UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Master Thesis

Using Reinforcement Learning for Personalization

Creation of an application to be used in pain therapy

Spring Term 2022

Department of Mathematics
University of Fribourg
Switzerland

Supervised by:
Prof. Dr. Christian Mazza
Post-Doc Xavier Richard

Author:
Jana Kolly

Contents

Symbols

ϵ	probability of taking a random action in an ϵ-greedy policy
α	step-size parameter
γ	discount-rate parameter
λ	decay-rate parameter for eligibility traces
s, s'	states
a	an action
r	a reward
$\mathcal{S}$	set of all states
$\vert\mathcal{S}\vert$	number of elements in $\mathcal{S}$
$\mathcal{A}$	set of all actions
$\vert\mathcal{A}\vert$	number of elements in $\mathcal{A}$
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
h_t	the history of states, actions and rewards until time step t, $h_t = (s_0, a_0, r_1 ..., s_t, a_t)$
O_t	the history of actions and rewards until time step t, $O_t := (a_0, r_1, a_1, r_2, ..., a_{t-1}, r_t)$
t	discrete time step
T	final time step of an episode
A_t	action at time t
S_t	state at time t, typically due to S_{t-1} and A_{t-1}
R_t	reward at time t, typically due to S_{t-1} and A_{t-1}
G_t	return following time t
π	policy, decision making rule
$\pi(a\vert s)$	probability of taking action a in state s under policy π
$p(s', r\vert s, a)$	probability of transition to state s' with reward r, from state s and action a
$p(s'\vert s, a)$	probability of transition to state s', from state s taking action a
$r(s, a)$	expected immediate reward from state s after action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy

d	dimensionality - the number of components of $\mathbf{w}$
$\mathbf{w}$, $\mathbf{w}_t$	d-vector of weights underlying an approximate value function
$\hat{q}(s, a, \mathbf{w})$	approximate value of state-action pair s, a given weight vector $\mathbf{w}$
$\nabla\hat{q}(s, a, \mathbf{w})$	column vector of partial derivatives of $\hat{q}(s, a, \mathbf{w})$ with respect to $\mathbf{w}$
$\mathbf{x}(s, a)$	vector of features visible when in state s taking action a, short: $\mathbf{x}$
$\mathbf{w}^\top\mathbf{x}$	inner product of vectors, $\mathbf{w}^\top\mathbf{x} = \sum_i w_i x_i$
δ_t	temporal-difference (TD) error at t
$\mathbf{z}_t$	d-vector of eligibility traces at time t
m	$\lvert\mathcal{A}\rvert$, number of different screen configurations
n	number of tuning parameter for one screen
k	parameter k from the k-th order history

Chapter 1

Introduction

Countless people all over the world are suffering from pain. This may be due to an acute injury caused by an accident or also more complex pain patterns, such as chronic pain. Treatment of pain is very complex and should be adapted to each patient individually. Patients often suffer not only from the pain symptoms themselves, but also from the side effects - caused by traditional medication treatment. On top of that, patients often suffer from accompanying symptoms, such as fatigue, depression, stress, lack of concentration, or existential fears [1]. All this impairs the patients in all activities of their daily life. Psychological pain management offers great potential to influence and improve not only the primary but also the secondary symptoms. Psychological pain management techniques are applied, among others, in alternative healing methods, such as hypnosis, meditation, color and music-therapy. The stimulation techniques used in these methods possess the ability to influence the pain sensation and to transfer the patients into a state of well-being [2]. Such stimulation techniques therefore offer great potential in the management of pain. Despite their potential, they are not yet widely used.

Machine learning is currently getting a lot of attention. It is a thriving field and offers potential for innovation in almost every area. Especially reinforcement learning, as a subfield of machine learning, is becoming more and more popular. The purpose of reinforcement learning is to teach a machine (or computer) an arbitrary task purely through "reinforcement", by sending positive or negative feedback in response to the machine's decisions [3]. It is used today in a variety of areas, such as personalizing advertising banners or teaching computers to play games, such as chess, at a world-record level [4]. Especially in personalization and recommender systems, reinforcement learning offers an attractive tool.

In the context of this thesis, one is interested in exploring the opportunities that reinforcement learning can offer to support the treatment of pain, as well as to alleviate accompanying symptoms. It has therefore been approached to develop a self-learning mobile application, which uses techniques from psychological pain management and thereby sends various stimulations to the user of the application. A built-in reinforcement learning algorithm personalizes the stimuli to each user. Such a personalized application would be beneficial, as it provides a tool, which is therapist-independent and fully personalized. In addition to this, it provides the opportunity to support the patients at any given time, without increased therapist involvement, saving both time and resources.

This is a large project, which brings together the utilization of a mobile application framework, a game engine and a reinforcement learning algorithm, providing personalized stimuli, based on real-time user-feedback, in order to influence the sensations of pain and general mental state. This thesis serves as the first steps in this innovative project.

The main goal is to explore the possibilities by first providing the necessary foundations of psychology and reinforcement learning topics, furthermore specifying the requirements for the application, as well as designing and developing a first prototype. The focus is laid on describing the initial situation and a detailed problem definition. This includes a mathematical modeling of the learning task, as well as a description of the requirements for a suitable reinforcement learning. Based on the requirements, a decision for a specific algorithm is made. This decision is justified, and the final algorithm is explained in detail. In addition, the theoretical learning effect and the functionality of a first prototype are presented. However, testing the application on real patients is not in the scope of this work. The thesis, with its prototype, is considered as a starting point for further developments.

Thesis Structure
This first chapter describes the background and motivation of the thesis. Chapter 2 provides basic knowledge, starting with an introduction to the topic of pain and methods of psychological pain management, continuing with an introduction to reinforcement learning. Chapter 3 describes the requirements engineering of the application, including a detailed mathematical modeling of the learning problem and a description of the learning algorithm requirements and its selection. In chapter 4 the process of designing, developing, and evaluating the application prototype is provided. Chapter 5 concludes the thesis with a discussion of the results and suggestions for further extensions.

Chapter 2

Fundamentals

2.1 Pain and Psychology

This chapter provides an introduction to the topic of pain. It also addresses the limitations of current treatment methods for pain patients. In a further step, some methods of psychological pain management are introduced.

2.1.1 Pain and Forms of Treatment

This section is primarily based on [1], as well as [5].
Pain is a sensory and emotional experience that is essential for survival. It tells us when we are injuring our bodies and is a warn signal to prevent further injury, while being unpleasant. In general, there are three main stages of pain perception. The first stage is the sensation of pain, followed by the second stage in which signals are transmitted from the periphery via the peripheral nervous system to the dorsal horn located in the spinal cord. Finally, in the third stage, the signals are transmitted to the higher brain via the central nervous system.

Pain has a multifaceted nature, is uniquely experienced by each individual and can even be influenced by multiple factors, such as culture, situations, and past experiences. It is therefore essential to adapt the pain treatment to each sufferer individually. Until today, providing an adequate pain management is still very difficult. Often, a traditional therapy with pain-killers is not sufficient. This is especially the case for *chronic pain*, where the pain lasts for more than 3 to 6 months and for *neuropathic* pain, where no tissue injury is present (anymore). Neuropathic pain results from abnormal activity of the nervous system and occurs when the nerves that transmit pain signals are spontaneously activated. A classic example for neuropathic pain is the phantom limb pain, which is a painful sensation in a part of the body that is no longer present. Not only for patients suffering from neuropathic or chronic pain, but indeed for most pain sufferers, the primary treatment with medication is not sufficient. The reason why medication treatment alone is often not sufficient for the patients, is not only because it often does not lead to complete pain relief, but also because the suffering caused by the pain brings with it other accompanying symptoms, which severely impact the patient in all his activities of daily living. Typical such accompanying symptoms are depression, anxiety, existential fears, fatigue and sleep disturbances. These symptoms, in turn, have a negative effect on the overall physical condition and can even intensify the perception of pain.

This variety of symptoms makes it practically impossible for one medication or form of therapy alone to be fully sufficient. Therefore, the *multidisciplinary approach* to managing pain has become very popular. In the multidisciplinary approach, different forms of therapy are combined and added to the classical medication treatment. Such forms are, for instance, hypnosis, meditation and biofeedback. The aim is to find the best combination of therapies to help the patient better manage their pain.

Although the multidisciplinary approach was strongly promoted as early as the end of the Second World War, even today non-pharmacologic methods are still underutilized. One reason for this could be the lack of knowledge in society about their potential for pain relief. Even though it is known that by gaining control over fundamental body functions (for example as one does in meditation or relaxation therapy), one can take advantage of the descending neuronal pathways to reduce activity in the pain-processing neurons and thereby reducing the amount of pain perceived. Well-trained individuals in such techniques can reduce their heart rate, alter respirations, and reduce or even eliminate pain.

Some of the forms of therapy already mentioned, such as meditation and relaxation therapy, but also many others, such as neuro-linguistic programming or guided imagery, use similar techniques and benefit from effects of different stimuli, which can influence our perception of pain. In the following sections several such techniques and types of stimulation will be discussed in more detail.

2.1.2 Methods of Psychological Pain Management

The power of the psyche on our body and our sensations is impressive. Everything we think and feel has an effect on our physical structures. Due to the property of *neuroplasticity* of the human brain, new *synapses* are constantly formed [6]. Synapses are specialized connections that transmit chemical and electrical signals through the complex neural network, consisting of numerous interconnected neurons, in our body. Thus, new synapses are constantly being formed and the nervous system can therefore change and adapt to varying stimuli. If we now influence synapse formation by perceiving certain stimuli, we can change our brain through specific training [7].

The extent of the power of the psyche can be well illustrated by the example of operations that have been performed with no medical anaesthetics [8]. Using different techniques, patients manage to control their body to such an extent, that they can influence their pain sensation and endure otherwise extremely painful operations, completely without opioids. This demonstrates the potential of specific techniques, where we learn to understand and even influence our mind and body. We can train our mind and body to create new behavior [2].

We will now take a closer look at individual effects in the following two subchapters. First, the effect of different forms of stimulation are discussed, followed by an explanation of the effects of techniques such as coupling and suggestion.

2.1.2.1 Healing with Stimuli

One can categorize the types of stimuli into different modalities. A conventional classification is into *visual* (such as colors, distances, contrasts, brightness) *auditory* (such as volume, sounds, rhythms) and *kinesthetic* (such as temperature, vibrations, pressure, movements, textures) stimuli [7]. Each modality is processed differently by our brain. Not everyone reacts the same way to such stimuli and often one has a dominant category to which one generally reacts most intensively. This can be observed, for example, by looking at the different types of learners: some people learn better visually, i.e. with mind-maps and written text, others can remember things better if they are explained to them orally.

A brief description of individual stimuli and their effects follows.

Music [9][10][11]
Music has the property of triggering emotions. This happens because music stimulates our limbic system in the brain, which is responsible for emotions. Listening to music can affect our heartbeat, breathing rate, blood pressure and can also tense or relax muscles. This is due to the fact that music affects our hormones. Depending on what music we listen to, different hormones are released into our bodies, whereas the particular effect of the music depends on the emotional state of the listener. This access via hormones and emotions creates the possibility of therapeutic application of music in various fields, such as intensive care, birth assistance, rheumatology, and treatment of pain. Different studies have found that music plays a role in reducing chronic and acute pain, but there are still few studies on the exact neurochemical effects of music.

Colors [12][13][14]
Each color has a different effect on our psyche and our body, because each color has a specific wavelength and energy that is transmitted to the human. Theories suggest that colors with longer wavelengths (such as red) are perceived as exciting or warm, while colors with shorter wavelengths (such as blue) are perceived as relaxing or cool. To this day, however, there has been very little theoretical or empirical work on the effect of color on psychological functioning. Previous work has been guided mainly by practical considerations ("What color should I paint the walls in my restaurant, so that my guests get a bigger appetite?") rather than scientific rigor. Nevertheless, color and light therapy, in which the patient is exposed to colored light to heal physical and psychological illnesses, is widely known as an alternative healing method. What concrete emotions and reactions the different colors trigger in people is nevertheless something that is still very individual and also depends on the situation. Blue water, for example, can be calming, but seeing blue food is rather repulsive. So the reactions do not depend purely on the color, but also on the objects that carry the color as well as what the people associate the colors with. This makes research in this field a difficult task.

Shapes [15]
Symbols and geometrical shapes are always coupled to emotions and thus symbols and their *healing energies* are also used in therapeutic settings. They were already assigned great importance in ancient cultures. For example, specific body paint was applied for fertility, religious rituals, and also for healing purposes. Many shamans use the power of symbols in their rituals. In Buddhism and Hinduism, *mandalas*, a sequence of geometric shapes and patterns, serve as meditation objects. Today, icons and symbols are omnipresent, be it in company logos, signals in road traffic or even on advertising posters.

We are constantly exposed to symbols and shapes and we automatically couple them with emotions. Such emotions can then be harnessed in a therapeutic context. Most symbols are based on a few basic shapes such as circles, triangles, squares, waves and spirals.

Movements [7][16]
In the category *movement*, we want to focus specifically on the effects of *eye movements*. Every movement we make with our eyes is connected to a certain functioning of the brain. An exciting observation is the following. A common thing for many right-handed people is that when they create mental images of memories, their eyes tend to turn upward to the left. When they develop mental images of things they have never seen (for example imagination is involved and maybe even lying), their eyes go to the upper right. When they talk to themselves, their eyes go down and to the left, and when they feel deep emotions, their eyes go down to the right. In 1987, the therapy form *EMDR* ("Eye Movement Desensitization and Reprocessing") was created. As the name implies, this form of therapy uses eye movements, among other things, as a form of stimulation to process trauma and negative experiences or memories, for example. The origin of this method lays in the observation by the psychologist Francine Shapiro, of the seemingly calming effect of spontaneous, repeated eye movements when faced with unpleasant thoughts. During an EMDR session, the patient is asked to visualize an experience and to perform specific eye movements (for example rapid horizontal eye movements or circular movements). Bringing the event into consciousness stimulates the physiologically stored information and the accompanying eye movements activate the information processing mechanism. Thus, feelings associated with the memory can be re-programmed and the event can be processed.

2.1.2.2 Coupling and Suggestions

The two techniques described in this section are used in various forms of treatments, such as autogenic training, relaxation exercises, hypnotherapeutic sessions or neuro-linguistic programming. Various studies support the effect of such techniques for a significant pain relief [8][17]. The techniques are used, not only in the treatment of pain, but in a much broader range, such as stress relief, dealing with anxiety, insomnia or panic attacks. Essentially, there are no limits to the use of such techniques. In relation to the topic of pain, such techniques are therefore interesting, not only for the direct influence on the sensation of pain, but also for the treatment of the various accompanying symptoms mentioned in section 2.1.1. It is also important to note that there is no clear distinction between the techniques, and they are often used in combination. The modality of performing such techniques can be various, for example in accompanying sessions, guided by a therapist or also by instructions from a particular audio tape, or also in individual, independent sessions, using one's own imagination.

Suggestions [18][19][20]
Suggestions are the invitation to experience the world in a different way. Suggestions are *initiated* by instructions from a therapist or an audio tape. An example of such an initiation could be: "Imagine putting your right foot into a cold mountain lake. You feel the cold water making your foot numb. You can observe how the numbness spreads by itself: over the foot, up to the knee, ... ". The same can be initiated without external suggestions, via so-called *imagery*, where you guide yourself independently through such imaginary journeys. Through positive imagery and suggestions one can guide oneself to a place of power.

One can create an imaginary place of well-being, which is free from suffering and where one can recharge one's inner energy. Popular suggestions for the treatment of pain are imaginations in which one makes individual parts of the body numb and then spreads this numbness throughout the whole body. Other typical suggestions are imaginary body journeys, in which one puts oneself into one's own body and connects with individual cells or nerves. The goal is to create a *dissociation*: One tries to disconnect from the unpleasant situation or pain, and to observe it from a different angle. One can learn to eliminate feelings from the painful parts of the body and instead bring pleasant feelings into awareness. It is not equally straightforward for every person to focus on mental images. To a certain extent, such fantasy journeys can be learned, but there are people who are unable to imagine inner images. In this case, another technique should be chosen.

Coupling [18]
Pain or negative emotions are often difficult to grasp. No connection can be made between the symptom and a psychological condition, because pain is a sensation and not a tangible object. This is where the technique of *coupling* offers an opportunity, which is intended to make the symptom itself more tangible for the patient, thus also more controllable. The goal is to *objectify the intangible*, such as chronic pain, then to modify this newly created object, and thus to co-modify the underlying symptom. An example of such an objectification could be: "And now I would like you to think of your pain. Imagine that you could give this pain a certain color. What color might that be?" When asked this question, one automatically and intuitively thinks of a certain color. One therefore assigns objectifiable properties to the intangible symptom. In the previous example, one couples the pain to a particular color. After selecting an initial attribute, the association must be strengthened and then be linked to the symptom. This process is also called *pacing*. Back to our previous example, let us assume the color red has been chosen. Then the next step could be initiated as follows: "You now see that this pain has an intense red color." Once the link is made, one can assign other attributes to the intangible symptom, such as shapes, sounds, surfaces, temperature, and consistency. After having coupled the symptom with such attributes, one tries to change the symptom indirectly, by mentally changing the created and linked object. To do so, one mentally modifies the associated object. By changing the associated object, one tries to change the linked symptom along with it. It is also common to apply the same coupling process with a *healing-object*, which gets associated to a pain-free body-part or emotion: "Choose a body part that is currently completely pain-free. What color would you give to that body part?" These positive attributes can then be used to transform the negative attributes into the positive ones. For example, if the color yellow was chosen as the positive attribute, and one created and linked a red hot fireball to the negative symptom, one could try to visualize something similar to the following: "Liquid yellow beeswax is slowly trickling over the red hot fireball, gradually transforming it into a yellow air balloon that becomes colder and colder, and lighter and lighter, until it rises up into the air. It rises and rises and becomes smaller and smaller until, eventually, it is no longer visible." Through such coupling and visualization of transformations, one tries to influence the perception of pain.

2.2 Reinforcement Learning

This section will describe the main ideas and foundations of reinforcement learning, closely following Sutton and Barto [3], which is highly recommended for further reading. Starting with describing the general idea of reinforcement learning, we will then dive into the mathematical modelisation and the classical setting of a Markov decision process (MDP).

2.2.1 Machine Learning and Reinforcement Learning

We begin by briefly explaining what we understand under the term *machine learning* and how it differs from ordinary programming.
In programming, we tell a machine (or computer program) exactly *what* to do to solve a problem. We do this for example with classical if-else statements. The person writing the program knows, already at the time of writing, how exactly the system is going to behave in any situation.
In machine learning, on the other hand, the programmer tells the machine only *how* to improve to *produce a solution* to a problem. This implies that the machine can learn and adapt to new data without human intervention. It does so by interacting with the presented data. The goal of machine learning is therefore to automate the learning mechanism, so that it can be generalized to unseen applications.
Machine learning based techniques have already been successfully used in a number of different fields [21][22]: computers can detect cancer cells on medical scans, computers can learn to play chess (in 1997, the chess computer "DeepBlue" from IBM defeated the world champion [4]). Autonomous driving is also a big area of machine learning research. These different examples were given to show the huge spectrum of machine learning applications. The examples all sound rather futuristic to non-specialists and one could think that machine learning is a relatively young branch of research. However, this is not the case. Machine learning is strongly linked to classical statistical data analysis. Many algorithms are based on concepts from classical statistics, which have been known for a long time, such as curve-fitting problems [23].
As mentioned above, machine learning applications are widely spread. Therefore, it is common and useful to divide the learning problems into three primary paradigms [22]: supervised learning, unsupervised learning and reinforcement learning. A high-level description of the three areas follows.

- In *supervised learning*, a machine uses matched pairs of input and output values, in order to try to learn a general function for predicting the outputs based on the inputs. A typical example is the curve-fitting problem.
- In *unsupervised learning*, a machine seeks to discover previously unknown patterns in a data set without having any prior knowledge about the data. A typical example is a clustering-problem, where data points are assigned into groups (or clusters), so that every group has similar properties.
- In *reinforcement learning*, which will be the main focus for the further sections, the machine tries to learn its environment via interactions, so that it can choose the best actions for a given situation, in order to maximize a numerical reward signal. A typical example is learning a bot how to play a certain game, such as playing chess.

The classification into these paradigms is fluid, by what we mean that there are applications that fit into multiple paradigms or are even mixed forms. The classification is also not consistent among the literature, as for example [21] includes reinforcement learning among supervised learning and adds a semi-supervised paradigm on top.

2.2.2 Introduction to Reinforcement Learning

Having classified reinforcement learning as a main paradigm of machine learning, we now want to take a closer look at it.
The reinforcement learning paradigm, among the other paradigms, is probably the most intuitive one. We just need to think about how humans and animals learn and we already find most of the key concepts of reinforcement learning. When a child learns how to walk, it does that by trial-and-error. It tries out how to get onto its feet and how to move its body to keep the balance. The child has no teacher, who is telling it exactly how to move, but it exercises by its own. At the beginning it is barely able to get on its feet. But by exercising more and more and by learning from interactions, it eventually learns how to walk.
In this very natural example of a learning process, we have already encountered some important elements of the reinforcement learning: an *agent* (child) interacting with its *environment* (its room with the furniture, where physical laws like gravity apply), choosing *actions* (body movements) and adapting its *policy* (strategy for how to behave in different situations) by receiving *feedback* (emotional feedback) in form of *rewards* (pain when failing or pleasure when succeeding). In the following sections, those terms will get analyzed in more detail.

2.2.3 Reinforcement Learning Principles

Reinforcement learning is about learning through interaction how to behave in order to achieve a goal. In this process, an *agent* learns to interact with its *environment.* For this purpose, the agent can choose different *actions*, the environment reacts to these actions and as a result the agent ends up in a new *state.* In addition, the agent receives feedback for its chosen actions in the form of a *reward.* Thus, the actions are the choices the agent makes; the states are the basis for the choices, the rewards are the basis for evaluating the choices and the environment is the systems source of information about states and rewards. It defines the "rules of the game": the set of possible actions which are available for any state, how the state changes over time based on the chosen actions and the rewards associated with each action. The agent does not know the rules of the game. Instead, he must interact with the environment to gain more and more experience. The agent's goal is to maximize the amount of reward he receives over time. He must learn a strategy to achieve this. He does this via a so-called *policy.* A policy is a function that tells the agent which action to choose depending on the states. Therefore, the ultimate goal is to find an optimal policy, which maximizes the cumulative reward.

2.2.4 Markov Decision Process

Having introduced the basic principles, rather informally, we will know start to map the problem of learning a specific task into a well-known framework, the Markov decision process (MDP), and introduce a more mathematical notation. The MDP provides a framework for modeling sequential decision making. It is highly applicable in the setting of reinforcement learning and therefore widely used in this context [24]. We will concentrate on the setting of *finite* MDPs (set of possible actions, states and rewards are finite), even though many results can be extended to countable or even continuous settings.

2.2.4.1 Agent-Environment Interface

As mentioned earlier, reinforcement learning is about learning via interactions to finally reach a goal (for example learning how to walk). The object that learns and makes decisions is called the *agent*. The object upon which the agent interacts is called the *environment*. It essentially includes everything that is outside of the agent. Everything that is counted as part of the agent is fully known and controllable by the agent itself. Everything that counts to the environment is completely uncontrollable by the agent, but not necessarily fully unknown. Over time, the agent learns more and more about the environment in which it is operating. The process of interaction between the agent and the environment is shown in figure 2.1.

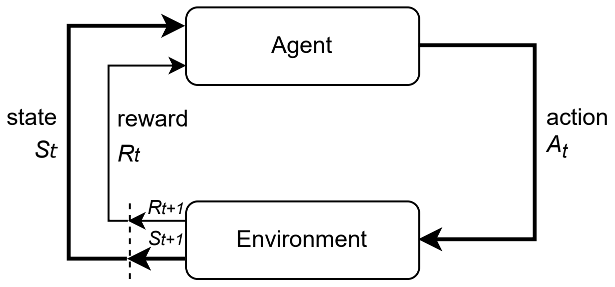


Figure 2.1: The agent-environment interaction in a Markov decision process (source: [3]).

We formulate this process mathematically as follows:

- Let $t \in \mathbb{N}$ be the discrete time step.
- Let $\mathcal{S}, \mathcal{A}, \mathcal{R}$ be the finite sets of all states, actions and rewards.
- At each time step t, the agent receives a state $S_t \in \mathcal{S}$.
- Given this state S_t, the agent decides for an action $A_t \in \mathcal{A}$.
- One time step later, in $t+1$, partially as consequence of the chosen action A_t, the agent receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$.
- The environment has now reached a new state S_{t+1}.

This flow naturally generates a *trajectory* of the following form, where S_t, A_t, R_t are random variables taking values in $\mathcal{S}, \mathcal{A}, \mathcal{R}$:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{2.1}$$

Notation:
We denote S_t, A_t, R_t in capital letters, to indicate that those are random variables, while their possible values will be denoted s, a and r.

For a finite MDP (with finite state, action and reward sets), the random variables S_t, A_t and R_t are defined via a discrete probability distribution, which follow the *Markov property* (eq. (2.2)):

- Let $h_t = (s_0, a_0, r_1 ..., s_t, a_t)$ be the history of states, actions and rewards until time step t.
- Let us write $P(x|y)$ the conditional probability of event x, provided that y is true.
- Then the probability of receiving reward r_{t+1} and reaching state s_{t+1} after having chosen action a_t is only a function of a_t and s_t, and not of the entire history h_t:

$$\forall h_t, a_t, r_{t+1}, s_{t+1} \quad P\left(s_{t+1}, r_{t+1} \mid h_t\right) = P\left(s_{t+1}, r_{t+1} \mid s_t, a_t\right). \tag{2.2}$$

.

Given the Markov property, the probability of each possible value for s_t and r_t depend on the immediately preceding state and actions (s_{t-1}, a_{t-1}) and not on all earlier observations. This means that the state must include all information about the past that is necessary for the future. If the state has this property, we say that the state is a *Markov state.*

Let us now introduce some functions, which will be useful later on.

We start with the *transition probability function* p, taking four arguments, $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, defining the probability of transitioning to state s' and receiving reward r, starting from state s and having chosen action a:

$$p(s', r|s, a) := P(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a) \tag{2.3}$$

Starting from the function p, we define the *state-transition probability function*, which will also be denoted by p, but takes only three arguments and defines the probability from starting in state s, choosing action a and then to transition to state s': $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$,

$$p\left(s' \mid s, a\right) := P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \tag{2.4}$$

We define as well the *expected rewards for state-action pairs* as a function taking two arguments: $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$,

$$r(s, a) := \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p\left(s', r \mid s, a\right). \tag{2.5}$$

2.2.4.2 Goals, Rewards and Returns

In reinforcement learning, the agent has a specific goal, which depends on the task he needs to complete. For the IBM DeepBlue robot, the goal was to learn how to play chess as good as possible [4]. To give another example, one could think of a problem, where we want to learn a robot how to escape from a 2D-maze. Here, the overall goal would be to find as fast as possible the exit of the maze, without knowing the landscape. To achieve this goal, at each time step t, the agent needs to decide in which direction to walk. In reinforcement learning, the environment sends a reward-signal to the agent, $R_t \subset \mathbb{R}$, evaluating the choice of the agent. The overall goal is not to gain the highest immediate reward possible, but to achieve the highest cumulative reward in the long run. Because we work with machines, we need to translate such goals and rewards into scalar signals. And because the agent strives to maximize its cumulative reward, it is crucial to define the rewards in a good way. In the maze-example, we want to find the exit as fast as possible. To encourage that, we should give negative rewards (e.g. -1) for each action that does not lead to the exit. And for the action (or actions) that leads to the exit, one could reward the agent with +1. If we do not punish the agent with negative rewards, what would happen? The agent, whose goal is to maximize the cumulative reward, would just walk around, collect rewards and stay forever in the maze, because it gives him a higher (or equal) reward as when he is quitting it. This shows why it is important to invest time in the definition of the reward signals to prevent the agent from finding loopholes and not aiming at the intended goal.

Let us define these concepts formally.
We introduce the notion of *episodes*. Very often, the interaction of the agent with the environment can naturally be broken down into subsequences. A subsequence could be a single chess-game or a trip in the maze. Such subsequences are called *episodes* and the final time step in such an episode will be called *final time step* T. The time of termination, T, is a random variable taking values in $\mathbb{N}$, which varies from episode to episode.
Given an episode and the corresponding sequence of returns R_1, R_2,...,R_T, we define the cumulative reward after the time step t as the *return* G_t,

$$G_t := R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{2.6}$$

In various tasks it can make sense to prioritize early rewards. This gives rise to the introduction of a *discount rate* $\gamma \in [0, 1]$ and the concept of the *discounted return*:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T \tag{2.7}$$

How we set the discount rate $\gamma \in [0, 1]$ can have a huge impact on the agent's behavior:

- If $\gamma = 0$, then the agents goal is to maximize the immediate reward: $G_t = R_{t+1}$ (all other terms in eq. (2.7) disappear).
- The more γ approaches 1, the stronger are future rewards taken into account.
- If $\gamma = 1$, then the agents goal is not prioritizing earlier rewards, but to count all rewards equally.

Having introduced the return G_t the agent's goal can now be defined as: at each time step t, choosing action A_t to maximize the expected (discounted) return $\mathbb{E}(G_t)$.

This leads us to the next section, where we formalize this concept even further, with the help of value functions.
But first, now that we have introduced all necessary concepts, let us give a formal definition of a Markov decision process:

Definition 2.2.4.1 *A* ***Markov decision process (MDP)*** *is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p, r, \gamma \rangle$, *where*

- $\gamma \in [0, 1]$ *is the discount rate,*
- $\mathcal{S}$ *is the set of states,* $\mathcal{A}$ *the set of actions,* $\mathcal{R}$ *the set of rewards (introduced in section 2.2.4.1),*
- p *the transition probability function from eq.* (2.3)*:* $p(s', r|s, a) := P(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)$,
- r *the reward function from eq.* (2.5)*:* $r(s, a) := \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right]$,

and where the ***Markov property*** *(see eq.* (2.2)*) holds: the probability of each possible value for* S_t *and* R_t *depend only on the immediately preceding state and actions* (S_{t-1}, A_{t-1}) *and not on all earlier observations.*

2.2.4.3 Policies, Value Functions and Optimality

We have learnt that the agent's overall goal in reinforcement learning is to always choose actions A_t to maximize the expected (discounted) return $\mathbb{E}(G_t)$. But how can he achieve this? In order to pursue this goal, the agent has to decide at each time step t, given a certain state, which action adds the most benefit. For this purpose, the *value function* will be introduced. The value function defines *how good* it is to be in a certain state. Before we can introduce this function, we need to understand the concept of a *policy.* A policy defines how the agent acts in any situation. It is a distribution function over actions, given states:

$$\pi : \mathcal{A} \times \mathcal{S} \to [0, 1], \quad \pi(a|s) := P(A_t = a|S_t = s).$$

A policy fully defines the behavior of an agent, as it says what action a to choose, given any state s. In the MDP setting, where the Markov property applies, the policies depend only on the state s and not on the time step t. This property is called *stationary*: $A_t \sim \pi\left(\cdot \mid S_t\right), \forall t > 0$.
The *state value function* $v_\pi : \mathcal{S} \to \mathbb{R}$ of an MDP is defined as the expected return starting from state s and afterwards following the policy π:

$$v_\pi(s) := \mathbb{E}_\pi\left[G_t \mid S_t = s\right]. \tag{2.8}$$

In some settings, it is helpful to define the value for the state-action pairs and not only for the states: The *state-action value function* $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as the expected return starting from state s, taking action a, and afterwards following policy π:

$$q_\pi(s, a) := \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]. \tag{2.9}$$

Solving an MDP in reinforcement learning consists therefore in finding the value function, which, for each state (or state-action pair), returns the maximum value: We introduce the *optimal state value function* $v_*(s) := \max_\pi v_\pi(s)$ and similarly the *optimal state-action value function* $q_*(s, a) := \max_\pi q_\pi(s, a)$, where the maximum is taken over all policies π.

Each policy π has a different value function v_π (resp. q_π). With the help of the value functions, we can therefore introduce an ordering of policies, which helps us to compare them. We say a policy π is better than or equal to a policy π', noted by $\pi \geq \pi'$, if $v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$. A policy π_* is called *optimal* if $\pi_* \geq \pi, \forall \pi$. There exists at least one such policy and these optimal policies achieve the optimal value function: $v_{\pi_*}(s) = v_*(s)$ (resp. $q_{\pi_*}(s,a) = q_*(s,a)$) $\forall s \in \mathcal{S}$. A way to find an optimal policy is to *act greedy* with respect to the optimal state-action value function, which means to always maximize over $q_*(s,a)$:

$$\pi_*(a \mid s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

This shows that, if we know $q_*(s,a)$ we can immediately solve the MDP. There are different solution methods to achieve this, or to approach the function as good as possible through experience. Some methods will be explained on a high level in section 2.2.6. We first want to have a look at some important concepts, that have not yet been introduced, but are essential in reinforcement learning.

2.2.5 Important Concepts

2.2.5.1 Prediction versus Control

Finding an optimal policy can be divided into two sub-processes: the evaluation and the improvement process.
The goal of the *evaluation process* (in literature often called "policy evaluation") is to compute the value function v_π (or q_π) for a fixed arbitrary policy π. There exists different iterative methods, where a sequence of value functions $\{v_k\}$, with $\lim_{k\to\infty} v_k = v_\pi$, is generated, iteratively updating the estimates $v_k(s)$ for the unknown values $v_\pi(s), \forall s \in \mathcal{S}$. And therefore one is changing the value function to be more like the true unknown value function, for the given policy π. There exists different update rules, depending on the method.
The other process, the *improvement process*, takes the value function as given and tries to improve it, assuming that the given value function is its true value function.

To solve an MDP, these two processes need to act in a cyclic manner, as the figure 2.2 indicates: The value function is repeatedly changed to match the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. Together, these two iterative adaptions, they cause both policy and value function to approach optimality [3]. Because when this iterative process adds no more change to the policy or the value function, so if the cyclic process stabilizes, then the obtained policy and corresponding value function must be optimal. This general idea of two interacting processes, the policy evaluation and policy improvement process, is also known as the *generalized policy iteration (GPI).*

Often, the algorithms solving an MDP are initially focused on the evaluation step. In this context, one is looking at the so-called *prediction problem* of an MDP. If one adds also the improvement step to the algorithm, then one is looking at the so-called *control problem*, which therefore contains the evaluation and the improvement step. There exists different prediction and control methods, see for example Szepesvári [25] chapter 2 for some prediction methods and chapter 3 for control methods.

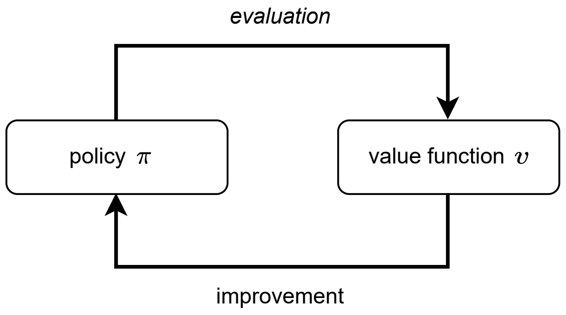


Figure 2.2: Policy iteration process, covering the prediction and control problem: The prediction problem addresses the evaluation process, while the control problem addresses both, the evaluation and improvement process.

2.2.5.2 Optimality and Approximation

Finding the optimal policy or creating the optimal policy on the fly with the concept from eq. (2.10) can be a simple task, if the sets of all states and actions, $\mathcal{S}$ and $\mathcal{A}$, are not too big. Essentially, it results in table look-ups to find a maximum value in a table of size $|\mathcal{S}| \times |\mathcal{A}|$. But when the sets $\mathcal{S}$ and $\mathcal{A}$ are big, then this table-based methods are not performing well enough and one has to find methods to best approximate the optimal solution, simply because there is not enough time (to collect enough experience) and computer power to search for the global optimal policy. To cite Sutton and Barto [3] (p. 85), "For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep." (Single sweep: performing one update for each state).

There exists multiple approximation methods, with the same overall idea: dimensionality reduction and generalization. This can be obtained by function approximation, where the value function gets parametrized: $v(s, \mathbf{w}) = v_\pi(s)$. The parameter $\mathbf{w}$ is called the *weight vector*. The goal of these algorithms, in the prediction step, is then to find the optimal weight vector $\mathbf{w}$ to best approximate the true, unknown value function $v_\pi(s)$ for a given policy π. Dimensionality reduction is obtained, due to the fact that the weight vector $\mathbf{w}$ should have less dimensions than the state space $\mathcal{S}$. The sub-problem of finding such a weight vector $\mathbf{w}$ can be looked at as a classical supervised learning problem, and one can include well known methods from supervised learning for the function approximation into the reinforcement learning algorithm, as for example linear approximation or artificial neural networks. We will have a closer look at this concept in section 3.3.3.3

2.2.5.3 Exploration versus Exploitation

A well-known dilemma in reinforcement learning is called the *exploration versus exploitation dilemma*. It can be simply explained with an example: Imagine that you are new to a town an there are six different restaurants. You want to find out which is the best, so you start by picking a restaurant at random. You go have lunch there and you like the food very much. Now, you face the dilemma: Should you just eat there every day and *exploit* the fact that you know you like this food, or should you try also the other restaurants and *explore* the different options, maybe finding an even better restaurant, but taking the risk of eating less tasty food over some time?

The same dilemma an agent is facing when he needs to decide which action to take, respectively which policy to follow. As long as he has not explored everything, he cannot be sure that the best policy he knows is also the optimal one. Therefore, as long as the agent has not yet visited all possible states and has chosen all possible actions, he should continue to explore, but only so much, that he is still exploiting the already known states, which bring him a high reward.

There exists different exploration methods which can be used in the context of any reinforcement learning algorithm. For example the *ϵ-greedy method*, which will be explained in the following. There exists many more sophisticated methods, such as softmax or recency-based methods, see for example Olivier Sigaud [24], p. 43.

The ϵ-greedy procedure
By a *greedy procedure* one understands the following: At each point in time, we choose the best possible next action. That is, the action that gives us the most value. So, the new so-called "greedy policy" (noted π') is as follows:

$$\pi'(s) := \arg\max_a q_\pi(s,a) \tag{2.11}$$

The greedy policy selects the action that looks best according to the current state-action value function q_π. Acting-greedy means to always choose the best possible action in the short term. But in order to guarantee a continuous exploration, we have to introduce a parameter $\epsilon \in [0,1]$, and thus arrive at the *ϵ-greedy policy*, which selects the short term best possible action a_t with a probability of $1-\epsilon$ and a random one with probability ϵ:

$$a_t \leftarrow \begin{cases} \arg\max_a q_\pi(s,a) & \text{with probability } 1-\varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases} \tag{2.12}$$

2.2.5.4 On-policy versus Off-policy Learning

There are generally two different approaches for learning methods: the on-policy and the off-policy approach. In the *on-policy approach*, the policy, that is currently being followed, is directly and continuously improved. In the *off-policy approach*, two different policies are involved. One distinguishes between the policy that is currently being followed and the policy that is being optimized in the background. This means that an optimal policy (also called "target policy") is learned, while another policy (also called "behavior policy") is followed. The advantage of off-policy methods is that they allow for more exploration without compromising the policy to be optimized. With the on-policy approach, the exploration must be built into the policy itself (for example with an ϵ-greedy policy), and in general they are more straightforward.

2.2.5.5 Model-based versus Model-free Methods

In reinforcement learning, we distinguish between model-based and model-free methods. With the help of a model, one tries to simulate the environment, in which the agent operates. It is used to predict how the environment will react to the chosen actions of the agent. Model-based methods try to reproduce the exact dynamics of the environment. That is, one tries to estimate the parameters of the unknown transition function $p(a, s)$ from eq. (2.3). Taking the model as input, the *planning step* follows, seeking the optimal policy. Model-free methods, on the other hand, do not build a model. They directly update the value functions locally, always after receiving feedback from the environment. Therefore, they do not require the transition probability function eq. (2.3) or reward function eq. (2.5) (or an approximation of them) as input, because they are not interested in modelling them. This is totally in contrast to model-based methods, where the global strategy is adapted and not only single values of the value function are adapted locally. Since model-free methods directly adjust the value function, they are also called "direct methods", while model-based methods are called "indirect methods". The flow of model-based and model-free methods is also shown in figure 2.3. The advantage of building a model is that the model can also be used for simulations. If learning is done without a model, then we can only learn from actually realized experience, since we cannot imitate the environment by only using the value function. The advantage of model-free learning is that the methods are much more direct and simpler. In practice, both methods are used [3][24].

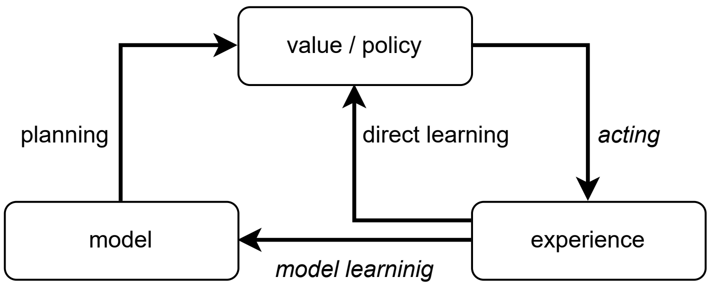


Figure 2.3: Flow of model-based and model-free learning methods (source: [3]).

2.2.6 Solution Methods

This section is intended to provide an overview of the core ideas of specific reinforcement learning algorithms. The goal is mainly to give a classification of the core principles, and less the to describe the methods or associated algorithms in detail. For precise descriptions of methods and specific algorithms, [3] and [25] are recommended.

Since one can translate almost any reinforcement problem into a setting of finite MDPs, we assume a given MDP. If one must give a classification for the main approaches to solving MDPs, then the following three methods would probably often dominate: Dynamic Programming (DP), Monte Carlo (MC) methods and Temporal Difference (TD) learning.

Dynamic Programming
Dynamic Programming is a general method for solving complex problems, not only known for in reinforcement learning setting. In the MDP setting, the model must be fully known: we need to know the elements of the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p, r, \gamma \rangle$. The most popular algorithms in Dynamic Programming are "Policy Iteration" and "Value Iteration", which are well described in [25], chapter 1.4. They essentially follow the evaluation and improvement process described in section 2.2.5.1 and thus, since the full dynamics are known, can converge to an optimal solution. As Dynamic Programming requires a perfect and fully known MDP model, which is often not the case, it makes it often not applicable for practical applications.

The next two methods, Monte Carlo and Temporal Difference learning, can be seen as attempts to mimic the setting of Dynamic Programming as good as possible, only with less computational effort and without the assumption that we have a perfect MDP model.

Monte Carlo Methods
Monte Carlo methods do not need the knowledge of the MDP model, they only need *experience*: samples of states, actions and rewards obtained by interacting with the environment. For this purpose, the interactions are divided into individual episodes. Then all selected actions, resulting states and received rewards during an episode are being stored. Having waited until the end of an episode, one can calculate the *sample returns* G_t and use them to update the state-action values. Because the updates are taken towards the observed sample returns, Monte Carlo methods can only learn in an episode-by-episode sense and not in an "online" (step-by-step) sense.

Temporal Difference Learning
Temporal Difference learning combines the Monte Carlo and Dynamic Programming ideas. As with Monte Carlo, it is not assumed that the dynamics of the MDP model are fully known, but instead one learns as well only through experience. The difference to Monte Carlo is that one does not have to wait until the end of an episode to update the value function. Instead, one waits only until the next time step and directly uses the newly gained information (observed reward r_t and new state s_{t+1}) to calculate a new estimate for the value of a state. For example: $v_{new}(s_t) \leftarrow r_t + v_{old}(s_{t+1})$. In section 3.3.3, we will learn more about the Temporal Difference concepts. Temporal Difference methods have in general, compared to Monte Carlo methods, lower variance but higher bias [3].

Each of these three methods contains a collection of different algorithms, all of which apply the respective principles of the methods described earlier, but each in its unique way, for example updating values on the basis of sample averages; updating values only after a certain amount of time steps (batch learning); updating values after each time step; etc. Of course, each method brings its own advantages and disadvantages. However, the methods can also be combined to take full advantage of their respective strengths and weaknesses. Such blend-ins can, for example, be made with *n-step algorithms*, which form a bridge between Monte Carlo and Temporal Difference methods ("n-step look-ahead", instead of "one-step look-ahead" as in Temporal Difference learning, or waiting until end of an episode as in Monte Carlo methods. For more details: [3], chapter 7). Or with the introduction of a parameter λ, which unifies the two methods even further. This results in the group of TD(λ) algorithms, which we will get to know in more detail in section 3.3.

Chapter 3

App Requirement Engineering

In the context of this thesis, it is intended to model the basis of an application, which shall be applicable in pain therapy. In this chapter, we will perform the requirement engineering of the application, before we address the implementation in chapter 4. First, the purpose and requirements of the application are described, followed by the mapping into a mathematical model. We will also describe the requirements for the learning algorithm, to be built into the application, and finally discuss the selected learning algorithm in detail.

3.1 Purpose and Requirements

The goal is to develop a self-learning application that can be used in a variety of settings to support mental and physical well-being, including and foremost in pain management. The application should enable the user to be transferred into a virtual world. In this world, into which one is transferred via the device's screen and speakers, different stimuli are sent to the user. The selection of stimuli is intended to cover the different modalities we learned about in section 2.1.2.1: auditory stimuli (such as sounds, tones, and music), visual stimuli (such as geometric shapes in different colors), and kinesthetic stimuli (such as provoking eye movements by projecting objects in motion). The purpose of using the different modalities is to allow the application to appeal to as many different users, having different learning types, as possible. The projection of the different stimuli is intended to generate different neural stimulations in the brain and thus influence the user's perception of pain and other emotional sensations. This is accomplished through the curative effects of the stimuli and with built-in techniques such as visualization, coupling, and objectification, which we learned about in sections 2.1.2.1 and 2.1.2.2. The goal of the application is thus to "train the mind to think differently", in some sense. We use the fact that we can trigger the creation of new synapses in our brain, which influences our behavior and sensations (for example sensation of pain).

The exact stimuli sent to a particular user should be decided by an algorithm and be unique to each user, as each user has individual needs and processing capabilities. For this purpose, the application contains a machine learning algorithm, which allows each user to train its application independently and thus receive an individual and personalized application. Consequently, the stimuli are adapted to each user according to their individual preferences. For this purpose, the user should have a feedback option to be able to report to the algorithm which stimuli he perceives as positive or negative.

The main target audience are people suffering from pain. However, the goal is not only to impact the sensation of pain, but also to improve the overall state of mind, by influencing the accompanying symptoms of pain patients, such as the reduction of anxiety or stress, strengthening of concentration, and aiming for an overall well-balanced state of mind. The application should enable to shift the focus away from any unpleasant feeling or sensation. Thus, the application obtains an even larger target audience, since it can also serve when no pain is present. There should be no limits to the use of the application and it should offer every user the possibility to dive into a virtual world, to train its individual algorithm and to experience the desired effect of the different stimuli (as described in section 2.1). The area of utilisation of the application differs significantly from conventional accompanied therapy sessions (as hypnosis or neuro-linguistic programming), since on the one hand it is therapist-independent and can therefore be used at any given time, and on the other hand it offers a possibility for people for whom imaginations are only possible with great difficulty or not at all, because the users do not have to picture something imaginarily, but can let themselves be stimulated by the application. For the time being, the application will be created on a 2D screen, but it could be continuously expanded, such as to a 3D or even to a VR (Virtual Reality) application, which would allow an even stronger immersion into a virtual place.

3.1.1 Necessity and Potential of Reinforcement Learning

In section 2.1, we have learned the basic knowledge about psychological pain management and different stimuli, as well as techniques which are already used in pain therapy today. The application will benefit from such techniques by incorporating such elements (like colored objects or music). But what distinguishes the application from conventional forms of therapy? The big advantage that makes the application remarkable, is the combination of neural stimulation methods and reinforcement learning. The therapy is not guided by a therapist, but the therapy happens solely with the user of the application and the underlying reinforcement learning algorithm. The user interacts with the application and thus the algorithm learns which stimulations are profitable for the user. It is quite different with traditional forms of therapy, where a middleman is still involved: a music therapist puts together music sequences for the patient, the hypnotherapist guides the patient through a hypnosis, etc. Of course, the therapists also try to respond to the patient in each case and offer an individualized session to the best of their knowledge. But simply because of the limited knowledge of the expert and the limited possibilities of action, such forms of therapy are always dependent on the therapist. We want to overcome this limitation. The ideal form of therapy for a patient should be found independently of the limited knowledge or resources of a therapist. This is where reinforcement learning offers its potential: The user interacts with the application, trains its own algorithm, and thus receives an individualized application. Hardly any other form of therapy can offer (or even guarantee) such individualization, which is why reinforcement learning offers great potential in our setting.

3.2 Mathematical Model

To better understand the big picture of the application, we will first look at the general flow and the various components involved.

Involved components

- *User*: the user of the application. The person who looks at screens and sends feedback via clicking buttons.
- *Application*:
  - *Front-end*: refers to the user interface (UI); includes scripts that display the objects on the screen, with which the user interacts.
  - *Back-end*: is the server side; includes machine learning scripts (specifically: the reinforcement learning algorithm) which compute the screen configurations and send them to the front-end.

Flow

The user is looking at the screen on which the application is running, for example on his smartphone. This screen contains two feedback buttons (thumbs up, thumbs down), which the user can press at any time. Thus, he can send either positive or negative feedback for a particular screen configuration. The feedback is received by the front-end and forwarded to the back-end. In the back-end, the feedback is getting processed and a new screen configuration is calculated. The back-end sends the new screen configuration to the front-end, which projects the new elements on the screen. Thus, the user sees the new screen configuration and can give feedback yet again. The goal of the application is to find the best screen configurations or sequence of screen configurations for the user. The figure 3.1 illustrates the flow as described.

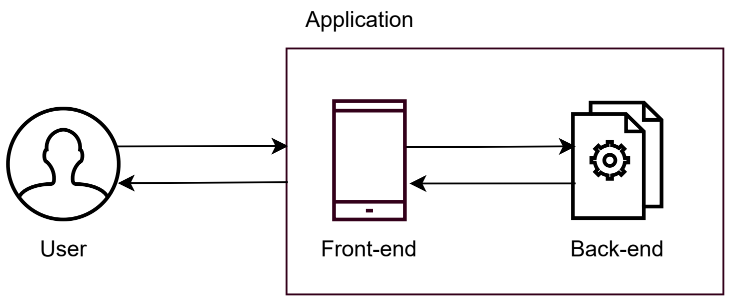


Figure 3.1: Application flow and involved components.

To reformulate the entire task into a mathematical model, we use the framework of the Markov decision problem (MDP), which we introduced in section 2.2.

3.2.1 Mapping into a Markov Decision Problem

It should first be pointed out that the following modelling, as well as the choice of the final algorithm (see section 3.3.2), are based on theoretical considerations. In practice, such an initial modeling is used to start with, and can (or should) then be adapted in an iterative process by trial-and-error. Adaptions could take place in the definition of the states, actions and reward representations or other learning parameters. The reason for adjustments may be that the agent does not reach the intended target or also to improve the performance. As Sutton writes in his book ([3], p. 50), "Of course, the particular states and actions vary greatly from task to task, and how they are represented can strongly affect performance. In reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science". Therefore, one should invest a great amount of time in defining the model, but one should always remain flexible and allow to adjust one's decisions.

Agent and Environment

We are in a reinforcement learning setting, so we first need an agent and an environment. The agent is not the user of the application, as one might first assume, but the learner and decision maker of the task. In our setting we have to decide which screen configuration should be shown to the user. Thus, our agent is the *reinforcement learning algorithm* (RL-algorithm), since it makes the decisions on which screens to show. Everything outside of the algorithm, i.e. the user and all components of the application - excluding the reinforcement learning algorithm - , we consider to belong to the environment. The agent's goal is to find the best screen configuration, or sequence of screen configurations, among all possible options, as quickly as possible.

Reward Function

According to eq. (2.5) the reward function must be defined from $\mathcal{S} \times \mathcal{A}$ to $\mathbb{R}$. It indicates how the environment reacts in a certain state to a certain decision by the agent. In our case, the reward function is unknown: We do not know in advance how the user will react to the different screen configurations. But what we can do, is converting the feedback we receive from the user into a reward signal:

Feedback options for the user:

- positive feedback: button with thumbs-up item is pressed
- negative feedback: button with thumbs-down item is pressed
- no feedback: no button is pressed

Mapping user feedback into reward-signals for the RL-algorithm:

- positive feedback $\rightarrow$ +10 reward
- negative feedback: $\rightarrow$ -10 reward
- no feedback: $\rightarrow$ 0 reward

Positive feedback from the user should tell the algorithm that the screen configuration was satisfactory, therefore a positive reward signal (+10) was chosen. Vice versa for the negative feedback (-10). Less obvious is the definition of the reward when no feedback was received for a particular screen configuration. Do we want to punish this behavior with a negative reward (e.g. reward = -1), or handle it neutrally (reward = 0). We strive for a positive state of mind of the user and "no feedback" does not mean that the screen configuration was unpleasant for the user, but rather "neutral". Therefore, the choice was made for a 0-reward.

Discounting Factor

The discounting factor $\gamma \in [0, 1]$ influences how much we want to weight later reward (see eq. (2.7)). In our case, we strive for early rewards (the user should be able to benefit as soon as possible), but later rewards are also considered important (the user should be able to have long-term benefits). Therefore, we decide to use a discounting factor of $\gamma = 0.9$, which weights earlier rewards only slightly stronger.

Set of States and Actions

Next, we need to define a set of states and actions.

The choice for the set of actions $\mathcal{A}$ is quite intuitive: We define this set as a collection of all different screen configurations we have to offer. So, an action $a \in \mathcal{A}$ is one specific screen configuration. We will represent each action by a vector. The number of tuning parameters, which specify the screen configurations, determines the length of the vector. In section 3.2.2, these tuning parameters will be discussed in more detail.

The definition of our state space is not as trivial and needs careful consideration. We want our states to summarize as compact as possible the past information, and to respect the Markov property as well as possible. Thus, a state at time t, s_t, should contain all information from the past (screen configurations and rewards) that are relevant for the future decision (for the new screen configurations), and represent this information as compactly as possible.
Suppose we have a sequence of screen configurations a_i and rewards r_i and note $O_t := (a_0, r_1, a_1, r_2, ..., a_{t-1}, r_t)$ for the history of observations up to time t. If we want to ensure that the states have the Markov property, then we could choose the entire history of observations as state representation: $s_t = O_t$. However, this would not be efficient, since the history of observations grows with time t and would probably never recur. An alternative would be to determine so-called *features*, which collect aggregate information from the history of observations O_t. But determining such features is a big challenge, as the following example should illustrate. Suppose we have the following (simplified) history $O_2 = (a_0 = \text{red triangle}, r_1 = 0, a_1 = \text{red square}, r_2 = +10)$. What was the decisive feature for the positive reward $r_2 = +10$? Was it the red color (feature: tracking occurrence of color red), was it the shape of the square (feature: tracking occurrences of shape square), or was it the event of two consecutive red elements (feature: tracking sequences of red elements), or was it the appearance of the square after the triangle (feature: tracking occurrence of shape square following triangle)?

We do not know the "rules of the game". Therefore, we do not want to limit the "game" by autonomously defining features, while hoping that we cover the essential ones. Therefore, the decision for our state representation falls on the so-called *k-th order history*, collecting the last k actions and rewards:

$$S_t := (R_t, A_{t-1}, R_{t-1}, ..., A_{t-k}) \quad \text{for some} \quad k \geq 1. \tag{3.1}$$

Thus, while we must autonomously define the parameter k, we leave everything else to the algorithm and therefore do not artificially constrain the learning problem. The parameter k is introduced, so that the sequence does not grow continuously. By imposing this constraint, we can no longer fully guarantee the Markov property. In reality, obtaining a state that fulfills the Markov property completely, is practically impossible. Therefore, it is common for almost any learning problem, that one has to settle for *near-markov* states, nevertheless reality shows that these models can also perform very well [3].

Transition Probability Function

The transition probability function (see eq. (2.3)) defines the probability that one ends up in a certain state s' and receives a certain reward r, if one chooses a specific action a, starting from a specific state s:
$p(s', r|s, a) = P(S_t = s, R_t = r \mid S_{t-1} = s, A_{t-1} = a)$.
It defines the dynamics of the model, so to speak. However, we do not know these dynamics, and therefore we cannot define such a transition probability function. Nevertheless, as we have seen in section 2.2.6, this will not prevent us from finding a suitable algorithm, since there exists methods that do not require the full MDP model as input.

In summary, this results in the following definitions and notations:

Definitions and notations

t	timestep, $\in$ N
$\gamma := 0.9$	discounting factor
$\mathcal{R} := \{+10, -10, 0\}$	set of rewards
r_t	reward at time t, $\in \mathcal{R}$
$\mathcal{A} := \cup_{i=1}^{m} a_i$	set of actions
$a_i := (a_i^1, ..., a_i^n)$	an action
a_i^j	j-th entry of an action a_i
m	$\lvert\mathcal{A}\rvert$, number of different screen configurations
n	number of tuning parameter for one screen
a_t	action at time t, $\in \mathcal{A}$
$\mathcal{S} := \cup_{i=1}^{l} s_i$	set of states
$s_t := (r_t, a_{t-1}, r_{t-1}, ..., a_{t-k})$	state at time t, based on k-th order history, $\in \mathcal{S}$
l	$\lvert\mathcal{S}\rvert$, number of different states
k	parameter k from the k-th order history

Figure 3.2 illustrates the flow of the application, now from the perspective of an MDP.

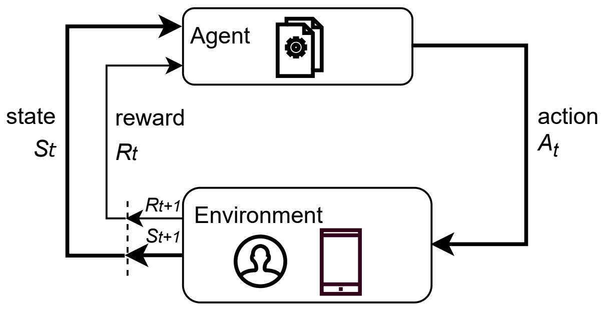


Figure 3.2: Application flow from the perspective of an MDP, whereby the agent is the RL-algorithm and the environment contains the user and the front-end of the application.

Further Concepts: Episodes and Timing of Feedback
Two other important concepts in our setting should be addressed at this point. First, the definition of episodes. As we have seen in section 2.2.4.2, the concept of episodes enables us to break down the interactions of the agent with the environment into subsequences, and can also have an impact on the way of learning, for a given algorithm (for example: in the Monte Carlo setting we update the values at the end of each episode). In our setting, we could essentially decide for two different definitions of episodes. Either we look at our setting as kind of a "game" and call an episode the sequence, which either ends with a "win" (positive user feedback) or a "fail" (negative user feedback). Or we define an episode as the set of all states, actions and rewards collected during one "user-session", whereby a user-session would start when opening the application and end when closing the application. From an implementation perspective, in both cases the data should be stored across episodes. This requires that users should be able to create an account (with user-name and password) and would have to log in every time they start the application, so that the back-end can identify the user and pick-up the user's personally trained algorithm. Because we want to encourage the user to give feedback as much as possible, so preferably rather at each "step of the game" (after presenting a certain screen configuration) than at the "end of a game", the second option was thought to fit better, and has therefore been chosen to start with: an episode starts and ends with the user-session.

The second concept we should address is the timing of the user feedback and how this impacts the flow of the application. The main questions that should be considered are: When do we want to ask for user feedback? How often to we want to ask for user feedback? What happens right after receiving the user feedback? What happens if the user provides no positive or negative feedback on a certain screen configuration? To give answers to these questions, we need to reflect about some requirements that are important to guaranty a pleasant user experience. The user should never see an empty screen and should theoretically be able to enjoy a endless sequence of different screens. The screens should change fluently, without any disturbances as interruptions, abruptly changing images or even empty screens. Having these requirements in mind, we should allow the user to give feedback at any time, and screen configurations should eventually change even if the user does not provide feedback for an extended period of time.

As we have decided for three different user feedback (neutral, positive, negative) a first approach could be to add for each screen-configuration a parameter ”duration of appearance” which defines how long a certain screen should be shown. As soon as this so-called duration of appearance has elapsed, the system checks whether the user has sent positive, negative or neutral (i.e. no) feedback and sends it as reward signal to the algorithm. (Edge case: What if we receive multiple feedback for the same screen? Solution: We take the latest feedback into account, as it seems most representative.)
Besides the parameter ”duration of appearance”, another parameter, which is directly linked to changing screen configurations, is the ”transition type”. It describes the way we transition from one screen (for example displaying a red circle) to another (for example displaying a blue square). From an implementation point of view, this raises several challenges, especially the inclusion of the transition types as parameters, since these depend on the particular screen configurations (old screen and new screen), and since we want to guarantee a fluent application. One idea to gain some time and to guarantee a fluent flow could be to include a *buffer*: We could always keep two (or more) pre-calculated screen configurations ready in the buffer, in order to gain time to process the real-time feedback from the user and to calculate new configurations. The various options and exact details need to be thought through during implementation and depend heavily on the capabilities provided by the implementation tool. Figure 3.3 illustrates the idea of the buffer with two screen configurations as back-up.

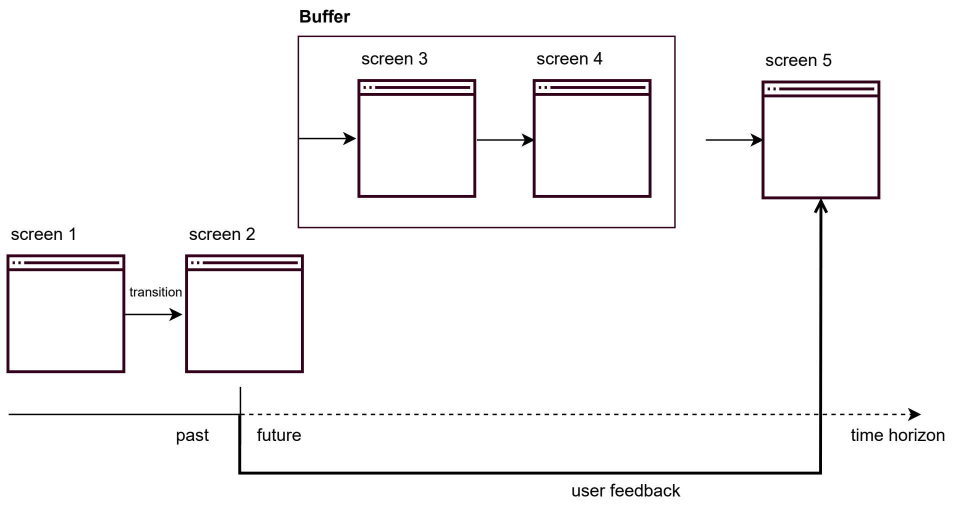


Figure 3.3: Timing of user feedback and concept of a buffer containing two screen configurations as back-up: After the ”duration of appearance” for screen 2 has elapsed, the user feedback is collected (positive, negative, or neutral) and processed to produce the screen configuration 5. Screen 3 and screen 4 are not affected by this user feedback, as they have already been configured with previous feedback and are in the buffer, ready to be presented next.

3.2.2 Tuning Parameters

In the last section we defined the ”actions” as a specific screen configurations. But what is actually meant by this? And how are they structured? To create a certain screen configuration, the front-end needs the values of certain parameters. For example, in order for the front-end to project an element such as a red triangle with radius 1 cm in the center of the screen, it needs the following inputs: {form: circle, color: red, location: center, size: 1cm}. In the following sections, we will present the parameter representation which was chosen for the application. The choice of the parameters is based on their effects on neurological stimuli, as introduced in section 2.1. The main focus will not be the description of the parameters and the justification of their choices, as this is covered by section 2.1, but the problem of how we structure the data, so that the algorithm can work with it. In chapter 4, where we will address the implementation of a first prototype, we will start by implementing only a certain selection of the parameters presented in the following section, allowing to gradually increase the complexity for further prototypes.

3.2.2.1 Parameter Representation

As in the example above with the red circle, an element (or ”feature”) can have several attributes. To maintain a certain structure, the following composition of objects and properties was chosen: feature $\longrightarrow$ attribute $\longrightarrow$ value.
A *feature* can have multiple *attributes*. Each attribute possesses a choice of *values*, but only one specific value is assigned to each attribute. The following example illustrates this structure:

feature 1	shape	**attribute 1.1**	form	**value**	circle
		attribute 1.2	color	**value**	red
		attribute 1.3	size	**value**	1 cm
		attribute 1.4	location	**value**	center
feature 2	sounds	**attribute 2.1**	sound	**value**	rain
		attribute 2.2	volume	**value**	medium

These inputs must then be transformed into a single vector. The representation of a vector was chosen, because it is easy to manipulate, and many classical reinforcement algorithms work with vectors as action representations. In the example given above, the vector representing the action a_t would be the following:

$$a_t = [\text{circle, red, 1, center, rain, medium}].$$

The length of the action-vector n is thus determined by the sum of the attributes over all features, our so-called *tuning parameters*. In the above example, we have $n = 6$. With a rough estimate of the desired features and their attributes in a final version of the application, a length of $n = 100$ (with approximately 10 features and around 10 attributes each) is expected. The following section describes a selection of these features and attributes.

3.2.2.2 Features and Attributes

Through the learned effects of the different stimuli in section 2.1, the following main features will certainly be included in the application:

1) *background picture/video*
2) *background music*
3) *shapes*
4) *sounds*

The shapes will be presented in different colors and also in motion, to provoke eye movements.

In order to use the suggestion and coupling effects (see section 2.1.2.2) in an active manner, the following additional features could be introduced:

5) *User inputs*:
   At the start of the application, one could ask the user certain questions, with the ulterior motive that he will be engaged with a certain topic and then assign attributes specifically to that topic. For example, as follows: Question 1: "What topic is currently on your mind?", Question 2: "What color would you assign to this topic?", Question 3: "What geometric shape would you assign to this topic?". Thus, the user dedicates each session to a topic (for example: pain) and couples a shape and color to it. This input can then be used in a variety of ways inside the application. For example, the algorithm could be manipulated at startup, so that the selected shape and color are more likely to appear at the beginning. Thus, the user is immediately confronted with his topic, respectively with the assigned object, and by passing into other screens, the object can be changed into something more pleasant and the coupled feelings about the topic can be influenced.

6) *Words*:
   One could also project *words* on the screens (also in different colors and in motion), with the ulterior motive that often people associate certain words with similar emotions. A selection of possible words, divided into different categories, could be as follows:

   a) category relaxation: "calm", "deep breath ", "relax ", "balance".
   b) category pain relief: "cool", "numb", "lightness".
   c) category positive suggestions: "happy", "joy", "freedom", "self-love".

   Reading these words can already trigger emotions for the user, as one automatically associates memories and/or sentiments with such words. For some users, reading could be unpleasant or too exhausting. The algorithm would be able to learn about this preference, and then over time would not project written words any more.

Now we have enumerated six feature categories: background picture/video, background music, shapes, sounds, user inputs and words. Each feature would have different attributes. The attributes corresponding to the features *shape* and *sounds* are given as examples in table 3.1 and table 3.2. The second column contains either a set of possible values or a description of the value unit.

Attribute	**Values**
Form	{circle, rectangle, square, triangle, rhombus, hexagon}
Location of appearance	some integer(s), representing screen coordinates
Movement	trajectory, defined by a function with screen-axes x,y as variables
Speed	some integer, representing speed
Size	some integer, representing size in pixels
Color	{red, blue, green, yellow, ...}
Color intensity (alpha level)	$\alpha \in [0, 1]$
Duration of appearance	$s \in$ [0,10], representing seconds

Table 3.1: Attributes and values for feature *shape*

Attribute	**Values**
Sound	{gong beat, high pitch, rain drop, wind, beach waves, ... }
Volume	{high, medium, low}
Repetition	$r \in$ N
Location	{left ear, right ear, center}
Duration of appearance	$s \in$ [0,10], representing seconds

Table 3.2: Attributes and values for feature *sounds*

The application could show several of the same features (with different attributes) on the same screen. For example, a shape=circle and a shape=square can be displayed on a single screen. Some features can thus be configured multiple times for a single screen configuration. Assume that for the screen configuration we allow two features from the category "shape" and one from the category "sounds". Since 8 attributes are assigned for a feature "shape" and 5 attributes for the feature "sounds" (according to table 3.1 and 3.2), we would have a vector of length $n = 21(= 8 + 8 + 5)$, defining a specific screen configuration. The RL-algorithm would have to find the $n = 21$ optimal entries (or sequences of entries) for a certain user.
Let us have a look at the size of our state space, respectively of the number of possible different screen configurations. Back at our example with $n = 21$ tuning parameters, assume we have a choice of 5 possible values to choose from for each parameter. We would therefore have a total number of different screen configurations of $m = 5^{21}$, which is already a great number of possible configurations. This is the number of possible "actions" that the RL-algorithm can select from, at each time step t. This small example already shows how large our action and state space will become, due to the exponential growth in the number of tuning parameters. In section 3.3 we will see the impact this will have concerning the choice of a suitable learning algorithm.

3.2.2.3 Parameter Encodings [26][27]

An important topic, generally in machine learning and also considering our application, is the encoding of categorical features. Categorical features are features whose values are not represented with numbers. The goal of encodings, in the machine learning context, is to transform categorical features into a computer-comprehensible language in the most efficient and compact way possible, so that algorithms can understand and process the values. Some encoding options are described in the next paragraphs. But first, it is useful to introduce a standard classification of categorical features:

1. **Binary features**: either/or grouping.
   Example: {True, False}

2. **Nominal features**: grouping without meaningful order.
   Example: {circle, square, triangle}

3. **Ordinal features**: grouping with meaningful order.
   Example: {small, medium, large}

Depending on which category a feature belongs to, a suitable encoder should be selected. The binary features are not a big challenge, because there, a simple mapping to the two values 0 and 1 can be chosen. More interesting are the encoders for nominal and ordinal features. In the following, three encoder types and their advantages and disadvantages will be discussed.

Label Encoding (also known as: Ordinal Encoding)
In label encoding, each categorical variable gets a scalar variable assigned to it. For example: the categorical values {circle, square, triangle} would be mapped to {1,2,3}, as shown in table 3.3.

categorical value	label encoded value
circle	1
square	2
triangle	3

Table 3.3: Short example of a Label Encoding

Label encoding is simple and therefore often used. However, one should always keep in mind that label encoding artificially introduces a kind of ordering into the categorical features, which makes no sense for nominal features. If one uses machine learning algorithms that exploit such orderings, such as a gradient descent method, the usage of such label encoders can lead to incorrect results, since in these cases the ordering of the assigned scalars has an influence on the weighting of the features (e.g. higher numbers contribute more to the gradient than lower ones). This is a big disadvantage of label encoding. Nevertheless, label encoding can be a starting point and then be combined with other encodings: Since some encodings need already an integer as input (as for example some libraries do for one-hot encoding), one could first apply a label encoding to the categorical features to map the categories into integers and then apply a suitable method to the obtained values in a second step. If we work with ordinal features, like {small, medium, large}, then a mapping to {1,2,3} is unproblematic, since here such an ordering makes sense (medium should be placed between small and large), and is even desired.

One-Hot Encoding (also known as: Dummy Encoding)
In one-hot encoding, one column gets created for each different categorical value. Each value is given a 1 in a column if the value is included in that column, and a 0 if it is not. The best way to understand it is by looking directly at the example in table 3.4, where each encoded value is represented by four different numbers, one for each category {circle, square, triangle, rectangle}.

categorical value	one-hot encoded value
circle	1000
square	0100
triangle	0010
rectangle	0001

Table 3.4: Short example of a One-Hot Encoding

One-hot encoding is a classical approach for nominal features, since the encoded values have no correlations. All values are equally different (respectively equidistant in the euclidean space). A disadvantage of one-hot encoding in practice is that the cardinality of the variables can be high. In such cases, one-hot encoding is less popular, since processing and storing the data can become a problem, due to the resulting large vectors. However, there exists already efficient libraries that can work around such storage issues by storing the one-hot encoded values in some compressed forms.

Binary Encoding
In binary encoding, the categorical values are first converted into an integer (e.g. with label encoding). In a second step, the integers (in base 10) are converted into their binary representation (base 2). In a third step, the digits are split by the binary strings into separate columns. Binary encoding creates fewer columns than one-hot encoding, thus requires less memory and can be applied well to high-dimensional ordinal features. An example of a binary encoding is given in table 3.5.

categorical value	step 1:	step 2:	step 3:
	label encoded value	binary string	binary encoded value
circle	1	1	001
square	2	10	010
triangle	3	11	011
rectangle	4	100	100

Table 3.5: Short example of a Binary Encoding

Besides the above three encoders (label enconding, one-hot encoding, and binary encoding), there are of course many more. For commonly used features there sometimes exists dedicated encoders, as for example the RGB-representation for color encodings (R=red, G=green, B=blue $\longrightarrow$ (100) represents color red). It must always be well considered which encoding is chosen for a certain feature in order to encode as simple as possible, but also as efficient as possible. In practice, it is not uncommon to test different encoders until an optimized solution is found.

Discretization
Another type of mapping, which does not concern categorical features but shall also be mentioned at this point, is the *discretization* of continuous features. Continuous features are already in numerical format, but in this context the mapping has a different purpose. When discretizing, the continuous features, such as for example the " duration of appearance" feature, which could take values in the time interval of [0.001 sec, 10 sec], are divided into so-called *value buckets* (also known as "bins"). Instead of all possible values between 0.001 and 10 seconds, one could for example only allow the values from the bucket {0.5 sec, 1 sec, 1.5 sec, 2 sec, ..., 10 sec}. The more we limit the number of possibilities, i.e. the size of the buckets, the less accurate the algorithm can learn the optimal values. But the larger one allows the bucket size, the slower the learning process will be, because the algorithm has to test many more possibilities before it even finds a trend (e.g. trend to subliminal appearances), and then keep looking for the more fine-tuned optimum. There is a trade-off here, between speed and accuracy.

3.3 Learning Algorithm

3.3.1 Requirements

In order to find a suitable algorithm for a particular learning problem, the exact requirements for the algorithm and the general setting, in which one finds oneself, must be clearly defined. Here follows a list of the most important requirements for the algorithm for our application, which thus determines the setting of our learning problem. Here, we use terms and concepts introduced in section 2.2.

1. **Model-Free**
   Since we know neither the transition probability function (eq. (2.3)), nor the reward function (eq. (2.5)), and are also not interested in modelling them, we are in a so-called model-free setting. Thus, an algorithm is needed which does not require these functions as input.

2. **Online Learning**
   The application learns through direct interactions with the user. Since we cannot model the user and his behavior (because we do not know the user's decision factors and environmental influences), a learning effect only takes place when the user works directly with the application. We are thus in an online learning setting: new data for the algorithm can only be generated through direct interaction. Since we want to offer to the user a smooth experience, this means that new screen configurations have to be computed at a reasonable speed (real-time computation).

3. **On-Policy Learning**
   It is intended that the user can benefit from his feedback as quickly as possible. At all times, the user should be able to experience the best possible experience. The algorithm should therefore adapt its screen configurations directly, rather than learning a target policy in the background, while showing the user a different behavior policy. Therefore, the policy, that determines directly what is currently displayed to the user, is always the same as the one being adapted from the received feedback. So, we are in an on-policy learning setting.

4. **Large Action and State Space**
   As described in section 3.2.1, our action space will reflect all possible screen configurations. For a screen configuration in a final version of the application, there could be around $n = 100$ different tuning parameters (different shapes in different forms, sizes and colors, with different movements, different sounds, etc.). If we assume that for each of these 100 parameters there is a possible choice of 10 different values (10 different colors, 10 different sizes, etc.), then we have 10^{100} different combinations of screen configurations, and thus an *action space* of size $m = 10^{100}$. The *state space* will be even larger, since we are using the k-th order history, which is a sequence of the past k actions and rewards. This action and state space magnitude is certainly worth considering and might needs a redesign, since it is enormous. It should only be mentioned at this point, to get a first impression of the order of magnitude we are dealing with and the effect of our exponential growth. In a final version of the application, the possible screen configurations (i.e. actions) should preferably be restricted, so that, for example, only 5 parameters can be changed from one screen to the next, in order to have a less bigger action space and as well to achieve a more pleasant flow. In this case, we would have a set of 5^{10} actions (which would vary depending on the current state). But even this order of magnitude is significant. Therefore, an algorithm must be chosen that is able to handle large action and state spaces.

5. **Possible Non-Existence of Global Optimum: Continuous Exploration**
   The user should receive a customized application based on the feedback he provides. The algorithm should therefore learn from the user feedback and find the best possible configurations for the respective user. So, it is important that the algorithm moves in the direction most profitable for the user. Convergence is a goal we want to achieve. However, we are not in a setting, like in most games, with strict rules and winning strategies, where the best or optimal strategy can be determined (for example as in Black Jack). Our user might react differently to the same stimuli (screen configurations) depending on internal and external factors, such as time of day or his general state of mind (sad, happy, stressed). That is where the importance of exploration and exploitation comes in. We do not want our algorithm to optimize strictly in one direction only, but to always remain flexible, since users do not necessarily always react the same way, respectively do not always perceive the very same stimuli as pleasant or optimal. Thus, the algorithm should always be exploring (using the knowledge from the feedback already received), but also constantly exploring (trying out unknown configurations, or even previous unfavourable configurations, which may have become favourable). A direct convergence to an optimum is therefore less important in our setting, since we also do not know whether there is such a global optimum for the users. Of course, the algorithm should adapt to the respective user, but rather in the short-term sense and most importantly avoid getting stuck in a local optimum.

6. **Sequential Knowledge**
   If the algorithm receives positive or negative feedback from the user, then this should not only have an effect on the last screen configuration, that was shown, but also on the previous ones. For example, if we show a sequence of images and the user gives positive feedback on the 5th image, then this could be because he perceived the entire sequence (or perhaps only the last three images) as pleasant, and not necessarily specifically only the most recent image. In addition, there is also some possible latency between receiving the stimulus, triggering a neural response, until effectively pushing the feedback button. Due to these factors, and not knowing which images contributed how much to the feedback, the algorithm must be capable of considering sequential knowledge.

Further add-ons would be:

1. **Dynamic Adaptability for Future Extension**
   Ideally, the application should be extensible without losing the learning effect each time the application is updated (for example, when introducing a new parameter). So, it should be considered how to keep the already learned values, while adding new ones.

2. **Defect Handling and Smoothness Guarantee**
   It should also be considered how to maintain a back-up sequence of screen configurations that can be shown alternatively in case the algorithm happens to block. This will ensure a smoother user experience. For example, it could be implemented that, if a certain time-out expires and the front-end has not yet received a new screen configuration from the back-end (algorithm), then it randomly displays a new screen configuration (or sequence of screen configurations).

Another point which enhances smoothness would be the idea of a buffer, mentioned in section 3.2.1, where the algorithm is always some steps ahead of the front-end. Alternatively, one could also define that the policy is not directly adapted after each feedback received, but after a certain number of screen configurations shown (keyword: batch-learning). The different alternatives should be tested in order to evaluate which is optimal.

3.3.2 Decision

Due to the above mentioned criteria, the following algorithm was chosen:

SARSA(λ) with function approximation.

It fulfills all of the above listed six main criteria: SARSA(λ) is a "model-free", "online-learning", and "on-policy learning" algorithm that can handle "sequential knowledge" very well, and with which we can easily guarantee "continuous exploration". By adding the notion of *function approximation* to the SARSA(λ) algorithm, the "large action and state space" problem is being addressed as well as possible. The further add-ons mentioned above need to be addressed in a second implementation phase and do not directly impact the algorithm choice. The algorithm 3 shows the pseudo-code of the algorithm "SARSA(λ) with a linear function approximation". In the following section, we will approach this final version of the algorithm, step by step, highlighting its key features and advantages.

3.3.3 Algorithm explained

In this chapter, the final algorithm will be introduced in a step-wise approach. First the concept of SARSA is explained, then the introduction of function approximations follows and lastly the extension to SARSA(λ) with function approximation and the introduction of the concept of eligbility traces are introduced. In section 3.3.3.5 the topic of convergence is addressed. At the end, critical and further points are mentioned. The main sources of this chapter are Sutton and Barto [3] and Seijen and Sutton [28]. The pseudo-codes, which will follow, are as well inspired by [3], but have always been adapted to fit into our setting.

3.3.3.1 SARSA: short introduction

As explained in section 2.2.5.1, one has to distinguish between prediction and control. For the prediction problem, SARSA uses the concept of Temporal Difference learning (TD), introduced in section 2.2.6. In the prediction problem, the goal is to find the value function for a given policy, or to iteratively approximate it as best as possible. Suppose we have an episode consisting of a sequence of states, actions and rewards, generated by a fixed policy:

$$\{S_0, A_0, R_1, ..., S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, ...\}.$$

In the context of SARSA, one always looks at state-action pairs and is interested in the transition from state-action pairs to state-actions pairs and wants to iteratively approximate the state-action value function $q(s, a)$ (from eq. (2.9)).

As we are in the context of TD-learning, one takes a "one-step look-ahead" and uses the newly gained information (observation of the new reward R_{t+1} and the new state S_{t+1}) to form a better estimate of the current value: $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$ is the new estimate for $q(S_t, A_t)$. And by introducing a step-size parameter $\alpha \in (0, 1]$ one can steer to what extend the update from the old estimate towards the direction of the new estimate takes place. Therefore, the update rule in the SARSA algorithm takes the following form:

$$q\left(S_t, A_t\right) \leftarrow q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma q\left(S_{t+1}, A_{t+1}\right) - q\left(S_t, A_t\right)\right], \qquad (3.2)$$

where $q(S, A)$ is the state-action value function (eq. (2.9)) , $\alpha \in [0, 1]$ the step-size parameter, and γ the discounting factor. This update happens after each time step t during an episode.
Since the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ is used in the update rule 3.2, the algorithm was given the name "SARSA" (for State, Action, Reward, State, Action). In the following section, we will see a complete pseudo-code for a simple SARSA algorithm, which will contain the update rule from eq. (3.2).

3.3.3.2 SARSA algorithm

The algorithm 1 shows the pseudo-code of the simple SARSA control algorithm, for estimating q, which approximates the optimal state-action value function q_*.

Algorithm 1 SARSA for estimating $q \approx q_*$

```
1: Algorithm parameters: step-size $\alpha \in (0, 1]$, small $\varepsilon > 0$, discount rate $\gamma \in [0, 1]$
2: Initialize $q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily
3: Loop for each episode:
    4: Initialize $S$
    5: Choose $A$ from $S$ using $\varepsilon$-greedy policy derived from $q$
    6: Loop for each step of episode:
        7: Take action $A$, observe $R, S'$
        8: Choose $A'$ from $S'$ using $\varepsilon$-greedy policy derived from $q$
        9: $q(S, A) \leftarrow q(S, A) + \alpha\left[R + \gamma q\left(S', A'\right) - q(S, A)\right]$
        10: $S \leftarrow S'; A \leftarrow A'$
    11: until $S$ is terminal
```

In line 9 of algorithm 1, we find the above explained update rule 3.2. Around this update rule, lines have been added to shift from the pure prediction setting (approximation of the value function: accurately predict the returns for the current policy) to a control setting (solving the MDP, finding the optimal value function). The added lines describe the so-called improvement process, which drives the policy to improve locally, with respect to the current value function. Thus, the state-action value function q is continuously estimated for a given policy π, which determines the current behavior, and at the same time this so-called behavior policy π is improved. Essential are the lines 5 and 8, which contain this improvement step: the given policy is continuously improved by selecting the next action and state A, S (respectively A', S') according to an ϵ-greedy procedure, which we have introduced in section 2.2.5.3: the policy π is locally improved, by acting greedy with respect to the current q.

SARSA is a widely used algorithm in reinforcement learning applications and is very popular, especially in online learning setting and due to its simplicity in implementation. In the next sections, this algorithm will be extended, making him slightly more complex and significantly more powerful.

3.3.3.3 SARSA with Function Approximation

Since we have a large state and action space in our application, a simple SARSA algorithm (see algorithm 1) is not efficient, simply because of the required computational effort and the limited time. Not only is the computational cost a challenge, but often there is not enough data to fill the algorithm accurately. In our case it will probably never be possible to show all screen configurations to the user, in fact only a very small selection of all possibilities will probably ever be shown to the user, since we have so many choices. This means that for a large number of pairs (A, S) we will not get any estimates for their value at all. In a setting with such large spaces, one cannot expect to find the overall optimal policy.
The new goal will be to find a good approximation solution. This is attempted by the *generalization approach*, where one tries to generalize findings from the limited subset of the obtained data to a larger subset. The kind of generalization we need is often called *function approximation*, since it takes samples from a desired function (in our case the state-action value function q) and tries to generalize, given the observed sample-values, in order to construct an approximation of the entire function. The methods used in this context belong to the "supervised learning" domain of machine learning, as it can be seen as working with labeled data (matched pairs of inputs (S,A) and outputs $q(S, A)$).

Function approximation for the state-action value function $q(s, a)$:
In the SARSA context, we need to approximate the action value function q. We denote the approximation by $\hat{q}$ and introduce the weight vector $\mathbf{w}$, parameterizing the state-action value function: $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$. Therefore, $\hat{q}(s, a, \mathbf{w})$ is the approximation of $q(s, a)$, and d is the dimension of the weight vector $\mathbf{w}$. Typically, the dimension of the weight vector $\mathbf{w}$ is significantly smaller than the cardinality of the state space, $d \ll |\mathcal{S}|$, which leads to the fact that by adjusting the weight vector $\mathbf{w}$, the value function is directly adjusted for multiple state-action pairs simultaneously.

Algorithm 2 shows the pseudo-code of SARSA with function approximation, where the gradient-descent methodology is used as the function approximation method. One can observe the gradient-descent technique in algorithm 2, on line 10 and 13. The description of the methodology follows.

Algorithm 2 SARSA with Semi-Gradient as Function Approximation for estimating $q \approx q_*$

```
1: Input: a differentiable state-action value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
2: Algorithm parameters: step-size $\alpha \in (0, 1]$, small $\varepsilon > 0$, discount rate $\gamma \in [0, 1]$
3: Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$ )
4: Loop for each episode:
    5: Initialize $S$
    6: Choose $A$ from $S$ using $\varepsilon$-greedy policy derived from $\hat{q}(S, \cdot, \mathbf{w})$
    7: Loop for each step of episode:
        8: Take action $A$, observe $R, S'$
        9: If $S'$ is terminal:
            10: $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R - \hat{q}(S, A, \mathbf{w})]\nabla\hat{q}(S, A, \mathbf{w})$
            11: Go to next episode
        12: Choose $A'$ from $S'$ using $\varepsilon$-greedy policy derived from $\hat{q}(S', \cdot, \mathbf{w})$
        13: $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[R + \gamma\hat{q}\left(S', A', \mathbf{w}\right) - \hat{q}(S, A, \mathbf{w})\right]\nabla\hat{q}(S, A, \mathbf{w})$
        14: $S \leftarrow S'; A \leftarrow A'$
```

Gradient Descent Methodology
The strategy of the gradient descent methodology in our setting is to try to minimize the error (or squared error) on the observed samples. We therefore want to minimize the following expression:

$$[q(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]^2 \tag{3.3}$$

This is done by adjusting the weight vector $\mathbf{w}$ after receiving a pair (S_t, A_t), by a small amount in the direction that would most reduce the error on that pair. Thus, the update from eq. (3.4) follows, where the update of the weight vector $\mathbf{w}$ is proportional to the negative gradient of the squared error, which indicates the direction that reduces the error most:

$$\begin{aligned} \mathbf{w}_{t+1} &:= \mathbf{w}_t - \frac{1}{2}\alpha\nabla [q(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha [q(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla\hat{q}(S_t, A_t, \mathbf{w}_t) \end{aligned} \tag{3.4}$$

But we still need to make a small adjustment to eq. (3.4) before it is usable for our setting. In fact, we do not know the term $q(S_t, A_t)$ from eq. (3.4). That is precisely what we want to approximate. Therefore, we need to use the best possible approximation for this term, and since we are in the TD-setting we will re-use our TD-estimate, which is $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)$. By inserting this estimated term, which depends on a next sample (S_{t+1}, A_{t+1}), we introduce a methodology that is called *bootstrapping*. The update is therefore no longer the update directly on $q(s, a)$, as in algorithm 1 on line 9, but the one from algorithm 2 on line 13, indirectly changing $q(s, a)$ via the weight vector:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla\hat{q}(S_t, A_t, \mathbf{w}_t) \tag{3.5}$$

Because of this slight change in the gradient-descent formula (eq. (3.4) vs eq. (3.5)), the method is actually called "*Semi*-Gradient Methodology" (see naming of algorithm 2).

Additions to Function Approximation: the Linear Approach
The question arises what kind of parameterization is chosen for the function $\hat{q}$, which is required as input in algorithm 2, line 1. A common approach is to approximate a *linear function*, where the value of a state-action pair is the inner product between the weight vector $\mathbf{w}$ and a state-action dependent *feature vector* $\mathbf{x}(S, A)$, which has the same dimension d as the weight vector:

$$\hat{q}(S, A, \mathbf{w}) := \mathbf{w}^\top \mathbf{x}(S, A) = \sum_{i=1}^{d} w_i x_i(S, A) \tag{3.6}$$

In case of such a linear function approximation, the gradient $\nabla\hat{q}(S, A, \mathbf{w})$ has a particular simple form:

$$\nabla\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A). \tag{3.7}$$

The linear approach is very popular due to its simplicity. But the question remains on how to set the features $\mathbf{x}(S, A)$, because they have to be defined manually. Furthermore, due to linearity, we have the limitation that dependencies within features cannot be learned. It is possible to get around this limitation, by introducing additional features that map such dependencies (e.g. product of two features as a new one). But this has a lot of manual leeway, which can easily lead to something being overlooked. After all, the system can only learn what you allow it to learn.

Therefore, one should also test out a non-linear function approximation, such as for example using an Artificial Neural Network (ANN), where we do not face such constraints. In this case, however, we have to be careful concerning the architecture (how many layers, how many nodes per layer) of the ANN, because too complex means too slow and inefficient and not complex enough could lead to not being able to learn the real unknown function. Each method has its advantages and disadvantages, which should always be carefully evaluated.

3.3.3.4 SARSA(λ) with Function Approximation

With the algorithm we have developed so far, we have ensured that we are in an online and an on-policy learning setting. We can also handle the large state and action spaces and we have integrated continuous exploration (with an ϵ-greedy method). Now we want to ensure one additional and very important point. That is the *sequential knowledge*. This will be enabled by the concept of eligibility traces.

Eligibility Traces
The idea behind the eligibility traces is to make not only the last element (i.e. the last action and state) responsible for the respective success or failure (reward), but also those from the "near-past". For each state-action pair we introduce a short-term memory vector: the eligibility trace z_t, which is drawn parallel to the long-term weight vector $\mathbf{w}_t$ and has the same dimension d as the weight vector. The eligibility trace should indicate how "eligible" a certain component of the weight vector was, at a certain point in time. The rough idea is that if a component of $\mathbf{w}_t$ has helped to calculate (or approximate) a value $\hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$, then the associated component in the vector z_t will be increased, and will then again slowly fade away over time. How fast this fade-away happens is determined by a newly introduced parameter, the so-called *trace-decay parameter* $\lambda \in [0, 1]$. It determines with which speed the traces decay. The eligibility traces thus support the learning process by being able to influence the weight vector $\mathbf{w}_t$ on a short-term basis (see eq. (3.9)), and this weight vector then drives the estimated value $\hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$.
Mathematically, this means the following: At the beginning of each episode, the eligibility trace vector z_0 is initialized to zero, is then increased by the value of the gradient $\nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$ (which indicates which components were "active"), and is then decreased by $\gamma\lambda$ at each time step t.

$$
\begin{aligned}
\mathbf{z}_{-1} &:= \mathbf{0} \\
\mathbf{z}_t &:= \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right), \quad 0 \leq t \leq T
\end{aligned}
\tag{3.8}
$$

This leads to the following update rule for the new algorithm:

$$
\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t, \text{ with}
$$

$$
\begin{aligned}
\delta_t &:= R_{t+1} + \gamma\hat{q}\left(S_{t+1}, A_{t+1}, \mathbf{w}_t\right) - \hat{q}\left(S_t, A_t, \mathbf{w}_t\right) \\
\mathbf{z}_t &:= \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right), \quad 0 \leq t \leq T \\
\mathbf{z}_{-1} &:= \mathbf{0} \\
\lambda &= \text{discount rate} \\
\gamma &= \text{trace-decay parameter} \\
\alpha &= \text{step-size parameter}
\end{aligned}
\tag{3.9}
$$

The Role of the Trace-Decay Parameter λ

Let us have a closer look at what happens with different values of the trace-decay parameter $\lambda \in [0, 1]$.

With $\lambda = 0$, the first term in $\mathbf{z}_t = \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$ from eq. (3.9) vanished and only the gradient $\nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$ remains. If we plug this into the weight-update from eq. (3.9), we get the following update rule:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t \\ &= \mathbf{w}_t + \alpha\left[R_{t+1} + \gamma\hat{q}\left(S_{t+1}, A_{t+1}, \mathbf{w}_t\right) - \hat{q}\left(S_t, A_t, \mathbf{w}_t\right)\right]\nabla\hat{q}\left(S_t, A_t, \mathbf{w}_t\right).\end{aligned}$$

This update rule is exactly the same as in eq. (3.5). Therefore, with $\lambda = 0$ we obtain the SARSA algorithm described in algorithm 2. In this case, only the one step preceding the current one is being updated.

The larger the λ value, the more of the preceding states are updated. So, one goes further back in time to make adjustments. But each state further away in time is updated less, because the corresponding eligibility trace is smaller (it slowly fades away over time).

With λ=1, there is no fading-away and we adjust all steps, from start until the terminal state, of an episode (assuming to be in a non-discounted setting with $\gamma = 1$).

Figure 3.4 illustrates the influence of the trace-decay parameter λ.

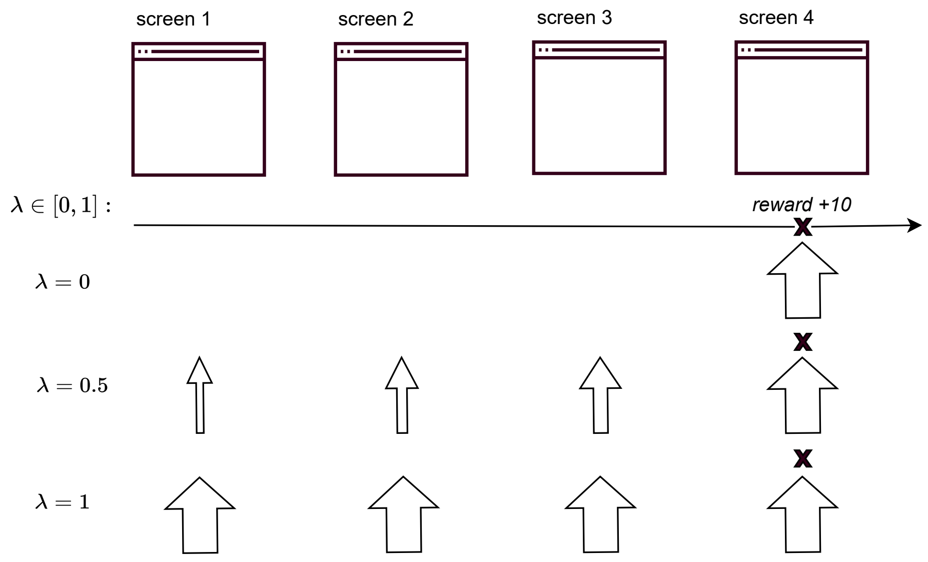


Figure 3.4: Role of the trace-decay parameter λ: The thickness of the arrows illustrates how strong the respective screen is considered to be part of the success (reward +10) received for screen 4.

Pseudo-Code of SARSA(λ) with Linear Function Approximation
At this point, we are ready to introduce the "SARSA(λ) with linear function approximation" algorithm, whose pseudo-code is illustrated in algorithm 3. We use the linear function approximation approach as function approximation, which we introduced above. Therefore, we can use the above derived equations eq. (3.6) and eq. (3.7): $\hat{q}(S, A, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(S, A)$ and $\nabla \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$.
If we plug these equalities in the update rule from eq. (3.9), we obtain the update rule, which we see in algorithm 3 on line 13 and 18.

Algorithm 3 SARSA(λ) with linear function approximation for estimating $\mathbf{w}^\top \mathbf{x} \approx q_*$

```
1: Input: a feature function $\mathbf{x}(S, A)$
2: Algorithm parameters: step-size $\alpha > 0$, trace-decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$,
   discount rate $\gamma \in [0, 1]$
3: Initialize $\mathbf{w} = (w_1, \dots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \dots, z_d)^\top \in \mathbb{R}^d$
4: Loop for each episode:
    5: Initialize $S$
    6: Choose $A$ from $S$ using $\varepsilon$-greedy policy derived from $\hat{q}(S, \cdot, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(S, \cdot)$
    7: $\mathbf{z} \leftarrow \mathbf{0}$
    8: Loop for each step of episode:
        9: Take action $A$, observe $R, S'$
        10: If $S'$ is terminal:
            11: $\delta \leftarrow R - \mathbf{w}^\top \mathbf{x}(S, A)$
            12: $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \mathbf{x}(S, A)$
            13: $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
            14: Go to next episode
        15: Choose $A'$ from $S'$ using $\varepsilon$-greedy policy derived from $\mathbf{w}^\top \mathbf{x}(S', \cdot)$
        16: $\delta \leftarrow R + \gamma\mathbf{w}^\top \mathbf{x}(S', A') - \mathbf{w}^\top \mathbf{x}(S, A)$
        17: $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \mathbf{x}(S, A)$
        18: $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
        19: $S \leftarrow S'; A \leftarrow A'$
```

3.3.3.5 Convergence

A proof for the convergence of the prediction method TD(λ) with linear function approximation, which is used in algorithm 3, is demonstrated in [3], chapter 9.4. To guarantee this convergence, the following conditions for the step-size α must be satisfied:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty, \tag{3.10}$$

whereby $\alpha_n(a)$ denotes the step-size parameter used to process the reward received after the n-th selection of action a. This means that the step-size changes over time. The first condition is necessary to guarantee that the steps are large enough to eventually clean out initial conditions or random fluctuations. The second condition guarantees that the steps eventually get small enough to ensure convergence. For example, both conditions are satisfied with $\alpha_n(a) = \frac{1}{n}$. It should be noted, that sequences with step-size parameters satisfying the conditions from eq. (3.10) often converge very slowly, and are therefore widely used in theoretical work, but only rarely in real-life applications or even empirical research [3].

In the two papers [29] and [30], the convergence topic of *SARSA(λ) with linear function approximation* method is being analyzed. They demonstrate that the weight vector $\mathbf{w}$ is not guaranteed to converge to a fixed point, but must converge into a bounded region. We take the liberty to omit the details here. It should be mentioned, however, that certain restrictions apply to guarantee a convergence into a bounded region, which cannot be modeled, but depend on the (unknown) transition probability function.
Such convergence proofs demonstrate the theoretical substance of the method we have chosen. But whether the method converges (and converges fast enough) in our learning problem setting can only be observed after implementation, and after tuning with certain parameters and representation choices. Because, although the method is generally valid, naturally it remains true: only if the problem has been modeled correctly, the learning problem can be solved effectively. In addition, even if we could not observe a convergence, this could also occur because the problem itself does not converge: It could be that the optimum for a user is a constant oscillation of stimuli. This is something we cannot know at the present time. But we need to select an algorithm that could handle both situations: convergence or continuous oscillation. With the choice we have made, this is theoretically fulfilled.

3.3.3.6 Further Topics

In the following, some topics are addressed, which could or should be further investigated in the future. They can offer explanations if the application does not show the desired learning effect, and they can highlight possible error potentials or also offer possible optimization suggestions, if the application shows the correct learning effect, but not with the desired performance (i.e. if the calculations are not efficient).

Choice of Learning Parameters
There are no strict guidelines or rules on how to choose the learning parameters. Therefore, the parameters should also be tested with different values, to see which values produce the best learning effects. Our learning parameters are:

1. the step-size α
2. the discounting rate γ
3. the trace-decay parameter λ
4. the ϵ parameter for the ϵ-greedy procedure.

It is common to define some parameters not as fixed, but as a function of the time step t. For example, it might be useful to define the step-size α (or even ϵ) as gradually decreasing over time to reduce the "jumps" and converge to a local (and hopefully global) optimum.

Different Options for Function Approximation
As already mentioned in section 3.3.3.3, there are many different possibilities for function approximations. Each method has its advantages and disadvantages, which should be carefully assessed. It can make sense to start with a less complex method, in order to generically test the application and its flow. In a further step, however, different methods should be tested to enable optimal learning. Within the methods, there is sometimes even a larger leeway, as for example with an artificial neural network (ANN), where one must determine the complete architecture of the network. All these possibilities and their influences on performance should be kept in mind.

Choice of Exploration Method
As with function approximation methods, there is a wide choice of exploration methods (to list some: upper confidence bound, Boltzmann exploration, Thompson sampling Olivier Sigaud [24]). It certainly makes sense to start with an ϵ-greedy approach, since it is by far the simplest in terms of implementation effort. Nevertheless, one should already be careful with this approach. In the ϵ-greedy approach a maximum value is calculated over all actions: $\arg\max_a q(s,a)$. This can be computationally expensive. There are however implementation techniques, with which the complexity of the computational effort can be shortened and which should be considered.
In our application, it might also make sense to limit the respective possible next actions, so that the maximum has to be selected over fewer values and is thus more efficient. This means more implementation effort, because a function must be written, which determines the allowed next actions depending on the last action and state. But in the long run one might gain a lot of computational speed.

Choice of State Representation
The choice of state and action representations is, again as Sutton [3] nicely puts it, *"more art than science."* The choice of state and action representation should therefore always remain flexible and one should always consider whether there is not a completely different concept to represent the current situations. The first choice for the k-th order history (see section 3.2.1) was made mainly because one wanted to approximate the Markov property as well as possible. However, it might make sense to let go of this concept and try a much more compact format. For example, one possibility would be to define only the last action a_t as the next state s_{t+1}. However, the question arises whether this would be sufficient, since we are not in a rather "simple" setting, such as learning a path to get out of a maze, but in a more complex domain: the human brain. The big challenge we face, is that we do not know how much the different, maybe strung together, sequences can affect the user's perception.

Chapter 4

App Implementation

This chapter describes the process of developing an application prototype. First, the environment and general setup of the application (”app”) is described. Then the design and implementation is explained in more detail. This is followed by some impressions of the developed prototype, with screen-shots and code-snippets. Lastly, an evaluation and some discussion points are provided.

4.1 App Environment and Setup

Since the application has many dynamic components, the decision was made to develop the application as a type of game. Therefore, it was decided to develop the front-end of the application in a game engine. For this purpose the game engine *Godot* was chosen. Godot (2014) is an open-source tool for the development of 2D and 3D games, which also offers extensions for VR (Virtual Reality) games. The application is created within Godot and can then be exported to macOS, Linux, Windows, Android, iOS or HTML applications. The game engine Godot offers an interactive editor and attractive features such as physical simulations and custom animations. The part of the application written in Godot defines the front-end of the application. An approach was chosen to write the back-end externally in Python scripts. In these scripts the reinforcement learning algorithm is written, which will decide what the front-end will show to the user. The back-end is connected to the front-end via *Flask*. Flask is a web framework, written in Python, which contains various tools and libraries that support developers to efficiently build online applications. Flask is used to create a server through which Godot can then communicate with the Python scripts. Initially, only the local host is used, so the application runs only locally on the machine, on which the scripts are executed. As a next step, if one wants to deploy the application to the Google Play Store, for example, the application must be deployed to an external Python server, which is chargeable (often on monthly rates). The figure 4.1 illustrates the flow of the application including the Godot and Flask components.

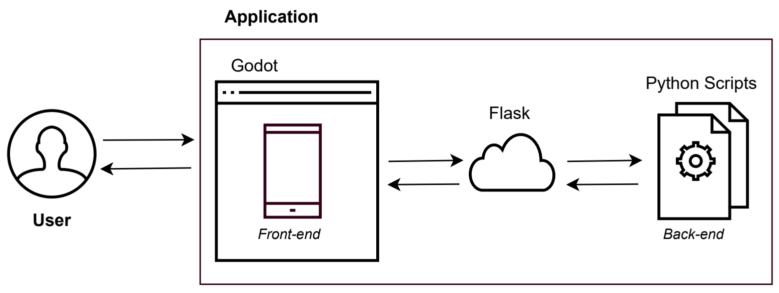


Figure 4.1: Flow of the application including the Godot and Flask components. The communication between Godot and Python is provided by Flask, a web framework supporting the development of online applications.

4.2 Prototype

In this section the design, implementation and evaluation of the prototype is described. This prototype is a starting point for further extensions. The primary goal is to enable the flow from figure 4.1. Therefore, for the time being, only a small set of possible tuning parameters, which were introduced in section 3.2.2, is used to limit the implementation effort and to set the focus on the basic functionalities.

4.2.1 Design

Tuning Parameters

The tuning parameters, which decide what is displayed on the application screen, have been limited to a single feature: ”shape”. The following two attributes are provided for this feature: form and color. Possible values for form are {circle, square} and for colors a value from the set {red, green, blue}. This results in two tuning parameters: {form, color}, with 2 and 3 different values respectively. So, for each screen we have a choice of $6 = 2 * 3$ possibilities (all possible combinations of forms and colors).

Actions

Recall from section 3.2.1 that in our MDP setting an action is a particular screen configuration. Using the above mentioned tuning parameters, we get the following set of actions $\mathcal{A}$:

$\mathcal{A} = \cup_{i=1}^{6} a_i$

a_1 := ’CircleRed’

a_2 := ’CircleBlue’

a_3 := ’CircleGreen’

a_4 := ’SquareRed’

a_5 := ’SquareBlue’

a_6 := ’SquareGreen’

Rewards

Three feedback buttons have been implemented, which the user can press at any time: {positive feedback, neutral feedback, negative feedback}. The neutral feedback button was introduced to facilitate signal transfer. For further prototypes the neutral feedback button should be omitted and the neutral signal should be sent when no feedback was provided during a certain period of time. When one of the three buttons is pressed, a respective event is triggered. The numeric signal corresponding to the feedback is sent to the back-end: {+10, 0, -10}. We thus obtain the following set of rewards $\mathcal{R}$:

$\mathcal{R} := \{+10, 0, -10\}$

States

As described in section 3.2.1, we use the k-th order history for the state representation, collecting the last k actions and rewards:

$$s_t = (r_t, a_{t-1}, r_{t-1}, ..., a_{t-k}) \quad \text{for some} \quad k \geq 1. \tag{4.1}$$

For the prototype, the decision was made for $k = 1$, in order to keep the complexity momentarily as small as possible. We thus obtain:

$s_t := (r_t, a_{t-1})$	state at time t, with
$r_t \in \mathcal{R}$	reward at time t,
$a_{t-1} \in \mathcal{A}$	action at time t-1,

Since $|\mathcal{R}| = 3$ and $|\mathcal{A}| = 6$, there are $18 = 3 * 6$ possible states, in which our application can be situated at a certain point in time t, and we can therefore note:

$\mathcal{S} := \cup_{i=1}^{18} s_i$.

We omit to list all 18 possible states. One example is the state $s_i = (+10,$ 'CircleRed'), in which one is situated if the last chosen action by the RL-algorithm was 'CircleRed' and the user pressed the positive feedback button (reward +10) for this projection.

Algorithm

To begin, we want to investigate how many values the state-action value function from eq. (2.9) in our setting is taking on. Recall that the state-action value function $q(s, a)$ is defined on the set $\mathcal{S} \times \mathcal{A}$ and is taking values in $\mathbb{R}$. If we calculate:

$$|\mathcal{S} \times \mathcal{A}| = |\mathcal{S}| * |\mathcal{A}| = 18 * 6 = 108,$$

we get 108 possible values, that our algorithm would need to learn.

Side note: short calculation how this number would scale in dependence of k (from the k-th order history), and the size of the set of actions $|\mathcal{A}|$:
$s_t = (r_t, a_{t-1}, r_{t-1}, ..., a_{t-k}) \Rightarrow |\mathcal{S}| = k * |\mathcal{R}| * |\mathcal{A}| \Rightarrow |\mathcal{S} \times \mathcal{A}| = k * |\mathcal{R}| * |\mathcal{A}| * |\mathcal{A}|$.

Since we have a limited number of parameters available in the context of the first prototype (feature: ”shape”, attributes: ”form”, ”color”), and consequently a rather small number of possible q-values ($|\mathcal{S} \times \mathcal{A}| = 108$), it was decided to implement the simple SARSA algorithm from algorithm 1, which is again illustrated in algorithm 4, but now slightly adapted to fit into our setting. It should be noted that for a further prototype, when one increases the number of tuning parameters, and thus the complexity of the learning problem, one should move to the appropriate adapted algorithms, as discussed in chapter 3.3 (include function approximation and eligibility traces).

4.2.2 Implementation

Code
The following versions were used to implement the prototype:

- python v3.9
- godot v3.4.4
- flask v2.1.3

The communication between the Python back-end and the Godot front-end is done via http requests over a local host initiated by Flask. Algorithm 4 represents the flow of the prototype from the back-end perspective. The communication via http requests happens in line 6 and 7. S’ and A’ represent the next state and next action. On line 10 we observe the TD-update rule from eq. (3.2), explained in section 3.3.3.

Algorithm 4 Prototype Algorithm

```
1: Set $\alpha$, $\epsilon$, $\gamma$
2: Set $q(S, A) = 0$, for all $S \in \mathcal{S}, A \in \mathcal{A}$
3: Choose $S$ randomly
4: Choose $A$ from $S$ using $\epsilon$-greedy policy derived from $q$
5: While the application is running:
    6: Send action $A$ to front-end
    7: Wait for user feedback and observe $R$
    8: Calculate $S'$
    9: Choose $A'$ from $S'$ using $\varepsilon$-greedy policy derived from $q$
    10: $q(S, A) \leftarrow q(S, A) + \alpha\,[R + \gamma q\,(S', A') - q(S, A)]$
    11: $S \leftarrow S'; A \leftarrow A'$
```

Impressions
This section is intended to provide some impressions from the developed prototype. Some screen shots from the prototype will follow. As one can observe, the texts inside the application have been written in German, since this work has been conducted in Switzerland, and therefore it is most likely that the first users of the application have German as their native language. Figure 4.2 shows the welcome screen, which is displayed when the user starts the application.

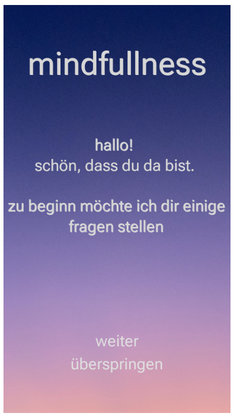
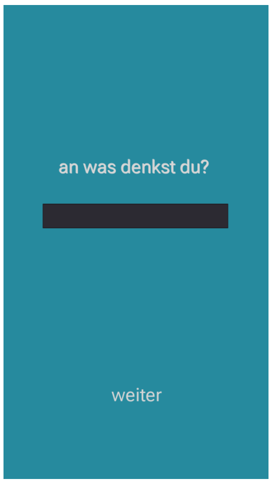
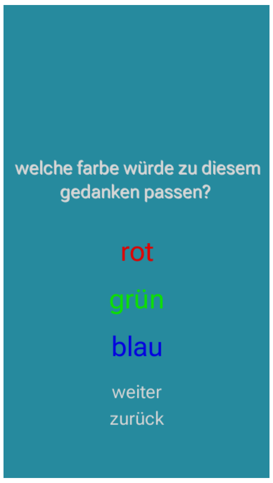


Figure 4.2: Welcome screen of the application.

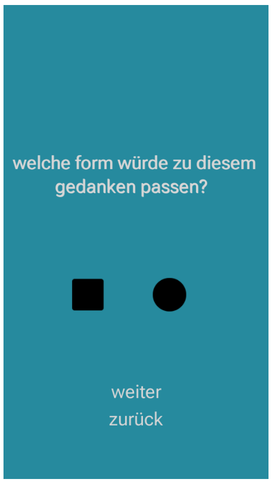
After having started the application, the user is asked to provide some personal inputs, see figure 4.3. The goal of these inputs is to dedicate the current session to a topic, and then to objectify the intangible topic by coupling some tangible attributes (as color and shape) to it. We learned about this coupling technique in section 2.1.2.2. Anything can be considered as a topic. It could be something that affects or concerns the user at this very moment of using the application, as for example: limb pain, fatigue, stress, lack of concentration, mental confusion, etc.
After the user has provided valuable inputs, he is redirected to a new screen where the main functions of the application are displayed. The user sees a calming video in the background (at the moment: calming views out of the universe). In the foreground specific stimulations are projected to the user. Figure 4.4 illustrates some possible stimulations.
The feedback buttons, which are visible at any time, allow the user to give feedback on the respective projection. The projection will then change based on the feedback received. During this process, the RL-algorithm is trained in the background and the user receives an increasingly personalized algorithm over time. Figure 4.5 illustrates what is triggered in the front-end, inside the Godot scripts, when the user presses the positive feedback button. It also illustrates what gets triggers in the back-end, on the Python side, when the Godot engine sends the positive feedback signal to the back-end. Similar happens when the user presses the negative, respectively the neutral, feedback button.
At the moment, the learning progress of the algorithm is not memorized over user-sessions. This means that the algorithm must be re-trained from the beginning when the application gets restarted. For future developments, it is planned to start with a login screen, where the user has to give his credentials, so that the application can pick up the user-specific algorithm from a database, and the learning process does not need to start from the very beginning. It could also make sense to store different algorithms per user, depending on the topic the user is currently dealing with (which is provided as input in figure 4.3), for example: separate algorithms for topic "sleep problems", for topic "headaches", for topic "concentration difficulties".



(a) Dedicating a topic to the session: The user can type in a topic, for example: ”strong headache”.

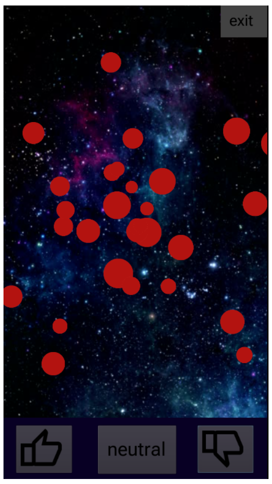


(b) Association of a color to the topic. Choices are: red, green, blue.

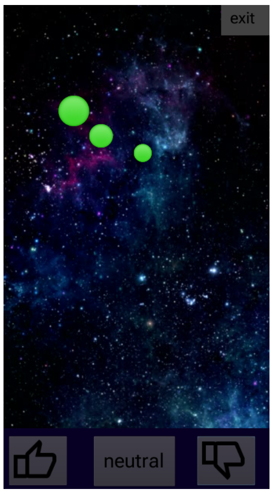


(c) Association of a shape to the topic. Choices are: square, circle.

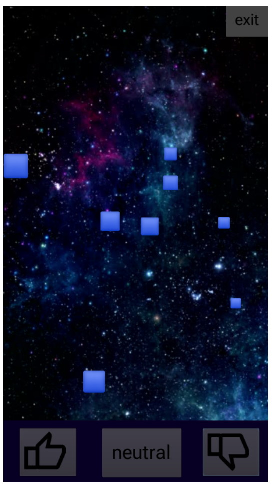
Figure 4.3: Screens asking for user inputs, to objectify the topic of the current session and enable a coupling effect.



(a)

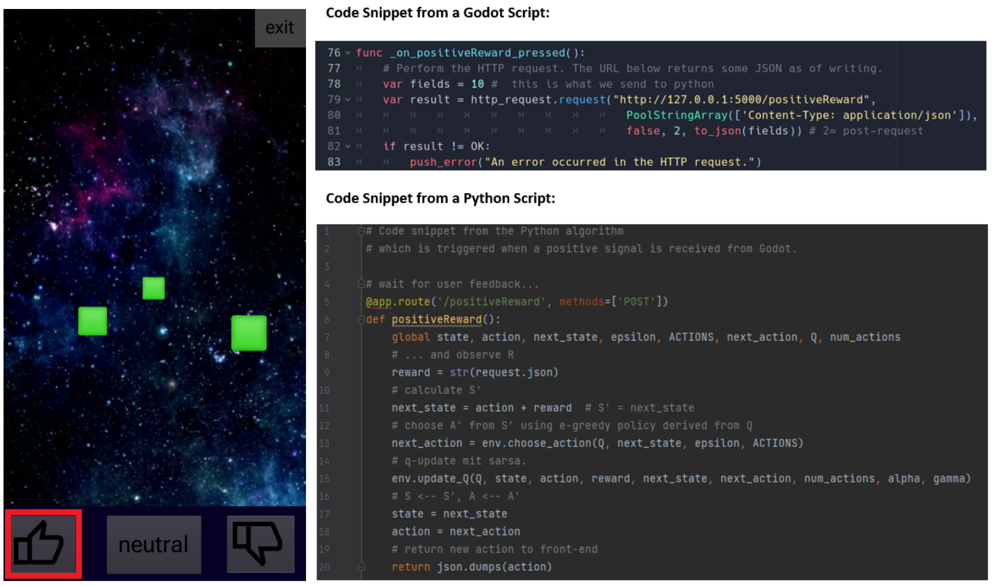


(b)



(c)

Figure 4.4: Screens of possible stimulations, which are projected to the user. They contain different amounts of moving circles and squares that appear in different places, in different sizes and colors, and move at different speeds.



Code Snippet from a Godot Script:

```
func _on_positiveReward_pressed():
	# Perform the HTTP request. The URL below returns some JSON as of writing.
	var fields = 10 #  this is what we send to python
	var result = http_request.request("http://127.0.0.1:5000/positiveReward",
										PoolStringArray(['Content-Type: application/json']),
										false, 2, to_json(fields)) # 2= post-request
	if result != OK:
		push_error("An error occurred in the HTTP request.")
```

Code Snippet from a Python Script:

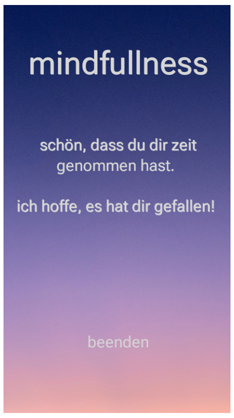
```
# Code snippet from the Python algorithm
# which is triggered when a positive signal is received from Godot.

# wait for user feedback...
@app.route('/positiveReward', methods=['POST'])
def positiveReward():
    global state, action, next_state, epsilon, ACTIONS, next_action, Q, num_actions
    # ... and observe R
    reward = str(request.json)
    # calculate S'
    next_state = action + reward  # S' = next_state
    # choose A' from S' using e-greedy policy derived from Q
    next_action = env.choose_action(Q, next_state, epsilon, ACTIONS)
    # q-update mit sarsa.
    env.update_Q(Q, state, action, reward, next_state, next_action, num_actions, alpha, gamma)
    # S <-- S', A <-- A'
    state = next_state
    action = next_action
    # return new action to front-end
    return json.dumps(action)
```

(a) Event of clicking the positive feedback button

(b) Code-Snippets, showing the chain of actions that gets triggered by event (a).

Figure 4.5: Implementation in-sights. Different events trigger different chains of action, hereby illustrated by the event of a positive user feedback.

The user can end the stimulation sequences at any given time, by pressing the exit button, which is always visible on top of the screen (see fig. 4.4). In this case, a final screen appears, illustrated in figure 4.6, to thank the user for his time and dedication, before the user can close the application completely. In a later prototype, further user input could be asked at this point, such as, "Did you like the session? How do you feel after this session?", which could be used to evaluate the effectiveness of the sessions.



mindfullness

schön, dass du dir zeit genommen hast.

ich hoffe, es hat dir gefallen!

beenden

Figure 4.6: Final screen, thanking the user for his time and dedication.

4.2.3 Evaluation

Learning Effect
In order to test the implemented algorithm without requiring real user interactions from the front-end, the algorithm was tested purely in the back-end. This required simulating the behavior of a possible user, which was implemented as follows:

Algorithm 5 Simulating User Feedback

```
1: if action == 'CircleRed' then
       reward = REWARD_POSITIVE
2: else
       reward = REWARD_NEGATIVE
3: end if
```

The algorithm was thus programmed, so that as soon as a red circle was selected as an action, the positive reward of +10 was sent. For any other action, the user response was programmed to -10.

The following parameters were used:

- $\gamma = 0.9$
- $\alpha = 0.6$
- $\epsilon = 0.2$

In the test scenario, the while loop on line 5 of algorithm 4 ran for $n = 50$ steps. This means that 50 different scenes and user responses were simulated. We were interested in the development of the received rewards, respectively how fast the algorithm finds out the preference of the test-user. It was measured how often the action "CircleRed" was shown within the 50 possible screens. Figure 4.7 shows the rewards per time step (+10 for 'CircleRed', -10 for everything else), as well as the cumulative reward.

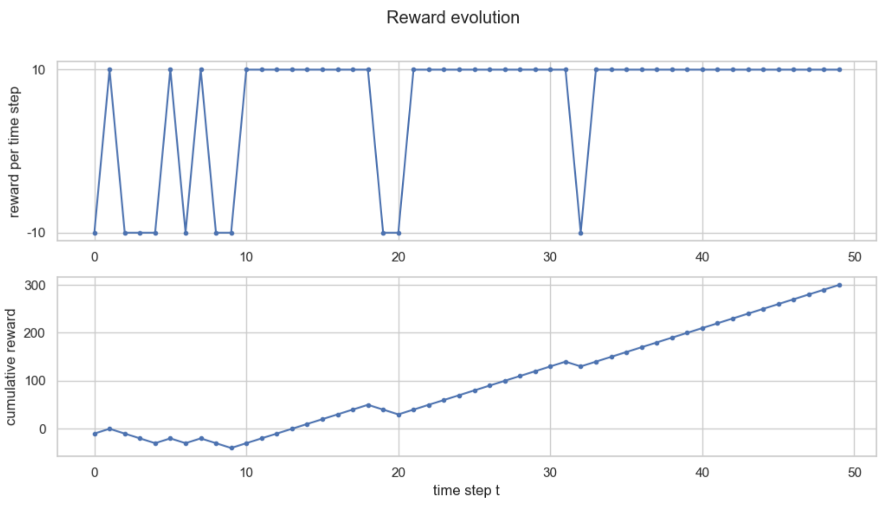


Figure 4.7: Results of the test-scenario, showing the evolution of the collected rewards.

Looking at figure 4.7 and counting how many times we received a reward of +10, we see that 40 times out of 50, the red circle was chosen as action. We can also observe that negative feedback was received more often in the early stages (reward -10). This can be explained by the need to learn the trend in the beginning. As soon as the trend for the red circle was recognized, it was selected most often. Nevertheless, some times the red circle was not chosen (see time steps t= 19, 20 and 32), which is to be explained by the fact that we still have an ϵ-chance of choosing a random action. With an even smaller ϵ, such later negative rewards could further be minimized, but this is not necessarily the goal, since the application should not constantly project red circles, but still have a certain variety.

The results of the test-scenario show that the implemented algorithm is able to learn the test user's preference for the red circle. The learning rate is impressively fast (learning curve already visible with n=50), but this is also due to the fact that we do not have, yet, many tuning parameters, respectively q-values, to learn.

Discussion
This first prototype achieves the desired flow of events. The main functionalities are ensured. There is already a functional underlying reinforcement learning algorithm, which is capable of learning a certain strategy and responding to user feedback. These were the most important goals to be achieved. Since no computer scientists were involved in the development of this prototype, but instead the programs (such as the game engine Godot) were discovered and learned autonomously, the process starting from the design to the completion of the implemented prototype required a considerable amount of time. The obtained outcome is also therefore more than satisfying.

In a next phase, the application should be tested with test subjects, not only in terms of functionality, but also in terms of usability and user-friendliness. The already existing parameter set should then also be extended - based on the feedback received. In addition to user-testing, it would also be helpful to involve experts from the fields of psychology and especially pain therapies. Their expertise could be used to adapt the application in an even more targeted manner. However, the prototype already provides an illustration of the central idea. With this prototype as a basis, the project can be further developed.

Chapter 5

Conclusion

In the context of this thesis, one was interested in exploring the opportunities that reinforcement learning can offer to support the treatment of pain, as well as to alleviate accompanying symptoms. It has therefore been approached to develop a self-learning application, which uses techniques from psychological pain management, and thereby sends various stimulations to the user of the application. The main advantages such an application offers, are therapist-independency, full personalization, and availability at any given time. With a combination of reinforcement learning techniques and projection of stimulations triggering neural effects, the application would offer new potential in the area of pain management.

For this purpose, the task was analyzed and mapped into the setting of a Markov decision process (MDP). It had to be defined what was considered as an "agent" and "environment", how we represent the "actions", "states" and "rewards", and what requirements our setting imposes on a learning algorithm. In a further step, it was also addressed what data would be used and how to represent it optimally. Thereby, several encoding methods were introduced. On these grounds, a suitable learning algorithm could be chosen: "SARSA(λ) with function approximation". With this algorithm the following main criteria can be fulfilled: model-free learning, online learning, on-policy learning, handling of large action and state spaces, handling possibility of non-existence of global optimum, and handling sequential knowledge.

Based on the detailed analysis and modelling of the task, a prototype of the application was developed. The prototype provides an illustration of the core idea: it achieves the desired flow of events; the main functionalities are ensured; and there is already a functional underlying reinforcement learning algorithm, which is capable of learning a certain strategy and responding to user feedback. This prototype is considered as a starting point for further developments.

It should be mentioned that this study did not test a specific hypothesis (the prototype of the application was not tested on patients), but instead built a foundation of a framework, on which one can build in the future. It was explored what should be taken into consideration for the development of this project, as well as for other similar projects. This thesis serves as the first steps in an innovative project. In a next phase, it should be considered to further develop the prototype in interdisciplinary cooperation with specialists from different fields, such as pain therapy, psychology, and hypnosis. It would be enriching and rewarding if this project could also support other initiatives, in which innovation and new technologies are being considered to ameliorate the health and well-being of people all around the world.

Bibliography

[1] M. H. J. G. I. Ph.D., *Understanding Pain: What It Is, Why It Happens, and How It's Managed*, 1st ed., ser. American Academy of Neurology. Demos Health, 2006.

[2] D. R. Patterson, *Clinical Hypnosis for Pain Control*, 1st ed. American Psychological Association (APA), 2010.

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

[4] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[5] M. F. Yam, Y. C. Loh, C. S. Tan, S. Khadijah Adam, N. Abdul Manan, and R. Basir, "General pathways of pain sensation and the major neurotransmitters involved in pain regulation," *International journal of molecular sciences*, vol. 19, no. 8, p. 2164, 2018.

[6] M. Costandi, *Neuroplasticity*, ser. The MIT Press Essential Knowledge series. The MIT Press, 2016.

[7] R. Leary, *Subliminal Psychology 101*, 1st ed. Robert Leary, 2020.

[8] A. H. Wobst, "Hypnosis and surgery: past, present, and future," *Anesthesia & Analgesia*, vol. 104, no. 5, pp. 1199–1208, 2007.

[9] P. P. Bernatzky, *Rezeptivev Musiktherapie bei Morbus Parkinson*, 1st ed. VDM Verlag Dr. Müller, 2008.

[10] H.-P. Hesse, *Musik und Emotionen*, 1st ed. Springer-Verlag Wien GmbH, 2003.

[11] G. B. Alparslan, B. Babadag, A. Özkaraman, P. Yildiz, A. Musmul, and C. Korkmaz, "Effects of music on pain in patients with fibromyalgia," *Clinical rheumatology*, vol. 35, no. 5, pp. 1317–1321, 2016.

[12] A. J. Elliot and M. A. Maier, "Color and psychological functioning," *Current directions in psychological science*, vol. 16, no. 5, pp. 250–254, 2007.

[13] A. J. Elliot, "Color and psychological functioning: a review of theoretical and empirical work," *Frontiers in psychology*, vol. 6, p. 368, 2015.

[14] F. Birren, *Color Psychology And Color Therapy; A Factual Study Of The Influence of Color On Human Life.* Hauraki Publishing, 2016.

[15] P. N. und Roswitha Stark, *Heilen mit Symbolen*, 2nd ed. Mankau Verlag GmbH, 2012.

[16] F. Shapiro, *Eye Movement Desensitization and Reprocessing (EMDR): Basic Principles, Protocols, and Procedures, 2nd Edition*, 2nd ed. Guilford Pubn 2001-08-06, 2001.

[17] T. Thompson, D. B. Terhune, C. Oram, J. Sharangparni, R. Rouf, M. Solmi, N. Veronese, and B. Stubbs, "The effectiveness of hypnosis for pain relief: A systematic review and meta-analysis of 85 controlled experimental trials," *Neuroscience & Biobehavioral Reviews*, vol. 99, pp. 298–310, 2019.

[18] C. Schwegler, *Der Hypnotherapeutische Werkzeugkasten*, 2nd ed. Christian Schwegler, 2014.

[19] A. K. Rekkas, *Innovative Hypnotherapie - Wie man ein Krokodil fängt, ohne es zu verletzen*, 3rd ed. Carl-Auer Verlag, 2018.

[20] A. K. Rekkas, *Klinische Hypnose und Hypnotherapie*, 7th ed. Carl-Auer Verlag, 2016.

[21] S. Wang and R. M. Summers, "Machine learning and radiology," *Medical image analysis*, vol. 16, no. 5, pp. 933–951, 2012.

[22] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborová, "Machine learning and the physical sciences," *Rev. Mod. Phys.*, vol. 91, p. 045002, Dec 2019.

[23] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning.* Springer, 2006, vol. 4, no. 4.

[24] O. B. Olivier Sigaud, *Markov Decision Processes in Artificial Intelligence.* Wiley-ISTE, 2010.

[25] C. Szepesvári, *Algorithms for Reinforcement Learning.* Morgan Claypool, 2010.

[26] J. T. Hancock and T. M. Khoshgoftaar, "Survey on categorical data for neural networks," *Journal of Big Data*, vol. 7, no. 1, pp. 1–41, 2020.

[27] C. Seger, "An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing," 2018.

[28] H. Seijen and R. Sutton, "True online td (lambda)," in *International Conference on Machine Learning.* PMLR, 2014, pp. 692–700.

[29] S. Zou, T. Xu, and Y. Liang, "Finite-sample analysis for sarsa with linear function approximation," *Advances in neural information processing systems*, vol. 32, 2019.

[30] G. J. Gordon, "Reinforcement learning with function approximation converges to a region," *Advances in neural information processing systems*, vol. 13, 2000.