Undergraduate/Master's Thesis

Your Title Goes Here

Robin Uhrich

Examiner: Prof. Dr. Joschka Boedecker
Advisers: Jasper Hoffman

University of Freiburg
Faculty of Engineering
Department of Computer Science
Neurobotics Lab

July 27, 2023

Writing Period

05. 07. 2016 – 05. 10. 2016

Examiner

Prof. Dr. Joschka Boedecker

Second Examiner

Prof. Dr. Wile E. Coyote

Advisers

Jasper Hoffman

Bachelor Thesis

Your Title Goes Here

Robin Uhrich

Gutachter: Prof. Dr. Joschka Boedecker
Betreuer: Jasper Hoffman

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik
Lehrstuhl für Neurorobotik

July 27, 2023

Bearbeitungszeit

05. 07. 2016 – 05. 10. 2016

Gutachter

Prof. Dr. Joschka Boedecker

Zweitgutachter

Prof. Dr. Wile E. Coyote

Betreuer

Jasper Hoffman

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

foo bar

Zusammenfassung

German version is only needed for an undergraduate thesis.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This is a template for an undergraduate or master's thesis. The first sections are concerned with the template itself. If this is your first thesis, consider reading Section 1.3.

The structure of this thesis is only an example. Discuss with your adviser what structure fits best for your thesis.

1.1 Template Structure

- To compile the document either run the makefile or run your compiler on the file 'thesis_main.tex'. The included makefile requires latexmk which automatically runs bibtex and recompiles your thesis as often as needed. Also it automatically places all output files (aux, bbl, ...) in the folder 'out'. As the pdf also goes in there, the makefile copies the pdf file to the parent folder. There is also a makefile in the chapters folder, to ensure you can also compile from this directory.

- The file 'setup.tex' includes the packages and defines commands. For more details see Section 1.2.

- Each chapter goes into a separate document, the files can be found in the folder chapters.

- The bib folder contains the .bib files, I'd suggest to create multiple bib files for different topics. If you add some or rename the existing ones, don't forget to also change this in thesis_main.tex. You can then cite as usual [1, 2, 3].

- The template is written in a way that eases the switch from scrbook to book class. So if you're not a fan of KOMA you can just replace the documentclass in the main file. The only thing that needs to be changed in setup.tex is the caption styling, see the comments there.

1.2 setup.tex

Edit setup.tex according to your needs. The file contains two sections, one for package includes, and one for defining commands. At the end of the includes and commands there is a section that can safely be removed if you don't need algorithms or tikz. Also don't forget to adapt the pdf hypersetup!!
setup.tex defines:

- some new commands for remembering to do stuff:
  - `\todo{Do this!}`: **(TODO: Do this!)**
  - `\extend{Write more when new results are out!}`:
    (EXTEND: Write more when new results are out!)
  - `\draft{Hacky text!}`: **(DRAFT: Hacky text!)**

- some commands for referencing, 'in `\chapref{chap:introduction}`' produces 'in Chapter 1'
  - `\chapref{}`
  - `\secref{sec:XY}`

  - `\eqref{}`
  - `\figref{}`
  - `\tabref{}`

- the colors of the Uni's corporate design, accessible with
  `{\color{UniX} Colored Text}`
  - UniBlue
  - UniRed
  - UniGrey

- a command for naming matrices `\mat{G}`, $\mathbf{G}$, and naming vectors `\vec{a}`, $\mathbf{a}$. This overwrites the default behavior of having an arrow over vectors, sticking to the naming conventions normal font for scalars, bold-lowercase for vectors, and bold-uppercase for matrices.

- named equations:

```
\begin{align}
    d(a,b) &= d(b,a)\\ \eqname{symmetry}
\end{align}
```

$$d(a,b) = d(b,a) \tag{1.1}$$

symmetry

1.3 Advice

This section gives some advice how to write a thesis ranging from writing style to formatting. To be sure, ask your advisor about his/her preferences.
For a more complete list we recommend to read Donald Knuth's paper on mathematical writing. (At least the first paragraph). `http://jmlr.csail.mit.edu/reviewing-papers/knuth_mathematical_writing.pdf`

- If you use formulae pay close attention to be consistent throughout the thesis!

- In a thesis you don't write 'In [24] the data is..'. You have more space than in a paper, so write 'AuthorXY et al. prepare the data... [24]'. Also pay attention to the placement: The citation is at the end of the sentence before the full stop with a no-break space. `... last word~\cite{XY}.`

- Pay attention to comma usage, there is a big difference between English and German. '...the fact that bla...' etc.

- Do not write 'don't ', 'can't' etc. Write 'do not', 'can not'.

- If an equation is at the end of a sentence, add a full stop. If it's not the end, add a comma: $a = b + c \quad (1)$,

- Avoid footnotes if possible.

- Use ``'' for citing, not "".

- It's important to look for spelling mistakes in your thesis. There are also tools like aspell that can help you find such mistakes. This is never an excuse not to properly read your thesis again, but it can help. You can find an introduction under `https://git.fachschaft.tf/fachschaft/aspell`.

- If have things like a graph or any other drawings consider using tikz, if you need function graphs or diagrams consider using pgfplots. This has the advantage that the style will be more consistent (same font, formatting options etc.) than when you use some external program.

- Discuss with your advisor whether to use passive voice or not. In most computer science papers passive voice is avoided. It's harder to read, more likely to produce errors, and most of the times less precise. Of course there are situations where the passive voice fits but in scientific papers they are rare. Compare the sentence: 'We created the wheel to solve this.' to 'The wheel was created to solve this', you don't know who did it, making it harder to understand what is your contribution and what is not.

- In tables avoid vertical lines, keep them clean and neat. See **??** for an example. More details can be found in the 'Small Guide to Making Nice Tables' `https://www.inf.ethz.ch/personal/markusp/teaching/guides/guide-tables.pdf`

2 Related Work

LASER: Learning Latent Action Space for Efficient Reinforcement Learning

3 Background

The following part is to provide a short introduction into the theory of reinforcement learning, the Soft Actor Critic Algorithm (SAC) as well to Variational Autoencoders (VAE) and Conditional Variational Autoencoders (CVAE). **(TODO: update!!!)**

3.1 RL Framework

Learning is a topic that is present in all our lives since their beginning. We all learn to move our arms, like an infant that is wiggling around, learn to walk, to speak and more or less any other skill that we find in our personal repository until now.

Reinforcement Learning (RL) is about to have a computational approach to this kind of learning. In the class of RL-Problems the environment provides a challenge like, riding a bicycle. But it is not told how this goal can be achieved. Therefore it has be learned how to derive actions from situations on or off the bike in order to complete this task or - speaking of the numerical approach - maximize a reward function. In most cases the problem solving methods that are classified as RL-methods are applied on tasks that require not only a sequence of actions to interact with an environment but also some kind of feedback. Therefore these setups are designed to provide this kind of feedback. Either if it is an immediate feedback like an additional time step the agent has not crashed the bike or if it is more like a long shot feedback, for example a bonus that the agent has earned after winning board game. These principals can be boiled down to three main characteristics of reinforcement learning problems:

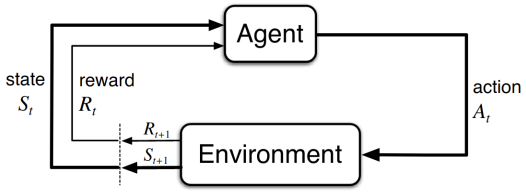


Figure 3.1: interaction between agent and the environment

- There is a closed loop mechanism between actions and feedback that is provided by the environment where the agent is operating in.
- The agent is not given any direct instructions how to solve the given problem.
- The actions an agent is performing have consequences over time. Either for him or the environment he is working in.

Before we can have an exemplary look on the closed loop mechanics, we would like to introduce a couple of key components of RL including the Markov Decision Process.

(TODO: cite figure)

Every agent exists in a specified environment and also interacts within a sequence of discrete time steps $t \in \mathbb{N}$. In order to do so the agent has to perceive the environment through a **state** representation $S_t \in \mathcal{S}$ which is an element of all possible states, the state space $\mathcal{S}$. To really interact with the environment the agent has to perform an **action** $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}(S_t)$ is the set of all possible actions in given state S_t. To chose an action the agent calculates for each possible action A_t a probability $\pi_{\theta,t}(A_t|S_t)$ for a given state S_t and chooses the maximum with respect to the given actions. To achieve a more compact notation we will refer to the action $A_t = a$ and the state $S_t = s$. To avoid confusions with the number $\pi \approx 3.141$ we refer to the policy always as a parameterized policy $\pi_\theta = \pi_{\theta,t}(a|s)$. This probability distribution is called the **policy**. After the agent has chosen and executed an action, which also means the environment will move one time step further $t \to t + 1$, the environment

will return the next state S_{t+1} and the agent will perceive a **reward** $R_{t+1} \in \mathbb{R}$ with $\mathbb{R}$ as the set of all possible rewards.
The long term goal of the agent is to adapt its policy to maximize the total amount reward the agent receives from the environment. These key terminology can also be found in the notation of a Markov Decision Process which is classified as a 4-Tuple of: $(\mathcal{S}, \mathcal{A}, P_a(s, s'), R_a(s, s'))$ with $P_a(s, s')$ as a probability distribution that action a in state state s will lead to the next state $s' = s_{t+1}$ and $R_a(s, s')$ as reward function for transitioning from s to s' with a.

(TODO: Explain the concept of reinforcement learning and its applications Describe the Markov decision process (MDP) and the Bellman equation Discuss the challenges of using reinforcement learning in practice)

(TODO: Es wird gern gesehen: Optimal value function optimal policy / Tradeoff, Sutton Barto. Generalized Policy iteration. Es geht darum einaml ne policy und ne value function zu fnden, Bellman equations)

The overall goal of an agent is to maximize its received cumulative return G_t for one run until time step T. This is metric is denoted in Equation (3.1). In Equation (3.1) γ is the discount rate, $0 \leq \gamma \leq 1$, a parameter to regularize the *importance* of individual received rewards over time.

(TODO: cite sutton barto)

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{3.1}$$

$$= R_{t+1} + \gamma G_{t+1} \tag{3.2}$$

Unfortunately the individual rewards R_t are highly dependent on state action pairs in the future, so they are not accessible at the current time step. To solve this problem

reinforcement algorithms are trying to maximize the expected return $v_{\pi_\theta}(s)$ as in Equation (3.3), or value function, for a given state. For Markov decision process we can define the value function as in Equation (3.4).

(TODO: cite sutton barto)

$$v_{\pi_\theta}(s) \doteq \mathbb{E}_{\pi_\theta}[G_t|S_t = s] \tag{3.3}$$

$$\begin{aligned}
&\overset{(3.2)}{=} \mathbb{E}_{\pi_\theta}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi_\theta(a|s) \sum_{s'} \sum_r P(s', r|s, a) \left[r + \gamma \mathbb{E}_{\pi_\theta}[G_{t+1}|S_{t+1} = s']\right] \\
&= \sum_a \pi_\theta(a|s) \sum_{s',r} P(s', r|s, a) \left[r + \gamma v_{\pi_\theta}(s'm)\right] \qquad (3.4)
\end{aligned}$$

For a finite Markov decision process we can also find a the optimal value function $v_*(s)$ with respect to the optimal policy in Equation (3.7).

$$q_{\pi_\theta}(s, a) = \sum_{s',r} P(s', r|s, a) \left[r + \gamma \sum_{a'} \pi_\theta(a'|s') q_{\pi_\theta}(s', a')\right] \tag{3.5}$$

$$v_*(s) \doteq \max_{\pi_\theta} v_{\pi_\theta}(s) \tag{3.6}$$

$$= \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma v_*(s')] \tag{3.7}$$

Because Equation (3.6) holds this implies the reinforcement learning problem in Equation (3.8) for the optimal policy π^*_θ and a performance measure $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[G_t|S_t = s]$

as following:

$$\pi_\theta^* = \arg\max_{\pi_\theta} J(\pi_\theta) \tag{3.8}$$

$$= \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}[G_t | S_t = s] \tag{3.9}$$

$$\overset{(3.3)}{=} \arg\max_{\pi_\theta} v_{\pi_\theta}(s) \tag{3.10}$$

$$q_*(s,a) = \sum_{s',r} P(s',r|s,a) \left[r + \gamma \max_{a'} q_*(s',a') \right] \tag{3.11}$$

3.2 Generalized Policy Iteration

This section is about to provide an introduction to the idea of policy iteration. For a more detailed look into the topic it is recommended refer to the Reinforcement Learning Textbook from Richard Sutton and Andrew Barto.

One fundamental concept for reinforcement learning algorithms is policy iteration. At Its core it is described as the alternating interaction between a policy improvement step and a policy evaluation step as in Figure 3.2. This interaction goes back and forth during the execution of most reinforcement learning algorithms until a state of convergence in the value function and policy has arrived. That means both value function and policy are optimal for the given problem. This state can be reached if the "value function is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function". This can be also seen in the Bellman optimally equation as in Equation (3.7).

We can also think about this process as a tradeoff between improving the policy towards an optimal greedy behavior which makes the value function incorrect (blue arrows indicate a policy improvement step) and adapting the value function with respect to the current policy which makes the policy automatically less optimal (red

Figure 3.2: schematic drawing of a conditional Variational Autoencoder

Figure 3.3: schematic drawing of a conditional Variational Autoencoder

arrows indicate a policy evaluation step). **(TODO: Why?)** Despite this tradeoff the algorithm tends to find an optimal solution for the policy and the value function in the long run. **(TODO: Where is the proof)**

3.3 Soft Actor Critic

Soft Actor-Critic was introduced by **(TODO: cite paper)** in **(TODO: year)**. It is an algorithm belonging to the family of model free, off-policy, actor-critic reinforcement learning algorithms. The family of actor-critic algorithms addresses challenges in

exploration and sample efficiency. Similar to other actor-critic algorithms it also employs the alternating concept of a policy improvement step and a policy evaluation step. In contrast to other actor-critic algorithms like XXXX **(TODO: fill)** it uses the concept of entropy regularization and uses a stochastic policy for promoting exploration in the reinforcement learning environment. By jointly optimizing the policy and value functions using the maximum entropy objective enabled by entropy regularized, SAC effectively explores the environment while seeking to maximize rewards. This approach results in robust and adaptive policies that can efficiently handle various RL tasks. **(TODO: examples by paper)**

In this section will introduce the concept of actor-critic algorithms, provide a detailed description of the SAC algorithm architecture, including entropy regularization and explain how to train the algorithm under the maximum entropy objective.

3.3.1 Actor-Critic Algorithms

The basic actor-critic architecture is a type of reinforcement learning algorithm that consists of two components: an actor and a critic. The actor also referred to as the policy π_θ is responsible for selecting actions based on the current state, while the critic also referred to as a value function v_{π_θ} or a state value function q_{π_θ}, evaluates the quality of the actor's actions by estimating the expected return. Like as introduced in Section 3.2 the actor uses the feedback from the critic to adjust its policy and improve its performance.

(TODO: Maybe in RL Formulation or a separated section?) One additional factor to distinguish between reinforcement learning algorithms and also actor-critic algorithms is by either on-policy learning or off-policy learning. On-policy learning or off-policy learning refer to different methods for updating the policy in reinforcement learning. In on-policy learning, the agent learns from the data generated by its current policy, while in off-policy learning, the agent learns from data generated by a

different policy. On-policy learning can be more stable, but it may require more data to converge. Off-policy learning can be more efficient, but it can be more sensitive to the quality of the data. **(TODO: cite sutton barto)**

The advantage of using a critic in the actor-critic algorithm is: it provides a more stable feedback signal than using only rewards. The critic estimates the expected return from a state, for the value function, or a state action pair for the action-value function, which takes into account the long-term consequences of actions. This allows the actor to learn from the critic's feedback and improve its performance more efficiently than if it only received reward signals. Additionally, the critic can help to generalize across different states and actions, improving the overall performance of the algorithm.

One limitation of the basic actor-critic architecture is that it can suffer from high variance and slow convergence due to the interaction between the actor and critic. This can be addressed through the use of techniques such as baseline subtraction and eligibility traces. **(TODO: look for paper) (TODO: cite sutton barto)**

Popular examples for actor critic algorithms are:

- Deep Deterministic Policy Gradient
- Soft Actor-Critic
- Proximal Policy optimization

3.3.2 Entropy Regularization

Entropy regularization is a policy regularization technique by incorporating the policy entropy, used to encourage the policy to explore a diverse range of actions during training. In SAC, entropy regularization is achieved by adding the entropy $H(\pi_\theta)$ as in Equation (3.12) to the policy objective function in Equation (3.27).

$$
\begin{aligned}
H(\pi_\theta) &= \mathop{\mathbb{E}}_{s\sim\mathcal{S},a\sim\mathcal{A}}[-\log(\pi_\theta(a|s))] \\
&= -\log(\pi_\theta(a|s))
\end{aligned} \tag{3.12}
$$

(TODO: is this correct?) (TODO: something has to be cleared up with $H(\pi_\theta)! = H(\pi_\theta(\cdot|s))$)

Including entropy into the objective function encourages the policy to generate actions with higher entropy (higher randomness), leading to more exploration, possibly accelerate training and preventing converging to a poor local optimum. **(TODO: cite: spinning up open ai)**

The randomness or uncertainty of a policy for a given state action pair $\pi_\theta(a|s)$ can be computed by seeing $\pi_\theta(a|s)$ as a density function and taking $\pi_\theta(\cdot|s) = \mathcal{N}(\mu(\pi_\theta(\cdot|s)), \sigma(\pi_\theta(\cdot|s)))$ as a parameterized normal distribution from which the action $a \sim \pi_\theta(\dot{|}s)$ is sampled. Therefor it possible to acquire the probability $\pi_\theta(a|s)$ of an action a with as in Equation (3.13).

$$
\pi_\theta(a|s) = \frac{1}{\sigma(\pi_\theta(\cdot|s))\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(a-\mu(\pi_\theta(\cdot|s)))^2}{\sigma(\pi_\theta(\cdot|s))^2}\right) \tag{3.13}
$$

As a result of entropy regularization in each time step both value function v_{π_θ} and action-valuefunction q_{π_θ} become affected and turn into their counterpart $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$ in Equation (3.14) and Equation (3.15) below. Note that Equation (3.16) is the recursive version of Equation (3.15).

$$v_{\pi_\theta,H}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t\left(R_t + \alpha H(\pi_\theta(\cdot|s_t))\right) \mid s_0 = s\right] \tag{3.14}$$

$$q_{\pi_\theta,H}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\gamma^t R_t + \alpha\sum_{t=1}^{T}\gamma^t H(\pi_\theta(\cdot|s_t)) \mid s_0 = s, a_0 = a\right] \tag{3.15}$$

$$= \mathop{\mathbb{E}}_{s'\sim P, a'\sim\pi_\theta}\left[R_t + \gamma(q_{\pi_\theta,H}(s',a') + \alpha H(\pi_\theta(\cdot|s')))\right] \tag{3.16}$$

(TODO: include entropy in the first time step -> look at the code and search for the appropriate paper)

This changes subsequently also the original relation between v_{π_θ} and q_{π_θ}. $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$ are connected vai Equation (3.17).

$$v_{\pi_\theta,H}(s) = \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a)\right] + \alpha H(\pi_\theta) \tag{3.17}$$

(TODO: proof in appendix)

Note that we introduce a new parameter α into $v_{\pi_\theta,H}$ and $q_{\pi_\theta,H}$. This parameter controls the tradeoff between exploration and exploitation. Higher values of α promote more exploration, whereas lower values of α encourage more exploitation in the Soft Actor-Critic algorithm. For more details how to this parameter is used in Soft-Actor-Critic please have look into Section 3.3.3

For simplification we will now refer to the entropy regularized value function $v_{\pi_\theta,H}$ as v_{π_θ} and action-value-function $q_{\pi_\theta,H}$ as q_{π_θ}.

This change also influences the original reinforcement learning problem as stated in Equation (3.10) and converts it into Equation (3.18). Due to the stated relation between regularized value-function and regularized action-value-function as in Equation (3.17) we can make the optimization problem independent from the value-function and get Equation (3.19).

$$\pi_\theta^* = \arg\max_{\pi_\theta} v_{\pi_\theta, H}$$

$$\overset{(3.10)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t \left(R_t + \alpha H(\pi_\theta)\right) \mid S_t = s\right] \tag{3.18}$$

$$\overset{(3.17)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a)\right] + \alpha H(\pi_\theta)$$

$$\overset{(3.12)}{=} \arg\max_{\pi_\theta} \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)\right] \tag{3.19}$$

(TODO: was ist allgemeiner .... bis inf oder bis T)

3.3.3 Algorithm Architecture

In contrast to other actor-critic algorithms like: XXX, or XXX, **(TODO: add examples)** SAC learns additional to a parameterized policy π_θ, two critics in the form of two parametrized q functions Q_{ϕ_0} and Q_{ϕ_1} as well as their corresponding target functions $Q_{\phi_{\text{target},1}}$ and $Q_{\phi_{\text{target},1}}$. The usage of target action-value functions enhances the accuracy of the estimates and contributes to a more stable learning process. **(TODO: proof)**

Further SAC utilizes a stochastic policy, which means that the policy outputs a parameterized probability distribution over actions instead of directly selecting deterministic actions and optionally adding noise on top. This stochastic nature allows SAC to capture the exploration-exploitation trade-off directly within the policy. The training process in SAC is based on the maximum entropy objective. Its goal is to find a policy that maximizes the cumulative reward Equation (3.1) and the maximum entropy Equation (3.12). Fortunately we have derived an entropy regularized value-function and action-value-function in Section 3.3.2 and are able to use those definition to derive loss functions for actor and critic based on the maximum entropy objective.

Because we will have a closer look into actual implementation, we now refer to the action-value-function as their parameterized approximations Q_{π_i}.

To **train the parameterized action-value-functions** the objective function (3.20) is defined as the expected squared difference between the action-state-value $Q_{\phi_i}(s,a)$ and its temporal difference target $y(r,s',d)$. Equation (3.20) leads directly to the update equation (3.26) as its derived empirical counterpart.

$$L_Q(\phi_i, \mathcal{D}) = \mathbb{E}_{\mathcal{D}}\left[\left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2\right] \tag{3.20}$$

The temporal difference target target y is defined as in Equation (3.21) with $\hat{a}' \sim \pi\theta(\cdot|s')$. **(TODO: drop the buzz word: td target)**. To stabilize the target signal SAC applies the clipped double-Q trick, where it selects the minimum q-value between the two target q-functions $Q_{\phi_{\text{target},i}}$.

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=0,1} Q_{\phi_{\text{target},i}}(s',\hat{a}') - \alpha \log \pi\theta(\hat{a}'|s')\right) \tag{3.21}$$

(TODO: where exactly is the target coming from?)

This function is employed in the SAC algorithm in Equation (3.25) but with the small difference that all arguments are sampled from a minibatch $\mathcal{B}$.

To **train the policy** we are required to use the reparameterization trick to be able to differenciate the parameterized policy π_θ. **(TODO: cite: https://sassafras13.github.io/ReparamT**
This process transforms the action $\hat{a}$ sampled from the policy into the function as specified in Equation (3.22).

$$\hat{a}_\theta(s,\epsilon) = \tanh(\mu(\pi_\theta(\cdot|s)) + \sigma(\pi_\theta(\cdot|s)) \bigodot \epsilon), \;\; \epsilon \sim \mathcal{N}(0,I) \tag{3.22}$$

Within the reparameterization of an action we also bound it into a finite range of $(-1, 1)$ this has the advantage , ... **(TODO: fill)**. Since we squash the actions with tanh we also have to adapt the log-likelihood $\log \pi_\theta(a|s)$ of an action into:

$$\log \pi_\theta(a|s) = \log \pi_\theta(a|s) - \sum_{i=1}^{|\mathcal{A}|} \log(1 - \tanh(\hat{a}_\theta(s, \epsilon))), \ \epsilon \sim \mathcal{N}(0, I)$$

For more details please refer to **(TODO: cite: https://arxiv.org/pdf/1812.05905.pdf)**.

To compute the policy loss, as in training the action-value-function approximation, the crucial step involves replacing q_{π_θ} with one of our function approximators Q_{ϕ_i}. SAC utilizes the minimum of the two q_{π_θ} approximators $\min_{i=0,1} Q_{\phi_i}$. Consequently, the policy is optimized based on this minimum action-state-value approximation and therefor make only more conservative estimates.

$$L_\pi(\theta, \mathcal{D}) = -\mathbb{E}_{\mathcal{D}, \mathcal{N}(0,I)} \left[\min_{i=0,1} Q_{\phi_i}(s, \hat{a}_\theta(s, \epsilon) - \alpha \log \pi_\theta(\hat{a}_\theta(s, \epsilon)|s)) \right] \tag{3.23}$$

Note that we are taking the negative expectation because we want to maximize the expected return and the entropy by using SGD as in Equation (3.27) in Algorithm 2. The different notation can be explained a $\{s, a, r, s', d\}_{\mathcal{B},k}$ stand for the kth element from the minibatch $\mathcal{B}$..

Now lets turn to the entropy regularization parameter α. In general there are two types of Soft-Actor-Critic implementations. On proposed by XXXX **(TODO: fill)** treads α as constant parameter. The optimal α parameter, leading to the most stable and rewarding learning, may vary across different environments, necessitating thoughtful tuning for optimal performance. The second approach how to treat α, proposed by XXX **(TODO: fill)**, adjusts α constantly during the training process. The optimization criterion is stated as:

$$L_\alpha = \mathop{\mathbb{E}}_{a_t \sim \pi_\theta} \left[-\alpha \log \pi_(a_t|s_t) - \alpha \bar{H} \right] \tag{3.24}$$

Equation (3.24) resembles an objective for dual gradient descent because we are trying to minimize α but also the expected difference between the policy entropy and a target entropy $\bar{H}$ as a hyperparameter. In practice this can be translated into a positive gradient if the expected difference is positive so the agent is less exploring as should be and into a negative gradient if the difference is negative so the agent is to focused on exploring. Selecting a target entropy is not as delicate as selecting a fixed α because you are able to read from training results if your environment requires a more or less greedy policy. **(TODO: dual gradient descent chapter.)**

(TODO: rework equations from pseudo code)

3.3.4 Advantages

SAC offers several advantages over other actor-critic algorithms. Firstly, the entropy regularization leads to improved exploration, enabling the agent to efficiently explore its environment and discover optimal or near-optimal solutions **(TODO: paper)**. Secondly, by encouraging stochastic policies, SAC provides more robust and adaptive policies that can handle uncertainties and variations in the environment.

Moreover, the SAC algorithm exhibits enhanced sample efficiency, meaning that it requires fewer interactions with the environment to learn effective policies. The utilization of two critics Q_{ϕ_0} and Q_{ϕ_1} further contributes to a more stable learning process and can mitigate the issues of overestimation bias, leading to more accurate value estimates. **(TODO: paper)**

Algorithm 1 Soft Actor Critic

Input: initial policy parameters θ, Q-function parameters ϕ_0, ϕ_1, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\phi_{\text{target},0} \leftarrow \phi_0$, $\phi_{\text{target},1} \leftarrow \phi_1$

for i in number of epochs **do**

  $s \leftarrow$ reset environment

  for t number of timesteps **do**

    $a \sim \pi_\theta(\cdot|s)$

    s', r, $d \leftarrow$ execute a in environment

    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

    if d is true **then**

      break and reset environment

    end if

    $s \leftarrow s'$

  end for

  if $|\mathcal{D}| >$ minimal buffer size **then**

    for number of train iterations **do**

      sample minibatch: $(s_\mathcal{B}, a_\mathcal{B}, r_\mathcal{B}, s'_\mathcal{B}, d_\mathcal{B}) = \mathcal{B} \leftarrow \mathcal{D}$

      compute td target y with $\tilde{a}'_\mathcal{B} \sim \pi_\theta(\cdot|s'_\mathcal{B})$

$$y(r_\mathcal{B}, s'_\mathcal{B}, d_\mathcal{B}) = r + \gamma \cdot d \cdot \left(\min_{i \in \{0,1\}} \left(Q_{\phi_{\text{target},i}}(s'_\mathcal{B}, \tilde{a}'_\mathcal{B}) \right) - \alpha \cdot \log \left(\pi_\theta(\tilde{a}'_\mathcal{B}|s'_\mathcal{B}) \right) \right) \tag{3.25}$$

      Update Q-functions for parameters ϕ_i $i \in \{0, 1\}$ using:

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{k \in |\mathcal{B}|} \mathcal{L}_\beta \left(y(r_{\mathcal{B},k}, s'_{\mathcal{B},k}, d_{\mathcal{B},k}), Q_{\phi_i}(s_{\mathcal{B},k}, a_{\mathcal{B},k}) \right) \tag{3.26}$$

      Update policy with $\tilde{a}_\mathcal{B} \sim \pi_\theta(\cdot|s_\mathcal{B})$ using:

$$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{k \in |\mathcal{B}|} \min_{i \in \{0,1\}} Q_{\phi_i}(s_{\mathcal{B},k}, \tilde{a}_{\mathcal{B},k}) - \alpha \cdot \log \left(\pi_\theta(\tilde{a}_{\mathcal{B},k}|s_{\mathcal{B},k}) \right) \tag{3.27}$$

      Update α with target entropy H_{target} and $\tilde{a}_\mathcal{B} \sim \pi_\theta(\cdot|s_\mathcal{B})$ using:

$$\nabla_\alpha - \frac{\alpha}{|\mathcal{B}|} \sum_{k \in |\mathcal{B}|} \left(\log \left(\pi_\theta(a'_{\mathcal{B},k}|s_{\mathcal{B},k}) \right) \ - H_{\text{target}} \right)$$

      Update target networks $\phi_{\text{target},i}$ $i \in \{0, 1\}$ with ϕ_i $i \in \{0, 1\}$ using:

$$\phi_{\text{target},i} \leftarrow \rho \phi_{\text{target},i} + (1 - \rho) \phi_i$$

    end for

  end if

end for

Applications

In robotics, the SAC algorithm has been used to control robots in tasks such as grasping objects, locomotion, and manipulation. The SAC algorithm's ability to handle continuous action spaces makes it a suitable choice for robotic control, where fine-grained control is often required. **(TODO: look for papers to support this claim)**

In game playing, the SAC algorithm has been used to train agents to play video games such as Atari and Super Mario Bros. **(TODO: paper)** The SAC algorithm's ability to balance exploration and exploitation makes it effective in game playing, where agents must learn to navigate complex environments and respond to changing conditions.

3.4 Neural Networks

Neural networks are a class of artificial intelligence algorithms inspired by the structure and functioning of the human brain. They consist of interconnected layers of artificial neurons that process and transform data. Neural networks are crucial in modern machine learning and deep learning applications due to their ability to learn complex patterns and representations from data, enabling them to solve a wide range of problems effectively.

In this chapter, we will delve into neural networks, looking at their evolution, architecture, training process and diverse applications.

3.4.1 The Rise of Deep Learning

Neural networks were first invented in the 1940s and 1950s as a computational model inspired by the interconnected structure and functioning of the human brain, but

their practical development was hindered by limitations in computing power and the lack of large datasets. The resurgence of interest in neural networks in the 2000s can be attributed to two major factors: the development of new algorithms and the availability of large datasets. Before the 2000s, neural networks faced limitations in training and optimization, hindering their effectiveness. However, new algorithms, such as the backpropagation algorithm and variants like stochastic gradient descent, emerged, making it feasible to train deeper networks efficiently. Additionally, advancements in computing power and the availability of vast datasets facilitated by the internet like image net **(TODO: cite)** allowed neural networks to leverage big data for improved learning and performance.

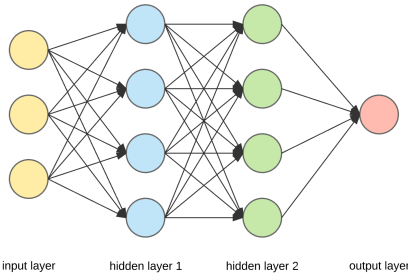
Nowadays neural networks are employed over a vast variety of tasks like image recognition, speech recognition, natural language processing or robotics.

3.4.2 Neural Network Architecture

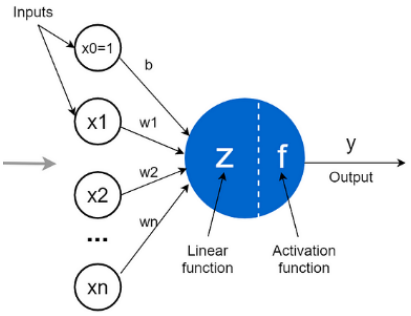
Neural networks are a type of machine learning model inspired by the structure and function by the cell type of neurons. It consists like their biological model, of interconnected layers of individual units, called neurons.

The basic architecture of a neural network as in Figure 3.4a, includes an input layer (yellow), one or more hidden layers (blue and green), and an output layer (red). The input layer receives the input data, which is then passed through the hidden layers before producing the output. Because the flow of information is only in the forward direction we call this basic architecture a feed forward neural network. The number of nodes in each layer and the connections between them are configured by the architecture of a network.

If we look at each individual neuron as in Figure 3.4b, we can observe that each neuron is designed in the same way. First it calculates a weighted sum z of its inputs x, the corresponding weights w and bias b, before passing it into an activation



(a) schematic drawing of a feed forward neural network



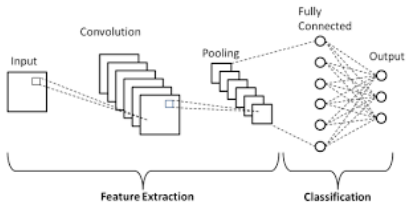
(b) schematic drawing of a single neuron

Figure 3.4: The figure shows the schematic drawing of a feed forward neural network and a neuron

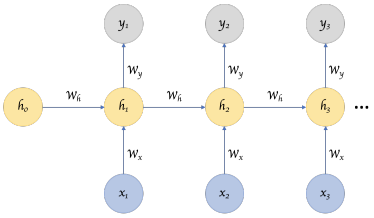
(a) Sigmoid **(b)** hyperbolic tangent function **(c)** rectified linear unit

Figure 3.5: common activation functions in a neural network

function h and finally passing the information to the next neuron. The activation function is designed to introduce nonlinearity into the model, allowing it to capture complex relationships between variables. Without a nonlinear activation function the whole network would be a single linear combination of its inputs and weights. Common activation functions include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function as in Figure 3.5



(a) schematic drawing of convolutional neural network



(b) schematic drawing of a recurrent neural network

Figure 3.6: architecture of more advanced neural networks like the convolutional and the recurrent neural network

There are several types of neural network architectures but most of them are based on following types:

- **Feedforward networks** as in Figure 3.4a are the simplest type of neural network, consisting of a series of layers that process information in a single direction.

- **Convolutional networks** as in Figure 3.6a are designed for image processing tasks and use convolutional layers to identify patterns and features within images.

- **Recurrent networks** as in Figure 3.6ballow information to be passed between nodes in a cyclical manner, making them suitable for processing sequential data. The development of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures further improved RNNs' ability to model long-range dependencies.

3.4.3 Train Neural Networks

Neural networks are trained using techniques such as backpropagation and stochastic gradient descent as presented in Algorithm 2. Backpropagation is an algorithm for calculating the gradient $\hat{g}$ of an optimization criterion with respect to the weights of the network. The gradient can then be used to update the weights and therefor improve the performance of the model with respect to the opimization function. A we can see in Algorithm 2 stochastic gradient descent minimizes the error presented by the opimization criterion by iteratively adjusting the weights based on randomly selected subsets of the training data.

Algorithm 2 Stochastic gradient descent

Input: Learning rate schedule: $\epsilon_1, \epsilon_2, \ldots$. Initial parameter θ

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples from training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

Compute gradient estimate:

$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Apply update:

$\theta \leftarrow \theta - \epsilon_k \hat{g}$

$k \leftarrow k + 1$

end while

Selecting a random subset of training data has a couple of advantages.

- **Computational Efficiency**: In one epoch we are computing the gradient only on a small subset. We do not have to pass the complete dataset through the network to get a sufficient gradient.

- **Improved Generalization**: Because each minibatch contains a different composition of data points from the dataset the gradient signal is also varying. These variations are improving the generalization of the model with respect to unseen data.

- **Avoiding Local Minima**: One of the risks during the training-process of a neural network are local minima. By introducing randomness through mini-batch sampling, SGD can escape local minima more easily and continue to explore the parameter space, increasing the chances of finding a global minimum.

- **Parallel Processing**: Using mini-batches allows parallel processing during training.

(TODO: find sources)

3.4.4 Applications of Neural Networks

Neural networks are widely used in real-world applications like in computer vision, marketing or medical applications. Particularly in combination with reinforcement learning they are successfully used in robotics for object recognition and manipulation, in gaming for complex decision-making, and in finance for predicting stock prices.

3.5 Variational Autoencoder

Variational autoencoders (VAEs) and conditional variational autoencoders (CVAEs) are types of deep generative models that are used for unsupervised learning. Unsupervised learning is a type of machine learning where the model is not given labeled data for training. Instead, the model is tasked with finding patterns or structure in the data on its own. The goal of unsupervised learning is often to find hidden relationships or groupings within the data that can be used for further analysis or decision-making VAEs and CVAEs are important because they can learn to generate realistic and diverse samples from complex high-dimensional data distributions, such as images or audio.

3.5.1 Autoencoders

Lets start with a Autoencoder. An Autoencoder $f = d \circ e$ consists of two parameterized function approximators in series, one encoder e_ϕ and one decoder d_ψ. The encoder maps the high dimensional input $x \in \mathbb{X}$ into a lower dimensional latent space $z \in \mathbb{Z}$, $z = e(x)$. The latent vector z is typically a lower dimensional representation of the input x. In the second stage we map the latent vector z back into the high dimensional features space X, $d : \mathbb{Z} \to \mathbb{X}$. This should optimally reconstruct the given input $\hat{x} = d(z) = d(e(x))$.

Because Autoenders are designed to learn a compact determin representation in the latent space it follows that similar input values x should correspond to similar latent vectors z and similar latent vectors z correspond to similar outputs $\hat{x}$.

Traditional Autoenders have prooven to be successful on a variaty of task like: ..., but their simple and effective design comes along with some limitations:

- Overfitting: Because of the deterministic mapping into the latent space and from the latent space back to the feature space traditional Autoencoders are

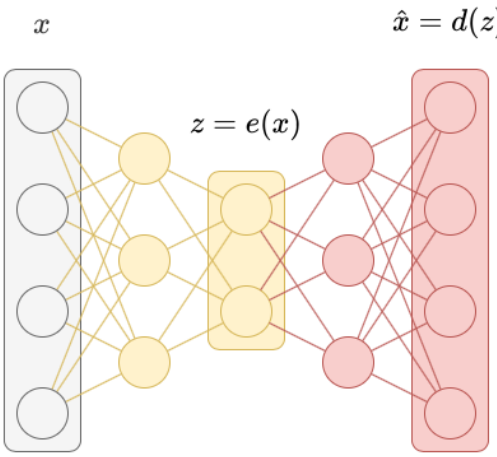


Figure 3.7: schematic drawing of a Autoencoder

prone to suffer from overfitting and can lack of regularization which could lead to poor performance on unseen data.

- Incomplete or noisy data: Due to the focus on only reconstructing the input, Autoencoders are not well suited for incomplete or noisy data. One reason could be the lack of disentanglement in the latent space, which means that different dimensions correspond to different underlying features in the input data distribution.

- Probabilistic Interpretation: Another drawback of the deterministic mapping approach is the lack of probabilistic interpretation capabilities for uncertainty in the learned representation and generated outputs.

3.5.2 Variational Autoencoders

(TODO: ELBO: https://probml.github.io/pml-book/book2.html 779)

A Variational Autoencoder is similar to an Autoencoder but also with some key changes. Similar are the basic setup as a generative unsupervised learning model and the underlying encoder decoder structure. The key difference lays in the in the latent

space between the encoder and decoder. Instead of having a fixed deterministic latent space VAEs operate on a probabilistic latent space.

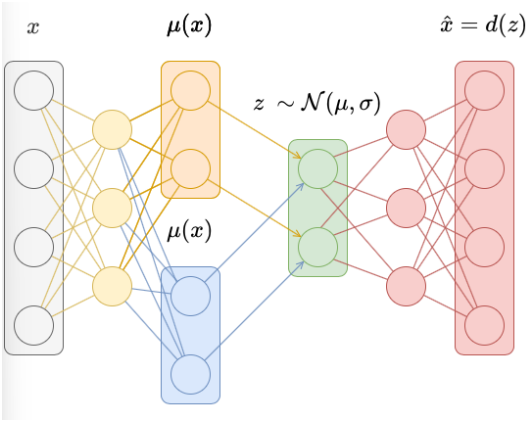


Figure 3.8: schematic drawing of a Variational Autoencoder

The encoder network denoted as a approximated parameterized posterior distribution $q_\phi(z|x)$, is a conditional probability distribution in Bayesian inference with the probability of the latent variable $p(z)$ (the prior) and the evidence $p(x)$:

$$q(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{3.28}$$

$$p(x) = \int p(x|z)p(z)dz \tag{3.29}$$

In the context of a Variational Autoencoder, calculating $q(z|x)$ is not possible. As we can see in Equation (3.29) accessing the evidence as the probability of the observed data is not possible because we have to integrate over all possible values of the latent variable z. Therefor we approximate the posterior distribution with a parameterized conditional distribution $q_\phi(z|x)$. $q_\phi(z|x)$ as known as the encoder network, parameterized with ϕ, hereby maps input data x to latent parameters like mean and variance for a gaussian distribution, of the latent distribution with random variable z.

On the other hand, the decoder network takes samples from the latent space with latent variables z and reconstructs the data from these samples, generating a parameterized probabilistic distribution over the data given the latent variables $p_\psi(x|z)$.

As mentioned before the objective in training a Variational Autoencoder is to find parameters ϕ and ψ and matches best the true posterior distribution. In order to do so we can use the Evidence Lower Bound as the objective function. We go a bit deeper into the fundamentals of the Evidence Lower bound in Section 3.5.4.
During training, the VAE aims to maximize the ELBO by iteratively adjusting its parameters ϕ andψ using backpropagation. The encoder network maps the input data to a distribution over latent variables, while the decoder network reconstructs the data from the sampled latent variables. The reparameterization trick similar to

SAC in Section 3.3 is employed to ensure differentiability during backpropagation through the stochastic sampling process. **(TODO: reparaphrase)**

3.5.3 Conditional Variational Autoencoders

Figure 3.9: schematic drawing of a conditional Variational Autoencoder

A Conditional Variational Autoencoder is an extension of the VAE architecture that takes into account conditional information c. As we can see in Figure 3.9In a Conditional Variational Autoencoder, both the encoder and decoder are modified to accept an additional input that represents the conditional information.

The role of the conditional input in the encoder is to help the model capture the conditional dependencies between the input data and the class label. So the conditional information could be for instance a label encoding or a set of labels that describe the class or category to which the input belongs. The encoder must learn to map both the input data and the conditional input to a meaningful latent space representation that captures the relevant information for the generation process. In the decoder, the conditional input is used to guide the generation process and ensure that the generated samples are representative of the specified class.

Similar to Variational Autoencoders the objective function to train the encoder and decoder is the Evidence Lower Bound. But there have to be a couple of changes to make the individual ELBO components suitable to take the conditional information c. The final objective can be observed in Equation (3.33).

3.5.4 Evidence Lower Bound

The Evidence Lower Bound (ELBO) is a fundamental concept in the training of Variational Autoencoders (VAEs) It is derived from the principle of variational inference and represents a lower bound on the log-likelihood of the data given the model's parameters. Maximizing the ELBO is equivalent to minimizing the Kullback-Leibler (KL) divergence between the true posterior distribution $q(z|x)$ over latent variables z and the approximated posterior distribution $q_\phi(z|x)$ used in the VAE.

$$L_{\text{ELBO}}(x, z) = \mathbb{E}_{z \sim q_\phi(\cdot|x)} \left[\log p\psi(x|z)\right] - \text{KL}\left(q_\phi(z|x) \| p(z)\right) \tag{3.30}$$

Further I will discuss the individual components of the Evidence lower bound

- **Posterior Distribution**: In general the posterior distribution $q(z|x)$ is considered as the probability of the latent variable z given the observed data x. Specific for Variational Autoencoder it can be computed from Bayesian Inference as in Equation (3.28) which is as mentioned in Section 3.5.2 not feasible is therefor approximated.

- **Likelihood**: The likelihood function $p(x|z)$ or conditional likelihood in the context of probabilistic models is the probability distribution of the observed data x given the latent variables z. Dependent on the observed data it is possible to chose from a set of different probability distributions. Examples are a multivariate gaussian distribution for continuous data, a Bernoulli distribution for binary data or categorical distribution for discrete data with multiple categories.

- **Prior Distribution**: The prior distribution $p(z)$ is a probabilistic distribution that represents the initial belief or assumption about the latent variables z in Bayesian inference. Additional $p(z)$ serves in Variational Autoencoder as a regularizer during training as it encourages the VAE to learn meaningful and smooth latent representations in the latent space. Most popular choice for the prior distribution is the standard normal distribution $(N)(0, I)$.

- **Evidence**: p(x) represents the evidence, also known as the marginal likelihood or model evidence, in the context of Bayesian statistics. It is the probability of the observed data x given a particular statistical model. The evidence serves in Equation (3.29) as a normalization constant, ensuring that the posterior distribution $q(z|x)$ over model parameters integrates to 1. It quantifies how well the model, with its specific set of parameters, explains or fits the observed data.

- **Reconstrion Loss**:

$$\mathbb{E}_{z\sim q(\cdot|x)}\left[\log p(x|z)\right] \approx \mathbb{E}_{z\sim q_\phi(\cdot|x)}\left[\log p_\psi(x|z)\right] \tag{3.31}$$

Therm (3.31) measures the similarity between the reconstructed data and the original input. Described in words it is the expected log-likelihood of the data given the latent variables, which is computed by sampling from the approximating distribution $q_\phi(z|x)$ and evaluating the likelihood $p_\psi(x|z)$ using the decoder network.

- **KL Divergence Loss**:

$$\mathrm{KL}\left(q(z|x)\|p(z)\right) \approx \mathrm{KL}\left(q_\phi(z|x)\|p(z)\right) \tag{3.32}$$

This term quantifies the difference between the approximated parameterized posterior distribution $q_\psi(z|x)$ over latent variables z and a chosen prior distribution $p(z)$. As mentioned before it is a popular choice to use a standard normal distribution $p(z) = (N)(0, I)$ as the prior distribution. The KL divergence encourages the latent variables to be close to the prior distribution, promoting regularization and preventing overfitting.

(TODO: Explain prior and posterior distributions)

The basic notation of the Evidence Lower Bound for Variational Autoencoders as in Equation (3.30) can be easily extented for an Conditional Variational Autoencoder:

$$L_{\mathrm{ELBO}}(x, z, c) = \mathbb{E}_{z\sim q_\phi(\cdot|x,c)}\left[\log p\psi(x|z, c)\right] - \mathrm{KL}\left(q_\phi(z|x, c)\|p(z|c)\right) \tag{3.33}$$

(TODO: Describe also the components of the CVAE ELBO)

By optimizing the ELBO, VAEs effectively learn to approximate the true posterior distribution and generate meaningful latent representations of the data, enabling

various tasks such as data generation, interpolation, and denoising within a Bayesian framework.

3.5.5 VAEs and CVAEs in Practice

VAEs and CVAEs have been successfully applied to a wide range of real-world applications. In image generation, VAEs have been used to generate novel images of faces, objects, and scenes. Similarly, CVAEs have been used for conditional image generation, allowing for the generation of images based on specific attributes or classes. In text generation, VAEs have been used to generate natural language sentences and paragraphs.

VAEs and CVAEs can also be used for data compression and denoising. By learning a compressed representation of the input data, VAEs and CVAEs can reduce the dimensionality of the input space while preserving important features. Similarly, by learning to reconstruct the original input from noisy or corrupted data, VAEs and CVAEs can be used for denoising and data restoration.

One of the advantages of VAEs and CVAEs is their ability to learn a continuous latent representation of the input data. This allows for easy manipulation and exploration of the latent space, enabling applications such as image editing and style transfer **(TODO: cite paper)**.

However, VAEs and CVAEs have some limitations. The generated samples may not be as sharp or detailed as those produced by other generative models such as GANs **(TODO: cite)**. Additionally, the trade-off between the reconstruction loss and the KL divergence term can be difficult to balance, potentially leading to overfitting or underfitting **(TODO: paper)**. Nonetheless, VAEs and CVAEs remain a popular and powerful tool for generative modeling and data compression.

3.6 Kinematics

Kinematics is the study of motion, specifically the description and analysis of the position, velocity, and acceleration of objects or systems without considering the forces causing the motion. It focuses on understanding the spatial relationships and geometrical aspects of moving objects. The following sections provide an insight into forward and inverse kinematics for kinematic chains. Those two concepts are often used in robotics, animation, virtual reality or even protein folding.

3.6.1 Forward kinematics

Forward kinematics is a concept from robotics that involves determining the position and orientation of an end-effector, like a gripper of a robot arm, based on the joint angles and geometric parameters, like segment length, of the system. It provides a mathematical model for mapping the joint angles to the end-effector position in order to understand the overall configuration and motion of the robot. Equation (3) describes a forward kinematics implementation.

Algorithm 3 Forward Kinematics

Input: Current joint angles q, Segment Length l.
Define origin position in 2D space: $p \leftarrow (0, 0)$
for each i in $[0, \ldots, N-1]$ **do**
    Update position
    $p_0 \leftarrow p_0 + \cos(q_i) * l_i$
    $p_1 \leftarrow p_1 + \sin(q_i) * l_i$
end for

Throughout this thesis this function is used with a constant segment length $l = \{1\}^N$ therefor it is referred to as in Equation (3.34) which maps an angle configuration q into the end-effector position in 2D space.

$$[p]fk : \mathbb{R}^N \rightarrow \mathbb{R}^2 \tag{3.34}$$

3.6.2 Inverse kinematics

Inverse kinematics (IK) is a fundamental problem in robotics, animation or virtual reality that involves finding a required joint angle configuration or positions to reach a desired end-effector position and optionally a desired orientation. It plays a crucial role in controlling the motion and manipulation of robotic systems, enabling them to interact with the environment and perform complex tasks. In this section you will find a brief summary of existing approaches, a deeper explanation of the Cyclic Coordinate Descent algorithm and the description of the used inverse kinematics RL environment.

Existing Approaches to Solve Inverse Kinematics

Numerous approaches have been proposed to solve the inverse kinematics problem. These approaches can be broadly categorized into:

- analytical methods: rely on geometric methods to derive closed form solutions.
- numerical methods: iteratively approximate the joint angles that satisfy the desired end-effector position and orientation.
- heuristic methods: iteratively propagate positions along the kinematic chain to converge on the desired end-effector position.

- sampling-based methods: randomized search strategy to explore the joint space and find feasible solutions to the inverse kinematics problem

Cyclic Coordinate Descent

Cyclic Coordinate Descent (CCD) is a popular numerical method for solving inverse kinematics. It is an iterative algorithm that adjusts the joint angles of a robotic system one at a time, from a base joint to the end-effector, in order to align the end-effector with the desired target position.

The algorithm works by iteratively updating the joint angles based on the discrepancy between the current and desired end-effector positions (p_{target}). At each iteration, CCD focuses on a single joint and adjusts its angle to minimize the positional error. By sequentially updating the joint angles in a cyclic manner, CCD aims to converge towards a solution that satisfies the desired end-effector position.

Algorithm 4 Cyclic Coordinate Descent Pseudo Code

Input: Current joint angles q, Desired end-effector position p_{target}.
while unitl convergence **do**
for each i in $[N-1, \ldots, 0]$ **do**
Calculate the vector from the current joint position to the end-effector position:
$V_{\text{current}} \leftarrow p_{N-1} - p_i$
Calculate the vector from the current joint position to the target position:
$v_{\text{target}} \leftarrow p_{\text{target}} - p_i$
Calculate the rotation necessary to align v_{current} with v_{target}:
$\delta q_i \leftarrow \text{angle\_between}(v_{\text{current}}, v_{\text{target}})$
Update the joint angel:
$q_i \leftarrow q_i + \delta q_i$
end for
end while

As illustrated in Algorithm 4 and Figure 3.10 in each iteration, CCD calculates the vector from the current joint position p_i with p_i as the position of the ith joint with $i \in \{0, \ldots, N-1\}$, to the end-effector position (v_{current}) and the vector from the current joint position to the target position (v_{target}). By finding the rotation necessary to align v_{current} with v_{target}, represented as $\delta\phi$, the algorithm updates the joint angle accordingly. This process is repeated for each joint in the kinematic chain until convergence is achieved.

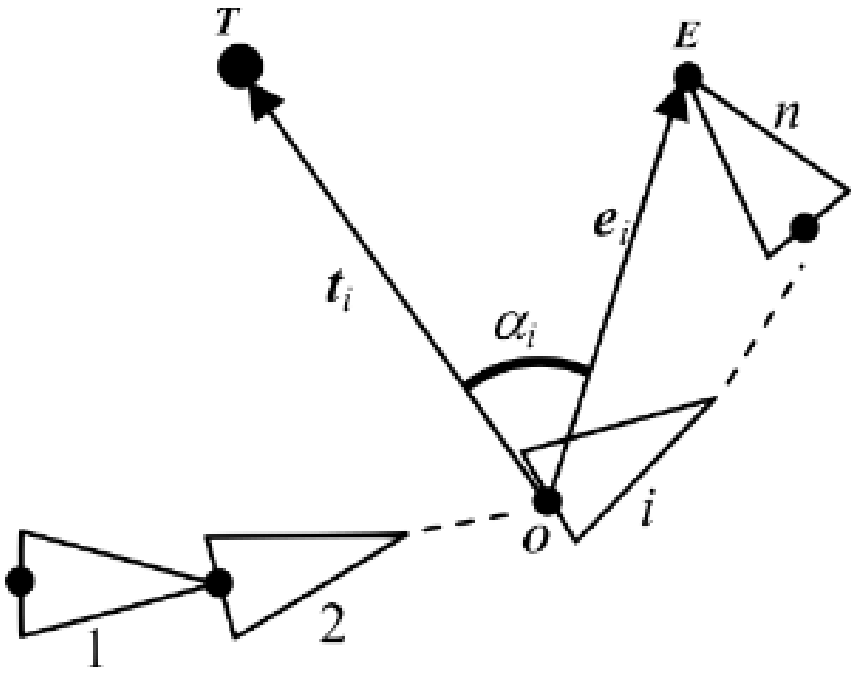


Figure 3.10: CCD geometry

(TODO: make own figure)

(TODO: experiments for runtime analysis in dependence of number of joints)

4 Methodology

covers: research idea rl-environment experiments on VAE, Supervised and merged pipeline

4.1 Research Idea

The primary research idea of my thesis is to pursue a transformation from the RL environment action space $\mathcal{A}$ into a lower-dimensional latent action space $\mathcal{A}_L$. This latent action space will align with the latent space between the encoder and decoder of a VAE model or the feature space of a feed-forward-neural network. By doing so, we aim to enable RL agents to effectively explore and learn within a more compact and smoothened action representation.

We propose two potential approaches for achieving this reduction in dimensionality: VAEs and supervised models.

In the VAE approach, a conditional or unconditional generative model is employed to learn a latent representation of the action space. By training the VAE on actions from an expert or on state target combinations to emphasis a solution which is independent from an expert, we aim to capture the underlying structure and patterns within the action space from the RL environment. This latent representation can potentially offer a more concise and informative representation of the actions, encourage more efficient learning and exploration for RL agents.

Alternatively, the supervised model approach involves training a supervised learning model, such as a neural network, to directly transform the a defined lower dimensional latent action into the high-dimensional action space. This transformation is learned based on labeled examples of actions and their corresponding latent representations. In the conducted experiments the lower dimensional-action space is just the desired action outcome. By leveraging supervised learning techniques, we aim to find a mapping between state information and desired action outcome to the RL environment action space, allowing RL agent to operate effectively on a higher level within the reduced-dimensional latent action space..

4.2 RL Environment

To apply RL in general you need an environment the agent can send actions to and receives feedback as discussed in Section 3.1. As previously introduced the key problem we are targeting is inverse kinematics of a robot arm. This problem turn out to be suitable because:

- the action space is scalable by simply adding additional joints to the robot arm
- it can be simplified into a 2D space with a fast and reliable implementation
- it is expandable. You are always able to extend the environment with additional constrains like objects the robot has to navigate around or joint angle constrains.

In this section I'm going to present the a novel RL-Environment for bench-marking the performance of different algorithms to solve inverse kinematics for a robot arm with N many joints and subsequently N many segments with lengths $l \in \mathbb{R}^N_{>0}$ in 2D space. For simplicity reasons l is constant with $l = \{1\}^N$.

4.2.1 State Space

The state space $\mathcal{S} \subset \mathbb{R}^{4+N}$ for this environment consists of three main building blocks.

- **goal information** $p_{\text{target},t} \in \mathbb{R}^2$: This vector provides information where to move the end-effector. This position is lies always in for the robot arm, reachable distance. Mathematical speaking: $||p_{\text{target}}||_2 \leq \sum l$

- **state position** $p_{\text{current},t} \in \mathbb{R}^2$: This vector contains the current end-effector position in 2D space and should help the agent to understand where the end-effector is placed and how an action has influenced the current end-effector position. Because the current position is attached to the robot arm: $||p_{\text{current}}||_2 \leq \sum l$.

- **joint angles** $q_t \in [0, 2\pi)^N$: This vector contains information about the joint angle configuration at time t.

A state $\int_t \in \mathcal{S}$ at time t has for all t the same composition

$$[p]\int_t = (p_{\text{target},t}, p_{\text{current},t}, q_t) \tag{4.1}$$

To refer to individual parts from index i to j of a state with: $s_{t,(i,j)}$. Therefor we can extract the individual parts with:

- $p_{\text{target},t} = s_{t,(0,1)}$

- $p_{\text{current},t} = s_{t,(2,3)}$

- $q_t = s_{t,(4,N+4)}$

4.2.2 Action Space

The action space $\mathcal{A} \subseteq \mathbb{R}^N$ for this particular environment is continuous and contains all possible joint angle configurations for a robot arm with N joints.

A generated action $\hat{a}$ from the agent is sent to the environment. Inside the environment the incoming action is added on top of the current state angles q_t. To ensure the constrains of $q_{t+1} \in [0, 2\pi)^N$ we take the signed remainder of a division by 2π to write into q_{t+1}:

$$q_{t+1} = (q_t + \hat{a}) \ \% \ 2\pi$$

Subsequently after updating the state angles the current end-effector position get updated by a forward kinematics call on q_{t+1}:

$$p_{\text{current},t} = fk(q_{t+1})$$

4.2.3 Reward Function

The reward function $R : \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ as in Equation (4.2) for the conducted experiments and this environment aims to minimize the distance between the current end-effector position $p_{\text{current},t}$ and the current target position $p_{\text{target},t}$. The current end-effector position is calculated by the forward-kinematics function on state-angles q_t plus action a_t. The target position is sampled at the beginning of an episode and stays constant throughout the episode until completion or time limit is reached.

$$R(s_t, a_t) = ||\text{forward kinematics}(s_{t,(4,\ldots,N+4)} + a_t), s_{t,(0,1)}||_2 \tag{4.2}$$

4.3 Dataset Creation

why do we need a dataset what kinds of different ways to create a dataset

4.3.1 Uniform Sampling

Vanilla uniform sampling algorithm for sampling from a uniform distributions. What can we observe: radius distribution of 2D forward kinematics is very similar to a half Normal distribution with increasing number of joints the std relative the arms reach shrinks plot how strong it shrinks but we need a uniform distribution wrt. radius and angle.

4.3.2 Expert Guidance

Dataset creation with expert guidance describes an algorithm: 1. sample a position in 2D space from a uniform and takes it as the state position 2. sample initial arm angles from a uniform distribution 3. solve IK for this position and with the random angles from 2. as start and take the resulting angles as the state angles 4. sample a second position in 2D as the target position 5. solve IK for the target position and with the state angles as start and take the difference between resulting angles and state angles as the action.

advantages: - we are able to sample target positions and state positions uniformly wrt. angel and radius

disadvantages: - we are bound the the solutions an expert is providing - it is slower compared to the vanilla uniform sampling algorithm

Plots Runtime complexity wrt. number of joints expert actions with 2 joints for demonstrating that the actions of the action space cover only a very limited subset of possible actions. Why Two -> because we can just draw a heatmap in two dimensions

4.4 Latent Criterion

In this section I will cover the employed loss functions to train the latent models.

(TODO: KL div)

Imitation Loss. The imitation loss is a criterion to minimize the mean squared error between a given label y, in this case an action from an expert which results in $y \in \mathbb{R}^N$ and the predicted action $\hat{x} \in \mathbb{R}^N$. It is defined as in Equation (4.3).

$$\begin{aligned} \mathcal{L}_{\text{Imitation}} : \mathbb{R}^N \times \mathbb{R}^N &\to \mathbb{R} \\ y, \hat{x} &\mapsto \frac{1}{N} \sum_{i=0}^{N-1} (y_i - \hat{x}_i)^2 \end{aligned} \tag{4.3}$$

[p]

Distance Loss. This loss function minimizes the distance between a desired position in 2D space as label p_{target} and the action outcome of a state action combination. The state information needed for this criterion are only the current robot arm angles q. Like before the action is referred to as $\hat{x}$. **(TODO: wirte forward kinematics algorithm)**

$$[p]\mathcal{L}_{\text{Distance}}(p_{\text{target}}, \hat{x}, q) = ||p_{\text{target}} - \text{forward kinematics}(q + \hat{x})||_2 \tag{4.4}$$

Inverse kinematics Loss. This criterion as in Equation (4.5)is the result of merging distance loss and imitation loss is one loss function as a weighted sum of those two components with weights $w_{\text{Imitation}}, w_{\text{Distance}} \in \mathbb{R}$. In the implementation the loss function can be configured to work also a pure distance loss function without providing any expert action.

$$\mathcal{L}_{\text{IK}}(y, p_{\text{target}}, \hat{x}, q) = w_{\text{Imitation}} \cdot \mathcal{L}_{\text{Imitation}}(y, \hat{x}) + w_{\text{Distance}} \cdot \mathcal{L}_{\text{Distance}}(p_{\text{target}}, \hat{x}, q) \quad (4.5)$$

4.5 Learning Variational Autoencoder

4.5.1

4.6 Learning conditional Variational Autoencoder

5 Experiments

5.1 VAE

In the folling section I will present the results of my experiments with the Variational-Autoencoder to encode and decode actions from the environment. Consistent over all experiments with the VAE sizes of the datasets are consistent:

- train: 10.000 actions
- validation: 2000 actions
- test: 1000 action

5.1.1 Pure Actions

In every state of the sequential decision making process there are multiple actions available to go from the current state s_t to the next state s_{t+1} where the arm end position is at the goal position. To gain a dataset that contains such actions I sampled the robot arm angles in s_t from a uniform distribution $\mathcal{U}_[0, 2\pi]$ and applied the CCD algorithm to get access to the action that leads from s_t to s_{t+1} with the robot arm end position near the goal state.

The results of the training process are shown in Figure **??**

n	latent dim	r loss	kl loss
2	2		
2	1		
5	5		
5	4		
5	3		
5	2		
5	1		
10	2		
10	2		
10	2		
10	2		

Table 1: n is the number of joints, latent dimension is the size of the dimension the action is compressed to, r loss is the test reconstruction loss I used the MSE as the metric for the reconstruction loss, kl loss is the Kullback Leiber diveregence from the standard normal distribution on the test dataset

5.1.2 Conditioning on States

After watching the pure action encoding and decoding fail in the all important reconstruction loss. I got inspired by the paper **(TODO: cite LASER)** and rerolled the experiments with an conditional Variational-Autoencoder where the conditional information is the information from s_t. The results of the training process are shown in Figure **??**

We can observe that the test reconstruction loss is much lower that before and the kl loss

(TODO: Concering the table: make a lot of experiments so I can get an std ?, or should I just say that the training turns out to be not as stable as it should be in comparison to an VAE on MNIST and that we sampled runs until we got a good result?)

5.1.3 Fitting Random Noise

In this section I will present the results on how we tried to encode and decode actions sampled from a parameterized distributuion. The idea is that in every state the agent can choose an action from $[-1, 1]^n$. Therefor I sampled a dataset independent and identically distributed from $\mathcal{U}^n_{[-1,1]}$. this could be describe of trying to fit random noise. The results for 800 epochs are shown in Figure **??**.

5.2 SAC

5.2.1 Hyperparameters

(TODO: description?)

5.2.2 Baseline

In this section I want to present the results of the baseline experiments. To mimic a sequential decision making process and constrain the actions I manually tuned the parameter "action magnitude" between 1, 0.5, 0.2, and 0.1. The results are shown in figures: Figure **??**

Stichworte die rein sollen - Mit erhörter Anzahl an joint wid die Performance ober alle experimente immer schlechter - Affällig hohe Varianz bei kleiner werdeneder Anzahl joint - Absacken der Performance von 2 joints mit weiter Erniedrigung der action magnitude - 5 joints scheinen eine ähnliche Performance über die action magnitude abzu geben - Scheinbar habe ich bei 0.2 einen Wert gefunden der auch für höhere Anzahlen von joints gut zu funktionieren scheint. -> Fahre daher weiter mit diesem Parameter fort.

name	values
task	ReachGoalTask, ImitationTask
lr pi	0.0005
lr q	0.001
init alpha	0.01
gamma	0.98, 0
batch size	32
buffer limit	50000
start buffer size	1000
train iteration	20
tau	0.01
target entropy	-40.0
lr alpha	0.001
n epochs	5000
action covariance mode	independent
action covariance decay	0.5
action magnitude	0.1

Table 2: Hyperparameters for the Soft-Actor-Critic algorithm

6 Conclusion

7 Acknowledgments

First and foremost, I would like to thank Jasper Hoffman for his support.

I also thank Jan Ole von Harzt and Jens Rahnfeld for great discussions about Variational Autoencoders

- advisers
- examiner
- person1 for the dataset
- person2 for the great suggestion
- proofreaders

ToDo Counters

To Dos: 54; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54

Parts to extend: 1; 1

Draft parts: 1; 1

Bibliography

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

[2] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "siamese" time delay neural network," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 04, pp. 669–688, 1993.

[3] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration.," *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.