISSN: 2522-5839. DOI: 10.1038/s42256-020-0146-9. URL: https://www.nature.com/articles/s42256-020-0146-9 (visited on 01/13/2024).

[28] Gabriel Popkin. “The physics of life”. In: *Nature* 529.7584 (Jan. 1, 2016). Number: 7584 Publisher: Nature Publishing Group, pp. 16–18. ISSN: 1476-4687. DOI: `10.1038/529016a`. URL: `https://www.nature.com/articles/529016a` (visited on 01/13/2024).

[29] C. Kaspar et al. “The rise of intelligent matter”. In: *Nature* 594.7863 (June 2021). Number: 7863 Publisher: Nature Publishing Group, pp. 345–355. ISSN: 1476-4687. DOI: `10.1038/s41586-021-03453-y`. URL: `https://www.nature.com/articles/s41586-021-03453-y` (visited on 01/13/2024).

[30] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. “Kilobot: A low cost scalable robot system for collective behaviors”. In: *2012 IEEE International Conference on Robotics and Automation*. 2012 IEEE International Conference on Robotics and Automation. ISSN: 1050-4729. May 2012, pp. 3293–3298. DOI: `10.1109/ICRA.2012.6224638`. URL: `https://ieeexplore.ieee.org/document/6224638` (visited on 01/13/2024).

[31] Tiago M. Fernández-Caramés and Paula Fraga-Lamas. “Towards The Internet of Smart Clothing: A Review on IoT Wearables and Garments for Creating Intelligent Connected E-Textiles”. In: *Electronics* 7.12 (Dec. 2018). Number: 12 Publisher: Multidisciplinary Digital Publishing Institute, p. 405. ISSN: 2079-9292. DOI: `10.3390/electronics7120405`. URL: `https://www.mdpi.com/2079-9292/7/12/405` (visited on 01/13/2024).

[32] Vijay Balasubramanian. “Brain power”. In: *Proceedings of the National Academy of Sciences of the United States of America* 118.32 (Aug. 10, 2021), e2107022118. ISSN: 0027-8424. DOI: `10.1073/pnas.2107022118`. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8364152/` (visited on 01/13/2024).

[33] *Spot*. Boston Dynamics. URL: `https://bostondynamics.com/products/spot/` (visited on 01/14/2024).

[34] Anton Koval, Samuel Karlsson, and George Nikolakopoulos. “Experimental evaluation of autonomous map-based Spot navigation in confined environments”. In: *Biomimetic Intelligence and Robotics* 2.1 (Mar. 1, 2022), p. 100035. ISSN: 2667-3797. DOI: `10.1016/j.birob.2022.100035`. URL: `https://www.sciencedirect.com/science/article/pii/S2667379722000018` (visited on 01/14/2024).

[35] *Collective animal behavior*. In: *Wikipedia*. Page Version ID: 1193934250. Jan. 6, 2024. URL: `https://en.wikipedia.org/w/index.php?title=Collective_animal_behavior&oldid=1193934250` (visited on 01/14/2024).

[36] Eric Bonabeau. “Agent-based modeling: Methods and techniques for simulating human systems”. In: *Proceedings of the National Academy of Sciences* 99 (suppl_3 May 14, 2002). Publisher: Proceedings of the National Academy of Sciences, pp. 7280–7287. DOI: `10.1073/pnas.082080899`. URL: `https://www.pnas.org/doi/10.1073/pnas.082080899` (visited on 01/13/2024).

[37] Carter Bays. “Introduction to Cellular Automata and Conway’s Game of Life”. In: *Game of Life Cellular Automata*. Ed. by Andrew Adamatzky. London: Springer, 2010, pp. 1–7. ISBN: 978-1-84996-217-9. DOI: `10.1007/978-1-84996-217-9_1`. URL: `https://doi.org/10.1007/978-1-84996-217-9_1` (visited on 01/14/2024).

[38] Nick Gotts. “Emergent Complexity in Conway’s Game of Life”. In: *Game of Life Cellular Automata*. Ed. by Andrew Adamatzky. London: Springer, 2010, pp. 389–436. ISBN: 978-1-84996-217-9. DOI: `10.1007/978-1-84996-217-9_20`. URL: `https://doi.org/10.1007/978-1-84996-217-9_20` (visited on 01/13/2024).

[39] Paul Rendell. “Turing Universality of the Game of Life”. In: *Collision-Based Computing*. Ed. by Andrew Adamatzky. London: Springer, 2002, pp. 513–539. ISBN: 978-1-4471-0129-1. DOI: `10.1007/978-1-4471-0129-1_18`. URL: `https://doi.org/10.1007/978-1-4471-0129-1_18` (visited on 01/14/2024).

[40] *Elementary knightship - ConwayLife.com.* URL: https://conwaylife.com/forums/viewtopic.php?f=2&t=3303 (visited on 01/14/2024).

[41] Uwe C. Täuber. *Critical Dynamics: A Field Theory Approach to Equilibrium and Non-Equilibrium Scaling Behavior.* Cambridge: Cambridge University Press, Mar. 6, 2014. 488 pp. ISBN: 978-0-521-84223-5.

[42] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook.* Cham: Springer International Publishing, 2018. ISBN: 978-3-319-94462-3 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0. URL: http://link.springer.com/10.1007/978-3-319-94463-0 (visited on 11/18/2023).

[43] *Explained: Neural networks.* MIT News — Massachusetts Institute of Technology. Apr. 14, 2017. URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (visited on 11/16/2023).

[44] *What are Neural Networks? — IBM.* URL: https://www.ibm.com/topics/neural-networks (visited on 11/16/2023).

[45] Alexandre Gonfalonieri. *Understand Neural Networks & Model Generalization.* Medium. Jan. 29, 2020. URL: https://towardsdatascience.com/understand-neural-networks-model-generalization-7baddf1c48ca (visited on 11/18/2023).

[46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Cambridge, Massachusetts: The MIT Press, Nov. 18, 2016. 800 pp. ISBN: 978-0-262-03561-3.

[47] Simon Haykin. *Neural Networks: A Comprehensive Foundation: A Comprehensive Foundation: United States Edition.* Subsequent Edition. Upper Saddle River, NJ: Pearson, July 1, 1998. 842 pp. ISBN: 978-0-13-273350-2.

[48] Tyler Elliot Bettilyon. *Deep Neural Networks As Computational Graphs.* Teb's Lab. May 5, 2020. URL: https://medium.com/tebs-lab/deep-neural-networks-as-computational-graphs-867fcaa56c9 (visited on 11/23/2023).

[49] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark.* June 28, 2022. DOI: 10.48550/arXiv.2109.14545. arXiv: 2109.14545[cs]. URL: http://arxiv.org/abs/2109.14545 (visited on 11/23/2023).

[50] Ryan P Adams. "Computing Gradients with Backpropagation". In: ().

[51] Louis B. Rall, ed. *Automatic Differentiation: Techniques and Applications.* Red. by G. Goos et al. Vol. 120. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1981. ISBN: 978-3-540-10861-0 978-3-540-38776-3. DOI: 10.1007/3-540-10861-0. URL: http://link.springer.com/10.1007/3-540-10861-0 (visited on 11/30/2023).

[52] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition.* Google-Books-ID: qMLUIsgCwvUC. SIAM, Nov. 6, 2008. 448 pp. ISBN: 978-0-89871-659-7.

[53] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* June 15, 2017. DOI: 10.48550/arXiv.1609.04747. arXiv: 1609.04747[cs]. URL: http://arxiv.org/abs/1609.04747 (visited on 01/01/2024).

[54] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 01/01/2024).

[55] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization.* Jan. 4, 2019. DOI: 10.48550/arXiv.1711.05101. arXiv: 1711.05101[cs,math]. URL: http://arxiv.org/abs/1711.05101 (visited on 01/01/2024).

[56] *What is Machine Learning? — IBM.* URL: https://www.ibm.com/topics/machine-learning (visited on 12/03/2023).

[57] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction.*

[58] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://dl.acm.org/doi/10.1145/203330.203343 (visited on 12/08/2023).

[59] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* Dec. 19, 2013. DOI: 10.48550/arXiv.1312.5602. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 12/08/2023).

[60] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning.* Dec. 13, 2019. DOI: 10.48550/arXiv.1912.06680. arXiv: 1912.06680[cs,stat]. URL: http://arxiv.org/abs/1912.06680 (visited on 12/08/2023).

[61] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (Sept. 1, 2013). Publisher: SAGE Publications Ltd STM, pp. 1238–1274. ISSN: 0278-3649. DOI: 10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721 (visited on 12/08/2023).

[62] OpenAI et al. *Learning Dexterous In-Hand Manipulation.* Jan. 18, 2019. DOI: 10.48550/arXiv.1808.00177. arXiv: 1808.00177[cs,stat]. URL: http://arxiv.org/abs/1808.00177 (visited on 12/08/2023).

[63] *Part 1: Key Concepts in RL — Spinning Up documentation.* URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html (visited on 12/08/2023).

[64] Richard Serfozo. "Markov Chains". In: *Basics of Applied Stochastic Processes.* Ed. by Richard Serfozo. Probability and Its Applications. Berlin, Heidelberg: Springer, 2009, pp. 1–98. ISBN: 978-3-540-89332-5. DOI: 10.1007/978-3-540-89332-5_1. URL: https://doi.org/10.1007/978-3-540-89332-5_1 (visited on 12/12/2023).

[65] *Part 2: Kinds of RL Algorithms — Spinning Up documentation.* URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below (visited on 12/17/2023).

[66] Christopher Watkins. "Learning From Delayed Rewards". In: (Jan. 1, 1989).

[67] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1, 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698 (visited on 12/17/2023).

[68] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015). Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 12/17/2023).

[69] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning.* July 5, 2019. DOI: 10.48550/arXiv.1509.02971. arXiv: 1509.02971[cs,stat]. URL: http://arxiv.org/abs/1509.02971 (visited on 12/17/2023).

[70] Tom Schaul et al. *Prioritized Experience Replay.* Feb. 25, 2016. DOI: 10.48550/arXiv.1511.05952. arXiv: 1511.05952[cs]. URL: http://arxiv.org/abs/1511.05952 (visited on 12/19/2023).

[71] Carolyn T. MacDonald, Julian H. Gibbs, and Allen C. Pipkin. "Kinetics of biopolymerization on nucleic acid templates". In: *Biopolymers* 6.1 (1968). _eprint: https://onlinelibrary.wiley.com/doi/p pp. 1–25. ISSN: 1097-0282. DOI: 10.1002/bip.1968.360060102. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/bip.1968.360060102 (visited on 12/22/2023).

[72] Frank Spitzer. “Interaction of Markov processes”. In: *Advances in Mathematics* 5.2 (Oct. 1, 1970), pp. 246–290. ISSN: 0001-8708. DOI: 10.1016/0001-8708(70)90034-4. URL: https://www.sciencedirect.com/science/article/pii/0001870870900344 (visited on 12/19/2023).

[73] Gunter M. Schütz. “Exact solution of the master equation for the asymmetric exclusion process”. In: *Journal of Statistical Physics* 88.1 (July 1, 1997), pp. 427–445. ISSN: 1572-9613. DOI: 10.1007/BF02508478. URL: https://doi.org/10.1007/BF02508478 (visited on 12/22/2023).

[74] R. A. Blythe and M. R. Evans. “Nonequilibrium Steady States of Matrix Product Form: A Solver’s Guide”. In: *Journal of Physics A: Mathematical and Theoretical* 40.46 (Nov. 16, 2007), R333–R441. ISSN: 1751-8113, 1751-8121. DOI: 10.1088/1751-8113/40/46/R01. arXiv: 0706.1678[cond-mat]. URL: http://arxiv.org/abs/0706.1678 (visited on 12/19/2023).

[75] Dmytro Goykolov. “ASYMMETRIC SIMPLE EXCLUSION PROCESS IN TWO DIMENSIONS”. In: (2007).

[76] Jonas Märtens. *jonasmaertens/TASEP*. original-date: 2023-10-05T16:07:08Z. Dec. 22, 2023. URL: https://github.com/jonasmaertens/TASEP (visited on 12/23/2023).

[77] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: a LLVM-based Python JIT compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: https://dl.acm.org/doi/10.1145/2833157.2833162 (visited on 12/24/2023).

[78] Greg Brockman et al. *OpenAI Gym*. June 5, 2016. DOI: 10.48550/arXiv.1606.01540. arXiv: 1606.01540[cs]. URL: http://arxiv.org/abs/1606.01540 (visited on 12/28/2023).

[79] Albert Bou et al. *TorchRL: A data-driven decision-making library for PyTorch*. Nov. 27, 2023. DOI: 10.48550/arXiv.2306.00577. arXiv: 2306.00577[cs]. URL: http://arxiv.org/abs/2306.00577 (visited on 01/01/2024).

[80] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. Jan. 29, 2019. DOI: 10.48550/arXiv.1709.06560. arXiv: 1709.06560[cs,stat]. URL: http://arxiv.org/abs/1709.06560 (visited on 01/07/2024).

[81] Jack Parker-Holder et al. “Automated Reinforcement Learning (AutoRL): A Survey and Open Problems”. In: *Journal of Artificial Intelligence Research* 74 (June 1, 2022), pp. 517–568. ISSN: 1076-9757. DOI: 10.1613/jair.1.13596. arXiv: 2201.03916[cs]. URL: http://arxiv.org/abs/2201.03916 (visited on 01/07/2024).

[82] *AutoML — AutoRL: AutoML for RL*. URL: https://www.automl.org/blog-autorl/ (visited on 01/07/2024).

[83] Logan Engstrom et al. *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*. May 25, 2020. DOI: 10.48550/arXiv.2005.12729. arXiv: 2005.12729[cs,stat]. URL: http://arxiv.org/abs/2005.12729 (visited on 01/09/2024).

[84] Marcin Andrychowicz et al. *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. June 10, 2020. DOI: 10.48550/arXiv.2006.05990. arXiv: 2006.05990[cs,stat]. URL: http://arxiv.org/abs/2006.05990 (visited on 01/09/2024).

[85] George Lawrence Daquila. “Monte Carlo analysis of non-equilibrium steady states and relaxation kinetics in driven lattice gases”. In: ().

Hyperparameter Optimization Plots (moving avg. of 8)

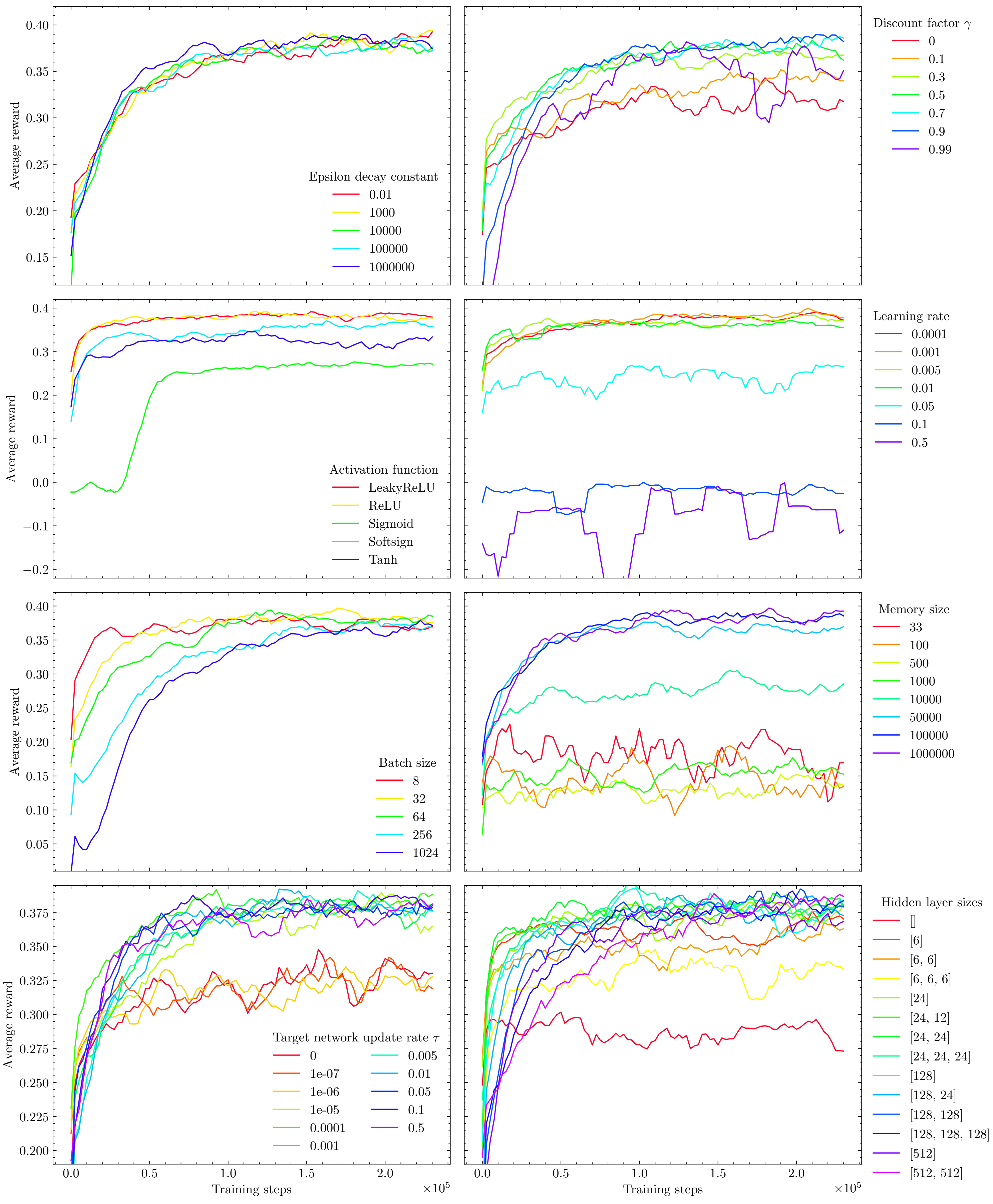


Figure A.1: Hyperparameter optimization plots analyzed in section 5.3.

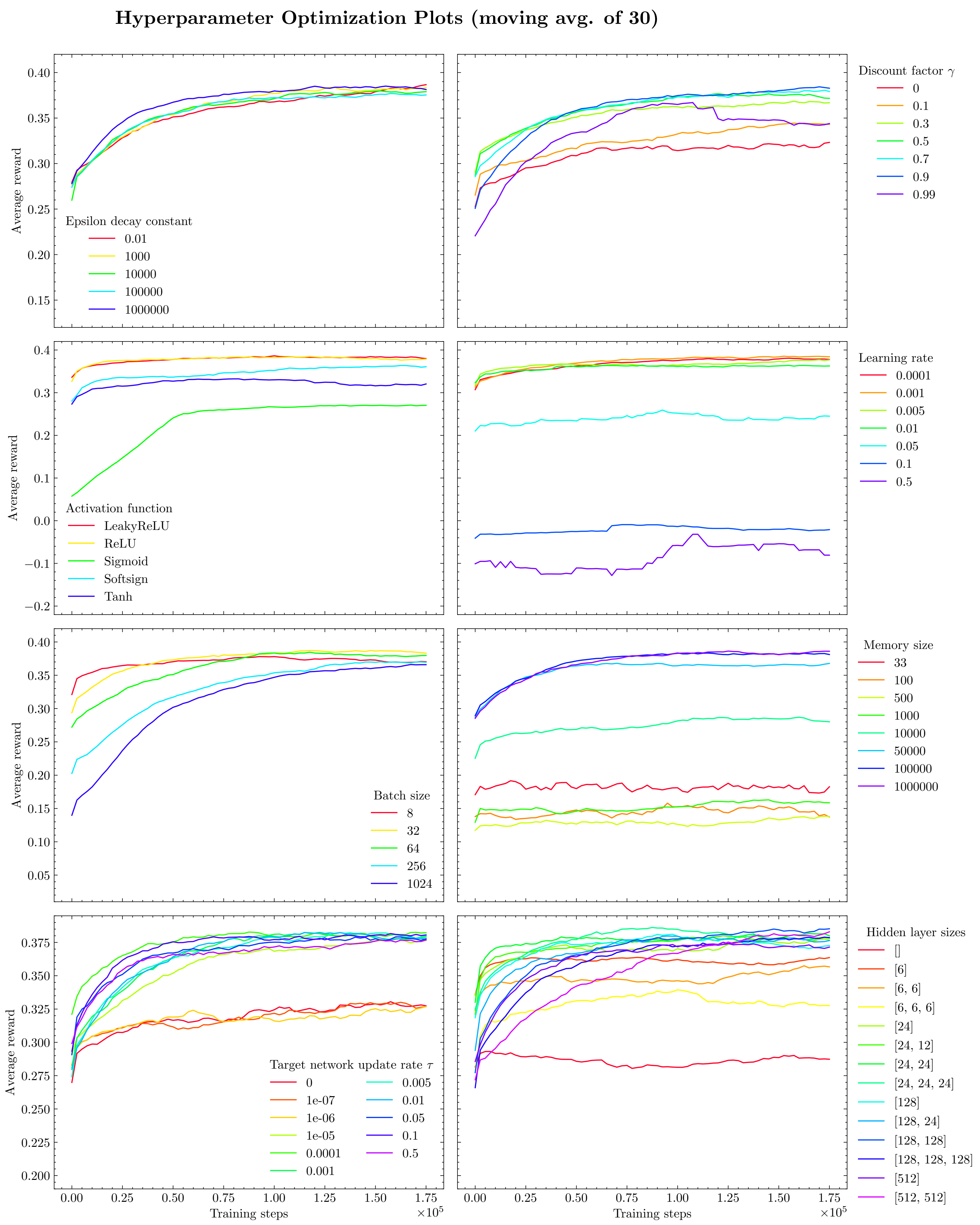


Figure A.2: Hyperparameter optimization plots analyzed in section 5.3.

Appendix B

Code

only for reproducibility rest of code is on github, packasge easy to use setup like this

B.1 First training

Selbständigkeitserklärung

Ich versichere hiermit, die vorliegende Arbeit mit dem Titel

Titel der Arbeit

selbständig verfasst zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Jonas Märtens

München, den 15. Januar 2024