Improving sample-efficiency of model-free reinforcement learning algorithms by learning latent space representations

A systematic analysis of leveraging state representation learning for more efficient model-free reinforcement learning

Master's thesis in Computer science and engineering

Marko Guberina, Betelhem Dejene Desta

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

Improving sample-efficiency of model-free reinforcement learning algorithms by learning latent space representations

A systematic analysis of leveraging state representation learning for more efficient model-free reinforcement learning

Marko Guberina, Betelhem Dejene Desta

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Improving sample-efficiency of model-free reinforcement learning algorithms by learning latent space representations
A systematic analysis of leveraging state representation
learning for more efficient model-free reinforcement learning
Marko Guberina, Betelhem Dejene Desta

Supervisor: Divya Grover
Examiner: Claes Strannegård

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Improving sample-efficiency of model-free reinforcement learning algorithms by learning latent space representations
A systematic analysis of leveraging state representation
learning for more efficient model-free reinforcement learning
Marko Guberina, Betelhem Dejene Desta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Will be written last.

Keywords: reinforcement learning, state representation learning, efficient model-free methods, autoencoder

Acknowledgements

We give special thanks to the supervisor, examiner and everyone else at Chalmers who made this work possible.

Marko Guberina, Betelhem Dejene Desta, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1

Introduction

1.1 What is reinforcement learning?

In computer science, reinforcement learning is the formalization of trial-and-error learning. While this is not the only legitimate interpretation of the concept, it is the most straightforward one: "trial" implies existence of an agent which observes its environment and interacts with it though its own actions. "Error" implies that the agent has a goal it tries to achieve and that it does not know how to achieve it (in the most effective manner). What it can do is take different actions and appraise them according to how closely they lead the agent toward its goal, thereby observing the quality of those actions. By repeatedly exploring the effects of various sequences of actions, the agent can find, i.e. learn, the sequence of actions which lead to its goal.

Here, it is important to discuss what a goal is. To formalize the process outlined above, one needs to describe it in purely mathematical terms. Thus, among other things, the goal needs to be described numerically. To achieve that, the notion of a reward function is used: it maps every state of the environment to a number which denotes its value called the reward. The state of the environment to which the highest reward is ascribed is then the goal. A more general description of the goal of reinforcement learning is to maximize the sum of rewards over time. The formalization of the entire process will be carried out later in the text, while here only the most important concepts will be outlined.

Due to its generality, reinforcement learning is studied in many different disciplines: control theory, game theory, information theory, simulation-based optimization, multi-agent systems etc.. Of these, control theory is of particular importance because it often enables clear analysis of various reinforcement learning algorithms. This foremost concerns the usage of dynamic programming which provides a basis for a large class of reinforcement learning algorithms. Reinforcement learning is also considered to be one of the pillars of modern data-driven machine learning.

In the context of machine learning, reinforcement learning can be view as a combination of supervised and unsupervised learning: the "trial" portion of the trial-and-error learning can be interpreted as unsupervised or as self-supervised learning because in it the agent collects its own dataset without any explicit labels to guide its way. This process is referred to as "exploration". The dataset created by exploration is labelled by the reward function. Thus the agent can learning from "past experience" in a supervised manner. This text will introduce concepts from both control theory and machine learning which are necessary to formalize the reinforcement learning objective and to develop algorithms to achieve it. It will not concern itself with other disciplines.

1.2 Why is reinforcement learning interesting?

Interest in reinforcement learning has grown tremendously over the past decade. It has been fueled by successes of deep machine learning in fields such as computer vision. The subsequent utilization of neural networks in reinforcement learning, dubbed deep reinforcement learning, led to impressive results such as achieving better-than-human performance on Atari games, in the game of go and in many others. Because large amounts of data are required for neural network training and thus for reinforcement learning algorithms which utilize them, most of these results are achieved in computer-simulated environments. [1] These recent success were kick-started by Deep Q-Network (DQN) algorithm [Mni+13] which crucially, by utilizing convolutional neural network, enabled the agents to successfully learn from raw pixels. Learning from pixels is incredibly important for many practical applications, such as those in robotics where it is often impossible to get full access to the state of the environment. The state then needs to be inferred from observations such as those from cameras. [2] Due to incredible results achieved in simulated environments, reinforcement learning holds the promise of solving many incredibly important engineering problems, for example robotic manipulation and grasping. Having that said, there exists a large gap between simple simulated environments and the real world, and many improvements to the current state-of-the-art algorithms are required to bridge that gap. To explain the approach investigated in this thesis, a bit more context is needed.

1.3 Efforts in making reinforcement learning more efficient

An important classification of reinforcement learning algorithm is the one between model-based and model-free algorithms. As the name suggests, model-free algorithms do not form an explicit model of the environment. Instead, they function as black-box optimization algorithms, simply finding actions which maximize reward without other concerns such as predicting the states resulting from those actions. In other words, they only predict the reward of actions in given states. Model-based algorithms on the other hand learn an explicit model of the environment and use it to plan their actions. They thus learn the dynamics of the environment and use that knowledge to choose actions which lead the agent to states with high reward. Both

[1] Simulated environments run as fast as the computers they run on, which enables generating thousands of trials in seconds.

[2] Here the state refers to the underlying physical parameters of the environment: the positions and velocities of objects, the friction coefficients and so on. Observations from sensors such as cameras do not explicitly provide such information. However, since humans and animals are able to utilize such observations to achieve their goals, we know that they implicitly hold enough information about the true state of the world for successful goal completion.

classes have their benefits and their drawbacks. Since model-free algorithms do not require any knowledge of environment dynamics to operate, they are more widely applicable and usually achieve better performance. But the fact that they can not leverage environment dynamics to create plans implies a harder learning problem: they need to implicitly learn those dynamics while only being provided the reward signal. This makes them much less sample-efficient.

By contrast, model-based algorithms are of course more sample-efficient. Furthermore, the plan generated from the learned model can be utilized to interpret the agent's actions which in turn leads to many further benefits such as the ability to guarantee outcomes in safety-critical operations. Unfortunately, the twin learning objective of learning the best action-choosing policy to maximize the reward over time, and the learning of the model results in fundamental training instabilities which usually results in worse final performance. In simple terms, the reason behind this is the following one: in the beginning of the learning process, both the policy and the model perform poorly. For the model to perform better, the agent needs to explore the environment and update its model. However, many parts of the environment are inaccessible to a poorly performing agent: for example, if an agent is playing a computer game, and it is not able to progress to further sections of the game, it will not be able to construct a model of that portion of the game. Thus, to explore the environment and improve its model, it needs to first learn exploit the model and perform sufficiently well using it. Furthermore, what it learned at this stage may become obsolete as the model changes. How bad this problem is depends on the specifics of the setting, and there are many ways to ameliorate it, but in most cases the necessary trade-offs result in a lower final performance. All this will be further discussed in a later chapter.

1.4 Goal of the thesis

Given the previous discussion, the goal of the thesis may be presented: the idea is to combine the sample-efficiency of model-based approaches with the flexibility of model-free methods. Another way to describe the same is to say that we want to utilize learning signals other than the reward signal to make the model-free learning more sample-efficient. In particular, we want to learn a latent representation of the environment, i.e. to find lower-dimensional embeddings of the environment, and learn a policy in this space. To make this a concrete and manageable goal, we constrain ourselves to the problem of learning from images in particular. To be able to compare our results to those of other researchers, we will test our algorithms on the standard benchmark tasks in the field, namely Atari57 games. The potential benefits of the proposed approach are two-fold:

- First, we know that, in general, lower-dimensional optimization problems are easier to solve than higher-dimensional ones.
- Second, it is known that when algorithms learn with direct state access, they learn much faster and often achieve better final results. The main reason behind this is that images are much higher-dimensional than underlying states, and this is self-evident in the case of Atari games. Since inferring states from

observations is not directly related to the reward, we expect that using unsupervised learning techniques will aid in feature extraction and thus make learning more sample-efficient.

and thus make learning more sample-efficient. Furthermore, since the goal is not to learn the dynamics of the environment, but simply to find an equivalent, but lower-dimensional representation of it, we expect that this approach won't suffer from the problems faced by model-based approaches. Of course, we are not the first to suggest such an approach. An overview of the field is provided in 3.

1.4.1 Hypothesis

As already stated, we believe that leveraging unsupervised learning techniques to learn state representations will make reinforcement learning from images more sample-efficient. Testing whether this is in fact true is one of our tasks. As will be shown in **??**, there are many different unsupervised learning techniques which can be adapted to the goal of state representation learning. Furthermore, there are many different ways in which state representation learning can be integrated in the reinforcement learning process. In this thesis, our goal is not to arrive at a new state-of-the-art algorithm, but to investigate which properties of both the state representation learning and its integration with reinforcement learning yield better results. We will not test all of the existing approaches, but rather identify their common properties, form hypothesis based on those properties and perform tests on a simple implementation. Here we offer our hypothesis:

1. State representations found by general unsupervised learning techniques will not equate to true states, although they will be closer to them than raw image-based observations. Reinforcement learning algorithms are able to implicitly learn true states, but because they do so indirectly and by using the weak reward signal, they do so very slowly. Thus we hypothesize that allowing the reinforcement learning algorithm to continue updating feature extraction provided the state representation learning algorithm will perform better than feature extraction learned solely through state representation learning. We further hypothesize that the best feature extraction will be obtained if both state representation learning algorithm and the reinforcement learning algorithm continuously update the feature extractor throughout the entire training process.
2. We hypothesize that state representation learning algorithms whose learned features better match the underlying Markov decision process will yield better results. This for example means that representations learned on future prediction tasks will perform better than those which are not incentivized to learn dynamics.
3. Finally, we hypothesize that strong regularization of the state representation learning algorithms will yield better results. We believe that proper regularization will yield broader features which the reinforcement learning algorithm will more easily integrate with.

1.4.2 Contributions

As already mentioned, our main goal is not to produce a state-of-the-art algorithm, but rather to find general properties which state representation learning algorithms should have and how they should best be integrated with reinforcement learning algorithms. In this thesis we provide the following contributions:

1. A systematic overview of recent works which leverage state representation learning to make model-free reinforcement learning more sample-efficient.
2. Extensive testing of our hypothesis which illuminate the problem and pave the wave for further algorithm development.
3. Implementation of our method in a high-quality reinforcement learning library. Despite the fact that our method is not the best available one, its generality and its implementation makes it easily accessible to practitioners and helps researchers who wish to build on top of it.

1.5 Outline

The rest of this text is organized as follows. We begin by describing the basics of reinforcement learning in 2.1. Here the problem setting and basic concepts are covered. Then main classes of reinforcement learning are introduced in 2.2. Having the basics established, in **??**e introduce the reinforcement learning algorithm we use in our implementation and discuss common reinforcement learning problems in 2.4. Following the reinforcement learning discussion, we turn our attention to unsupervised learning on images in **??** and introduce state representation learning in 2.6. Following the background, discuss related work in 3. Here we cover several existing approaches to bolstering the sample-efficiency of model-free reinforcement learning with state representation learning. Having covered the field, in **??** we identify key factors which lead to state representations which can be leveraged by reinforcement learning algorithms. In other words, we then form the basis for our hypothesis.

2

Background

2.1 Introduction to reinforcement learning

2.1.1 Problem setting

In the usual engineering approach to problems, prior scientific knowledge is used to first describe the problem and then to define it mathematically. Once this is done, unknown variables are measured and solutions are calculated. This approach works if the inherent stochasticity of the environment can be controlled, i.e. if bounds of stochasticity are known the solution account for them and be designed to be robust to them. But some problems have circumstances which can not be known in advance, or which are incredibly hard to hand-engineer.

In those cases, an entirely different approach becomes the only viable one: designing a system which can produce and refine its own solution, or in other words, designing a system which, in a way, learn the solution by itself. This is the idea behind the learning-based approach: automating the process of learning. Crucially, now the world and how it operates is unknown and has to be discovered. The schematic 2.1 shows how this process is formulated in reinforcement learning. Reinforcement

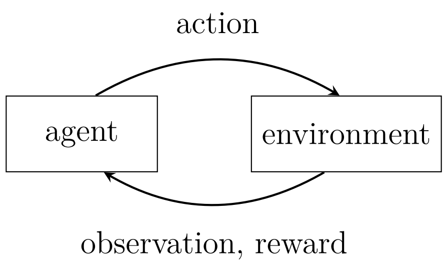


Figure 2.1: Conceptual schematic of reinforcement learning.

learning is a 2-step iterative process. The **agent**, which represents the computer program, takes **actions** in its **environment**. It then **observes** the resulting state of the environment and is also given a **reward** which is a function mapping every state of the environment to a number.

To introduce reinforcement learning more formally, we first describe the simplest possible problem to which reinforcement learning is the best solution.

2.1.2 Bandit problems

Reinforcement learning uses training information that evaluates the actions taken rather than instruct by giving correct actions. Consider this learning problem. The agent is faced with k different gambling slot machines. Each of them give random rewards under an unknown distribution. At each turn, the agents has to select one of the machines and pull its lever. The goal is to maximize the expected total reward over some number of turns. If the agent knew the distribution of rewards of each of the slot machines, it would simply choose the one with the highest expected reward in number of turns it has been given. At any time step the agent will be able to select at least one action whose estimated value is greatest. When the agent selects on of this actions it is exploiting the current knowledge of the values of the actions. If the agents keep exploiting the goal of maximizing reward over period of time will be trivial. If instead the agent selects one of the non greedy action this will enables it to improve the average expected reward over time.
When addressing the canonical problem of sequential decision making under uncertainty, the exploitation-exploration trade-off is highlighted.More specifically, as depicted in Fig.1, an agent interacts with an unknown environment in a sequential manner to obtain rewards. The ultimate goal is to maximize the rewards. For one thing, the agent takes advantage of existing knowledge of the environment. For another, the agent investigates an unfamiliar environment.

2.1.3 Markov Decision Processes

The environment of k-bandit problem is static — the actions do not change the **state** of the environment. To model environments in which states change, Markov chains are used. They capture the stochastic nature of state transitions, while Markov property allows for easier mathematical analysis. The schematic of a Markov chain is shown in 2.3.

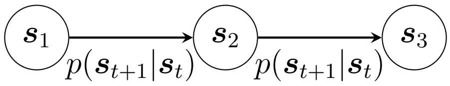


Figure 2.2: Schematic of a Markov chain.

Formally, a Markov chain $\mathcal{M}$is defined by its state space $\mathcal{S}$ with discrete or continuous state $\boldsymbol{s} \in \mathcal{S}$ and the transition operator $\mathcal{T}$. The notation $\boldsymbol{s}_t$ denotes the state at time t and it is a vector of real numbers. The transition operator allows for a succinct description of environment dynamics. For a transition probability $p(s_{t+1}|s_t)$, let $\mu_{t,i} = p(\boldsymbol{s}_t = i)$ and $\mathcal{T}_{i,j} = p(\boldsymbol{s}_{t+1} = i|\boldsymbol{s}_t = j)$. Then $\overrightarrow{\mu}_t$ is a vector of probabilities and $\overrightarrow{\mu}_{t+1} = \mathcal{T}\overrightarrow{\mu}_t$. Importantly, $\mathcal{T}$ is linear.
To model the agent's actions, we simply augment the Markov chain by adding actions as priors to state transition probabilities and defining the reward function, thereby constructing a Markov decision process. It's schematic can be seen in **??**.
The Markov decision process is thus defined by the tuple $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$. $\mathcal{A}$ denotes the action space, where $\boldsymbol{a} \in \mathcal{A}$ is a continuous or discrete action and r is the reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. It should also be noted that the transition operator is

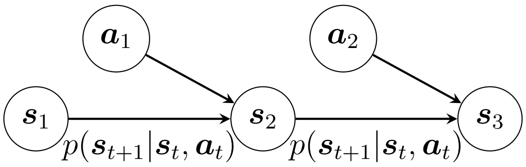


Figure 2.3: Schematic of a Markov decision process.

now a tensor. Let $\mu_{t,j} = p(s_t = j), \xi_{t,k} = p(a_t = k), \mathcal{T}_{i,j,k} = p(s_{t+1} = i|s_t = j, a_t = k)$. Then $\mu_{t+1,i} = \sum_{j,k} \mathcal{T}_{i,j,k}\mu_{t,j}\xi_{t,k}$. Therefore, $\mathcal{T}$ retains its linearity.
Finally, partial observability also needs to be accounted for. To do so, a partially observable Markov decision process (POMDP) needs to be constructed. This is done by augmenting the Markov decision process to also include the observation space $\mathcal{O}$, where observations $\boldsymbol{o} \in \mathcal{O}$ denote the discrete or continuous observations and the emission probability $\mathcal{E}$ which describes the probability $p(\boldsymbol{o}_t|\boldsymbol{s}_t)$ of getting the observation $\boldsymbol{o}_t$ when in state $\boldsymbol{s}_t$. The schematic can be seen in 2.4.

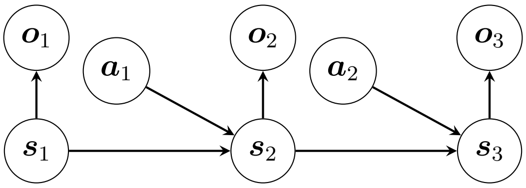


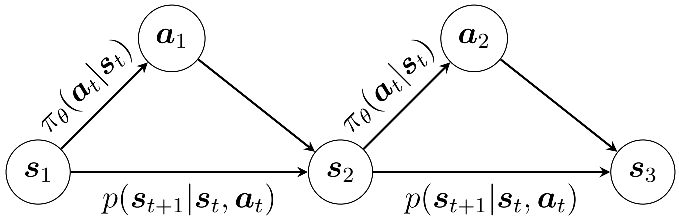
Figure 2.4: Schematic of a partially observable Markov decision process.

It is important to note that not all elements of POMDP are present in every problem: for example, the reward may be a deterministic function of the state and so on. In general through the text, to aid in simplifying notation, only the necessary elements will be explicity referenced in sketches and written out in the equations (most often using just the Markov decision process).

2.1.4 Key concepts in reinforcement learning

2.1.4.1 Policy

With the problem space being formally defined, we may introduce definitions which will allow the construction of a reinforcement learning algorithm. The reinforcement learning problem can be defined in finite or infinite time horizons. Different environments usually naturally fall in either category. For the agent to learn, it needs to be able to try out different actions from the same, or at least similar states. This is usually achieved by having the agent return to a set of starting states. The period between two such returns is called **an episode**. The agent selects actions based on its **policy** π. The policy is a function which maps states to actions. The schematic showing it in the context of a Markov decision process is given in **??**.
The policy is a stochastic function. The intensity of stochasticity determines the trade-off between exploration and exploitation. To emphasize that the policy depends on some parameters θ, we usually write π_θ.



2.1.4.2 Goal of reinforcement learning

For simpler notation, the finite horizon form is assumed for the following definitions. Since the environment is modeled as a Markov decision process, we can write the probability of observing a trajectory of states and actions as:

$$\underbrace{p_\theta(\boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_T, \boldsymbol{a}_T)}_{p_\theta(\tau)} = p(\boldsymbol{s}_1) \prod_{t=1}^{T} \underbrace{\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)}_{\text{Markov chain on}(\boldsymbol{s},\boldsymbol{a})} \tag{2.1}$$

A bit more explicitly, we can a transition probability as:

$$p((\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1})|(\boldsymbol{s}_t, \boldsymbol{a}_t)) = p((\boldsymbol{s}_{t+1}|(\boldsymbol{s}_t, \boldsymbol{a}_t))\pi_\theta(\boldsymbol{a}_{t+1}|\boldsymbol{s}_{t+1}) \tag{2.2}$$

With this, we may now formally define the goal of reinforcement learning. It is to find policy parameters $\theta^\star$ such that:

$$\theta^\star = \operatorname*{argmax}_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{2.3}$$

$$= \operatorname*{argmax}_\theta \sum_t^T E_{(\boldsymbol{s}_t, \boldsymbol{a}_t) \sim p_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t)} \left[r(\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \tag{2.4}$$

To ensure that the expected sum of rewards, also know as the **return**, is finite in the infinite horizon case, a **discount factor** $0 < \gamma < 1$ is introduced in the sum. The discount factor also play a role in modelling because often times it makes sense to value immediate rewards more. It is important to note that we are maximizing the *expected* sum of rewards. This makes the goal a smooth and differentiable function of the parameters, which means we can employ gradient descent to find the optimal parameters. This leads us to the first class of reinforcement learning algorithms: policy gradient algorithms. They will be introduced with the other classes of algorithm in the next subsection, while here additional concepts required by other classes of algorithms will be introduced here.

2.1.4.3 Value functions

Value functions are functions which map states or state-action pairs to the expected returns obtained under a fixed policy. They are a concept from dynamic programming. In fact, reinforcement learning can be interpreted as an extension of dynamic programming, as shall be done in the following subsection. Having that said, value function can be interpreted in other ways as well. The **Q-function** maps state-action pairs to the estimated sum of returns under policy π_θ:

$$Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{t'=t}^{T} E_{\pi_\theta} \left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) | \boldsymbol{s}_t, \boldsymbol{a}_t \right] \tag{2.5}$$

thus denoting the total reward from taking $\boldsymbol{a}_t$ in $\boldsymbol{s}_t$. **Value functions** map states to to the estimated sum of returns under policy π_θ:

$$V^\pi(\boldsymbol{s}_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}|\boldsymbol{s}_t)\right] \tag{2.6}$$

The connection between the two is the following:

$$V^\pi(\boldsymbol{s}_t) = E_{\boldsymbol{a}_t \sim \pi(\boldsymbol{s}_t, \boldsymbol{a}_t)}\left[Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t)\right] \tag{2.7}$$

And we can also write the RL objective as:

$$E_{\boldsymbol{s}_1 \sim p(\boldsymbol{s}_1)}\left[V^\pi(\boldsymbol{s}_1)\right] \tag{2.8}$$

2.2 Classes of reinforcement learning algorithms

2.2.1 Policy gradients

Policy gradients are derived by directly solving for the reinforcement learning objective with gradient descent with respect to the policy parameters. To do so, the reinforcement learning objective needs to be evaluated. We begin by introducing a notational shorthand:

$$\theta^\star = \underset{\theta}{\operatorname{argmax}}\, \underbrace{E_{\tau \sim p_\theta(\tau)}\left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right]}_{J(\theta)} \tag{2.9}$$

We estimate $J(\theta)$ by making rollouts from the policy (below i is the sample index and i, t is the t^{th} timestep in the i^{th} sample):

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\sum_t r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right] \approx \frac{1}{N}\sum_i \sum_t r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \tag{2.10}$$

Simplifying the notation further, we get:

$$J(\theta) = E_{\tau \sim p_\theta(\tau)} \underbrace{[r(\tau)]}_{\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)} = \int p_\theta(\tau) r(\tau)\, d\tau \tag{2.11}$$

The goal now is to compute the derivative of the estimated reinforcement learning objective:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau)\, d\tau \tag{2.12}$$

Since the goal of this text is just to introduce the necessary concepts and algorithms, the derivation(s) will be ommited. We encourage the interested reader to consult the literature [SB18] and CITE LEVINE'S BERKLEY LECTURES to find them.
Here we will just note that it is crucial that the final expression can be estimated by sampling the agent's experience as the other quantities are not available. The resulting expression for the policy gradient is:

$$\nabla_\theta J(\theta) = E_{\tau \sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)\right)\left(\sum_{t=1}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t)\right)\right] \tag{2.13}$$

To evaluate the policy gradient we can sample:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right) \tag{2.14}$$

With the gradient we can do a step of gradient ascent and use it to form the REINFORCE algorithm, also known as "vanilla policy gradient":

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\boldsymbol{a}_t|\boldsymbol{s}_t)$ by running the policy
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t^T \nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \right) \left(\sum_t r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

This algorithm does not work well in practice. The main reason for that is that the variance of returns is very high. However, there are a number of modifications which dramatically improve its performance. Since the goal of this text is not to outline every reinforcement learning algorithm, we will introduce only the modifications which outline general trade-offs and principles in reinforcement learning algorithm design.

2.2.1.1 Baselines

The policy gradient in the REINFORCE algorithm lacks some important properties. One of them is that it should, ideally, make bad actions less likely and good actions more likely. However, if all rewards are positive, then all actions' probabilities will be increased, only by different amounts. This can be changed if a **baseline** $b$is added to actions:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log p_\theta(\tau)[r(\tau) - b] \tag{2.15}$$

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau) \tag{2.16}$$

This addition does not change the gradient in expectation, i.e. it does not introduce bias, but it does change its variance. Although an optimal bias can be calculated, it is rarely used in practice due to its computational cost. Using baselines is one of the key ideas in actor-critic algorithms so they will be discussed further there.

2.2.1.2 Off-policy gradients

An important property of the REINFORCE algorithm is that it is an **on-policy** algorithm. This means that new samples need to be collected for every gradient step. The reason behind this is the fact that the expectation of the gradient of the return needs to be calculated with respect to the current parameters of the policy. In other words, because the policy changes with each gradient step, old samples are effectively collected under a different policy. This means that they can not be used to calculate the expected gradient of the return with respect to the current policy — it would not produce those trajectories. In mathematical notation:

$$\nabla_\theta J(\theta) = \underbrace{E_{\tau \sim p_\theta(\tau)}}_{\text{this is the trouble!}} [\nabla_\theta p_\theta(\tau) r(\tau)] \tag{2.17}$$

If the policy is a neural network, which requires small gradient steps, the cost of generating a large number of samples for every update could make the algorithm entirely infeasible. This of course depends on the cost of generating samples, which is entirely problem dependent — policy gradient algorithms are often the best solution when the cost of generating samples is low.
However, on-policy algorithms can be turned into off-policy algorithms through **importance sampling**, which is the name given to the following mathematical identity:

$$E_{x\sim p(x)}[f(x)] = \int p(x)f(x)\,dx \tag{2.18}$$

$$= \int \frac{q(x)}{q(x)}p(x)f(x)\,dx \tag{2.19}$$

$$= \int q(x)\frac{p(x)}{q(x)}f(x)\,dx \tag{2.20}$$

$$= E_{x\sim p(x)}\left[\frac{p(x)}{q(x)}f(x)\right] \tag{2.21}$$

which is exact in expectation. To use importance sampling to create an off-policy policy gradient algorithm, certain approximations need to be made. Again, the details of the derivation are omitted and what follows is just the final result.

$$\nabla_{\theta'} J(\theta') \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})}{\pi_{\theta}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})\hat{Q}_{i,t} \tag{2.22}$$

To get this equation, the factor $\frac{\pi_{\theta'}(\boldsymbol{s}_{i,t})}{\pi_{\theta}(\boldsymbol{s}_{i,t})}$ had to be simple ignored in the expression because it is impossible to calculate the state marginal probabilities. This means that the expression works only if $\pi_{\theta'}$ is not too different from π_{θ}.

2.2.1.3 Advanced policy gradients

The basic algorithm we have outlined is essentially just a basic gradient descent method. From convex optimization, we know that it can be made much better if second order derivatives or their approximations are used. For example, conjugate gradient descent can be used. Further, there are various ways in which this optimization problem can be better conditioned. Such improvements led to algorithms such as PPO and TRPO, which will not be discussed here.

2.2.2 Actor-critic algorithms

Actor-critic methods can be seen as making a different trade-off between variance and bias in policy gradient estimation. We begin with the following observation: [1]

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_{\theta}\log\pi_{\theta}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})\underbrace{\left(\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})\right)}_{\hat{Q}_{i,t}\text{: “reward to go”}} \tag{2.23}$$

[1] In this equation, the summation of rewards is done from time t to T because actions and states prior to that time do not affect the return from that time onward. This leveraging of causality reduces the variance of the estimate.

Simply put, in the policy gradient method a single-run Monte-Carlo (MC) is used to estimate the return. This causes high variance, while incurring no bias. Another option is to try to estimate the full expectation $\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'})|\boldsymbol{s}_t, \boldsymbol{a}_t\right]$. Since the estimate won't be perfect, it will introduce bias. Of course, using multiple runs from the same state-action pair would reduce variance, but this is sometimes impossible to procure and is certainly more costly. However, if our estimator of "reward to go" can generalize between states, we will be able to get good estimates regardless.
Like the policy, the return estimator will have to be learned. In this approach, the policy is also called the **actor** and the return estimator is called the **critic**. We proceed by discussing how the critic can be constructed. If we had the correct Q-function (i.e. not the estimate, but the actual values), we could improve the policy gradient estimate by using it both to estimate the return and as a baseline:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t})(Q(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) - b) \tag{2.24}$$

$$b_t = \frac{1}{N}\sum_{i} Q(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \tag{2.25}$$

However, having a baseline that depends on actions leads to bias. Thus we employ a baseline dependent on the state:

$$V(\boldsymbol{s}_t) = E_{\boldsymbol{a}_t \sim \pi_\theta(\boldsymbol{s}_t, \boldsymbol{a}_t)}[Q(\boldsymbol{s}_t, \boldsymbol{a}_t)] \tag{2.26}$$

Since the value function 2.6 tells us the expected return of the average action, we can calculate how much better a certain action is by substracting its Q-value 2.5 for the value function. The result is called the **advantage function**:

$$A^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) - V^\pi(\boldsymbol{s}_t) \tag{2.27}$$

Thus we can fit either the Q-function, the value function or the advantage function. Of these, it is best to learn the value function because there are less states than state-action pairs. We then calculate the advantage function in the following way:

$$A^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) \approx r(\boldsymbol{s}_t, \boldsymbol{a}_t) + V^\pi(\boldsymbol{s}_{t+1}) - V^\pi(\boldsymbol{s}_t) \tag{2.28}$$

The value function can be estimated through samples

$$V^\pi(\boldsymbol{s}_t) \approx \sum_{t'=t}^{T} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) \tag{2.29}$$

After collecting many such samples

$$\left\{\left(\boldsymbol{s}_{i,t}, \underbrace{\sum_{t'=t}^{T} r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t})}_{y_{i,t}}\right)\right\} \tag{2.30}$$

we can fit the value function through supervised regresion with the loss being:

$$\mathcal{L}(\phi) = \frac{1}{2}\sum_i ||\hat{V}^{\pi}_{\phi}(\boldsymbol{s}_i) - y_i||^2 \tag{2.31}$$

However, this process can be sped up with bootstraped estimates:

$$y_{i,t} = \sum_{t'=t}^{T} E_{\pi_\theta}\left[r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'})|\boldsymbol{s}_{i,t}\right] + V^{\pi}(\boldsymbol{s}_{i,t+1}) \approx r(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) + \hat{V}^{\pi}_{\phi}(\boldsymbol{s}_{i,t+1}) \tag{2.32}$$

This will further reduce variance, but again increase bias.
Fortunatelly, we can tune the trade-off between bias and variance. In the Monte Carlo estimate, the entire trajectory was used to estimate the return. In the bootstrap estimate, only a single step in the future was used along with the estimate. Instead, a **n-step** return estimator can used:

$$\hat{A}^{\pi}_{n}(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) - \hat{V}^{\pi}_{\theta}(\boldsymbol{s}_t) + \gamma^{n}\hat{V}^{\pi}_{\theta}(\boldsymbol{s}_{t+n}) \tag{2.33}$$

In most cases the ideal trade-off for n lies somewhere between 1 and ∞ (the MC estimate). Finally, an average of all n-step return estimators can be used. This is called the generalized advantage estimator (GAE):

$$\hat{A}^{\pi}_{GAE}(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{n=1}^{\infty} (\gamma\lambda)^{t'-t} r(\boldsymbol{s}_{t'}, \boldsymbol{a}_{t'}) + \gamma\hat{V}^{\pi}_{\theta}(\boldsymbol{s}_{t'+1}) - \hat{V}^{\pi}_{\theta}(\boldsymbol{s}_{t'}) \tag{2.34}$$

where the factor λ controls the weight of future values.
Combining this into an iterative algorithm, and fixing the issues of naiive implementations results in the following algorithm:

<u>Actor-critic algorithm template</u>

1. take action $\boldsymbol{a} \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s})$, get $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'}, r)$, store in $\mathcal{R}$ (replay buffer)
2. sample a batch $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}'_i, r_i)\}$ from buffer $\mathcal{R}$
3. update $\hat{Q}^{\pi}_{\theta}$ using target $y_i = r_i + \gamma\hat{Q}^{\pi}_{\theta}(\boldsymbol{s}'_i, \boldsymbol{a}'_i)\forall \boldsymbol{s}_i, \boldsymbol{a}_i$
4. $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i \nabla_\theta \log \pi_\theta(\boldsymbol{a}^{\pi}_i|\boldsymbol{s}_i)\hat{Q}^{\pi}(\boldsymbol{s}_i, \boldsymbol{a}^{\pi}_i)$, where $\boldsymbol{a}^{\pi}_i \sim \pi_\theta(\boldsymbol{a}|\boldsymbol{s}_i)$
5. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

2.2.3 Value function methods

Value function methods use only the critic from actor-critic algorithms. Suppose that the advantage function $A^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t)$ is known. It tells us how much better the action $\boldsymbol{a}_t$ is than the average action according to the policy π. Thus, if we knew the advantage function, we could construct a deterministic **greedy policy**:

$$\pi_{\text{greedy}}(\boldsymbol{s}_t|\boldsymbol{a}_t) = \begin{cases} 1 & \text{if } \boldsymbol{a}_t = \text{argmax}_{\boldsymbol{a}_t} A^{\pi}(\boldsymbol{s}_t, \boldsymbol{a}_t) \\ 0 & \text{otherwise} \end{cases} \tag{2.35}$$

which would yield the highest expected return. In other words, if we knew the advantage function, the policy would be reduced to the argmax operation.

2.2.3.1 Dynamic programming

Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. They are of limited utility in reinforcement learning due to the perfect model requirement and their great computational expense, but are important theoretically — they provide an essential foundation for understanding the other methods. Usually a finite MDP is assumed. DP can be applied to continous problems as well, but exact solution exist only in special cases.
Value iteration and Q-learning make up two fundamental algorithms of Reinforcement Learning.Q learning is an Off policy algorithm, which means it uses a different policy for exploring actions from the target policy of being optimal. Many of the amazing feats in RL over the past decade, such as Deep Q-Learning for Atari, or AlphaGo, Rainbow were rooted in these foundations. As stated in section 2.2.3 value function ,is a measure of the expected reward you can receive from any given state given an MDP and a policy describing which actions an agent takes in each state. Value iteration is a computational algorithm that provides a means of finding the optimal policy,$\pi*$. The algorithm works by iteratively determining the value of being in each state, assuming that the agent takes the best possible action in that state under the current estimate of the value function.Fitted value iteration algorithm are used for approximating the value function of a continuous state MDP.Unlike value iteration over discrete state of states fitted value iteration can not always converge.In order for certain dynamic programming algorithms (e.g. policy iteration, value iteration) converge to a unique fixed point Bellman equation was rewritten as operator.

2.3 Deep Reinforcement Learning and DQN

The success of a deep neural network in computer vision has introduced a new paradigm in learning from raw pixels. The goal of Deep Reinforcement Learning is to connect the reinforcement learning algorithm to a deep neural network that operates directly on RGB images and efficiently processes training data using stochastic gradient updates [Mni+13].
The use of label data and the assumption that the distribution of data is identical and independent throughout the deep learning training process makes it complex to use it directly in reinforcement learning algorithms, which must be able to learn from sparse,noisy,delayed reward and highly correlated data. To address these issues and successfully apply deep learning to reinforcement learning [Mni+13] used the experience replay mechanism [Mni+15] which randomly samples previous transitions and thereby smooths the distribution of training over many past behaviors. A convolutional neural network was used to learn successful control policies from raw video data in a complex reinforcement learning environment. The network is trained with a variant of Q-learning algorithm,with stochastic gradient descent to update weights.

This architecture was based on the Tesauro TD-Gammon architecture [17] that updates the parameters of the network that estimate the value function directly from

the sampled experience of the policy. Similarly to this approach, the online network in DQN uses a technique known as experience replay [18] where the agent's experiences at each time-step, $\rceil_t = (s_t, a_t, r_t, s_t + 1)$ is stored in a data set $\mathcal{D} = e1, ..., e_N$ pooled over many episodes in a replay memory. By drawing random samples from this pool of stored experiences, the Q-learning is updated.After performing the experience replay, the agent selects and executes an action according to a ϵ-greedy policy. The use of experience replay and target networks enables relatively stable learning of Q values, and led to super-human performance on several Atari games.
The advantage of using deep Q-learning over Q-learning includes allowing to have greater sample efficiency, reduced variance by randomizing the sample bias, and avoiding being stuck in local minimum. The drawbacks of deep Q-learning is that it only handle discrete, low-dimensional action spaces.

2.3.1 Extension of DQN

Although the initial architecture of deep Q learning introduced by [Mni+13] paves the way for the use of deep neural network in reinforcement learning, it comes with its own drawbacks;this leads to more studies and an improved DQN architecture.In this section, we will discuss these extensions of DQN architectures.

2.3.1.1 Double Deep Q-networks: DDQN

DQN suffer from overestimation bias due to due to the maximization step in optimisation function in Q-learning.Q-learning can overestimate actions that have been tried often and the estimations can be higher than any realistic optimistic estimate .Double Q-learning [19], addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation.Double Q-learning stores two Q-functions,The average of the two Q values for each action and then performed ϵ-greedy exploration with the resulting average Q values.It was successfully combined with DQN to reduce overestimations.
TODOS: DDQN target equation

2.3.1.2 Prioritized replay

The main use of replay buffer is to sample transitions with maximum probability.Both DQN and DDQN samples experiences uniformly.Prioritized replay [20] samples transitions using the maximum priority ,providing a bias towards recent transitions and stochastic transitions even when there is little left to learn about them.

2.3.1.3 Dueling Network

Dueling Network was designed for value based learning,this architecture separates the representation of sate-value and state-dependent action advantages without supervision[6].its consists of two streams that represents the value and advantage functions,while sharing a common convolutional feature learning module.This network has a single Q-learning network with two streams that replace DQN architecture[3].

$$Q(s, a; \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a; \theta, \alpha) \tag{2.36}$$

2.3.1.4 Multi-step learning

Previously stated extension of DQN have indicated that the use deep learning has enhanced the learning capability of Q-learning.The performance of Q-learning is still limited by greedy action selection after accumulating a single reward.An alternative approach was multi-step targets:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{\boldsymbol{a}_{j,t+N}} Q_{\phi'}(\boldsymbol{s}_{j,t+N}, \boldsymbol{a}_{j,t+N}) \tag{2.37}$$

A multi-step variant of DQN is then defined by minimizing the alternative loss[16],

$$R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\theta}^{-}(S_{t+n}, a') - q_\theta(S_t, A_t) \tag{2.38}$$

2.3.1.5 Noisy Nets

The one limitation of ϵ-greedy policy is many actions must be executed to collect the first reward.Noisy Nets proposed a noisy linear layer that combines a deterministic and noisy stream.Depending on the learning rate the network ignores to learn the noisy stream.

2.3.1.6 Integrated Agent:Rainbow

In the Rainbow architecture [**raibow**] several architecture changes included the one stated above where applied to DQN. Distributional loss was replaced by a multi-step variant. The target distribution was constructed by contracting the value distribution in St+n according to the cumulative discount, and shifting it by the truncated n-step discounted return. multi-step distributional loss with double Q-learning by using the greedy action in St+n selected according to the online network as the bootstrap action $a \cdot t + n$, and evaluating such action using the target network.

2.3.2 Deep autoencoders

Reinforcement learning requires learning from large high-dimensional image data-set.For example, In Atari games the environment is composed of images with 210 * 160 pixels and 128 color palette. Each image is made up of hundreds of pixels, so each data point has hundreds of dimensions. The manifold hypothesis states that real-world high-dimensional data actually consists of low-dimensional data that is embedded in the high-dimensional space. This is the motivation behind dimensionality reduction techniques, which try to take high-dimensional data and project it onto a lower-dimensional surface.
Autoencoders are a special kind of neural network used to perform dimensionality reduction. They act as an identity function, such that an auto encoder learns to

output whatever is the input.They are composed of two networks, an encoder e and a decoder d.
The encoder learns a non-linear transformation that projects the data from the original high-dimensional input space X to a lower-dimensional latent space Z. This is called latent vector $z = e(x)$. A latent vector is a low-dimensional representation of a data point that contains information about x. This is commonly known as latent space representation, it contains all the important information needed to represent raw data points.Auto encoders manipulates the "closeness" of data in the latent space.
A decoder learns a non-linear transformation d:Z→X that projects the latent vectors back into the original high-dimensional input space X. This transformation takes the latent vector and reconstruct the original input data :

$$z = e(x) \rightarrow \hat{x} = d(z) = d(e(x)) \tag{2.39}$$

The autoencoder is trained to minimize the difference between the input x and the reconstruction $\hat{x}$ using a kind of reconstruction loss.

In traditional autoencoders, the latent vector should be easily decoded back to the original image as a result the latent space z can become disjoint and non-continuous. Variational autoencoders try to solve this problem.
In variational autoencoders, inputs are mapped to a probability distribution over latent vectors, and a latent vector is then sampled from that distribution. As a result the decoder becomes more robust at decoding latent vectors.
Specifically, instead of mapping the input x to a latent vector $z = e(x)$, we instead map it to a mean vector $\mu(x)$ and a vector of standard deviations $\sigma(x)$. These parametrize a diagonal Gaussian distribution $\mathcal{N}(\mu, \sigma)$, from which we then sample a latent vector $z \sim \mathcal{N}(\mu, \sigma)$.
This is generally accomplished by replacing the last layer of a traditional autoencoder with two layers, each of which output $\mu(x)$ and $\sigma(x)$. An exponential activation is often added to $\sigma(x)$ to ensure the result is positive.
However, this does not completely solve the problem. There may still be gaps in the latent space because the outputted means may be significantly different and the standard deviations may be small. To reduce that, an auxiliary loss is added that penalizes the distribution $p(z|x)$ for being too far from the standard normal distribution $\mathcal{N}(\prime, \infty)$. This penalty term is the Kullback-Leibler(KL) divergence between $p(z|x)$ and $\mathcal{N}(\prime, \infty)$, which is given by$\mathbb{KL}\left(\mathcal{N}(\mu, \sigma) \parallel \mathcal{N}(0, 1)\right) = \sum_{x \in X}\left(\sigma^2 + \mu^2 - \log\sigma - \frac{1}{2}\right)$.This expression applies to two univariate Gaussian distributions by summing KL divergence for each dimension we are able to extend it to our diagonal Gaussian distributions.
This loss is useful for two reasons. First, we cannot train the encoder network by gradient descent without it, since gradients cannot flow through sampling (which is a non-differentiable operation). Second, by penalizing the KL divergence in this manner, we can encourage the latent vectors to occupy a more centralized and uniform location. In essence, we force the encoder to find latent vectors that approximately follow a standard Gaussian distribution that the decoder can then effectively decode.

2.4 Problems with RL

In previous studies presented in Chapter 3 we see some successful application of unsupervised learning techniques applied to improve the performance of the underlying RL algorithms, even though these and other studies conducted previously have shown remarkable results. RL comes with the following challenges.

One of the most difficult aspects of RL is learning efficiently with little data. The term "sample efficiency" refers to an algorithm that makes the most of a given sample. To put it another way, it's the amount of experience the algorithm has to gain during training in order to achieve efficient performance. The difficulty is that the RL system takes a long time to become efficient.

Neural networks are opaque black boxes whose workings are mysteries to even the creators. They are also increasing in size and complexity, backed by huge data sets, computing power and hours of training.This referred to as Reproducibility crisis, These factors make RL models very difficult to replicate.

Another major challenge in Rl is agents are trained in simulated environment ;In this environment they can fail and learn, but they do not have the opportunity to fail and learn in real-life scenarios. Usually, in real environments, the agent lacks the space to observe the environment well enough to use past training data to decide on a winning strategy. This also includes the reality gap, where the agent cannot gauge the difference between the learning simulation and the real world.

The reward technique discussed in the previous sections is not foolproof. Since the rewards are sparsely distributed in the environment, a possible issue is an agent not observing the situation enough to notice the reward signals and maximise specific actions. This also occurs when the environment cannot provide reward signals in time; for instance, in many situations, the agent receives a green flag only when it is close enough to the target.

Curiosity-driven methods are widely used to encourage the agent to explore the environment and learn to tackle tasks in it. The researchers in the paper 'Curiosity-driven exploration by self-supervised prediction' proposed an Intrinsic Curiosity Module (ICM) to support the agent in exploration and prompt it to choose actions based on reduced errors. Another approach is curriculum learning, where the agent is presented with various tasks in ascending order of complexity. This imitates the learning order of humans.

2.5 Unsupervised learning on images

?? TBD - citing papers rl papers cite. Point here is pretraining is good and stuff like contrastive loss is introduced.

2.6 Introduction to state learning learning

As discussed in the introduction, learning control from images is very desirable. Images, and observations in general, only implicitly provide information about the underlying state. Finding a good policy from observations, especially images, is much more difficult than finding a policy with direct state access because the state first needs to be inferred from those observations. Reinforcement learning algorithms can by themselves implicitly extract the relevant information from observations, but this at best results in much less sample-efficient training and at worst results in complete failure. Often a problem which a reinforcement learning algorithm can solve with direct state access, can not achieve any progress when provided only image observations.

It is clear from the previous section that amazing results were achieved in the field of computer vision. However, to leverage these results for the purposes of reinforcement learning, the methods in question need to be applied for state estimation. This is a drastically different problem than for example image classification. The key difference is that now dynamics need to be inferred. While neural network architectures like the convolutional neural network are able to achieve great successes in timeless problems, neural architectures like LSTMs aimed at learning from sequential data comparatively perform much worse. Results in problems such as video prediction or action classification leave much to be desired.

Having that said, the learning signal generated from for example image reconstruction loss is substantially stronger than the reward signal, especially in settings with sparse rewards where it is not present most of the time. Thus it stands to reason that somehow leveraging the learning signal from some computer vision method should aid the reinforcement learning process. One way to do this is to explicitly use such methods to learn a function which maps from observations to states and then use reinforcement learning methods these learned state representations. This approach is explored in this section, mainly with the help of the [**srloverview**] overview paper. In this section state representation learning for control in general is discussed. Importantly, this does not concern learning a model which can be used to achieve control through planning, although there are similarities between these approaches.

In general, representation learning algorithm are designed to learn abstract features that characterize data. In the simplest forms they include methods such as k nearest neighbors. In state representation learning (SRL) the learned features are of low dimension, evolve through time and are depended on actions of an agent. The last point is particularly important because in reinforcement learning, features that do not influence the agent and that can not be influenced by the agent are not relevant for the problem of optimally controlling the agent. Also, simply reducing the dimensionality of the input to a reinforcement learning agent results in a computationally easier learning problem, which can make a difference between the solution being feasible or infeasible. Ideally, state representation learning should be done in an without explicit supervision as it can then be done in tandem with the likewise unsupervised reinforcement learning.

While we assume that state-transitions have the Markov property, partial observability denies the possibility of having a one-to-one correspondence between each

observation and state — an object whose position is required may be occluded by another. Thus prior observations have affect the mapping to the current state. Images in particular also do not encode kinematic or dynamic information: to get that crucial information a sequence of images is required. Hence we define the SRL task as learning a representation $\tilde{\boldsymbol{s}}_t \in \tilde{\mathcal{S}}$ of dimension K with characteristics similar to those of true states $\boldsymbol{s}_t \in \mathcal{S}$. In particular, the representation is a mapping of the history of observation to the current state: $\tilde{\boldsymbol{s}}_t = \phi(\boldsymbol{o}_{1:t}$. Actions $\boldsymbol{a}_{1:t}$ and rewards $r_{1:t}$ can also be added to the parameters of ϕ. This can help in extracting only the information relevant for the agent and its task. Often the representation is learned by using the reconstruction loss; $\hat{\boldsymbol{o}}_t$ denotes the reconstruction of $\boldsymbol{o}_t$.
In the context of reinforcement learning, state representations should ideally have the following properties:

- have the Markov property
- be able to represent the current state well enough for policy improvement
- be able to generalize to unseen states with similar features
- be low dimensional

We now discuss different types of models and learning strategies which can be used to learn state representations.

2.6.1 Representation models in general

In general, representation learning refers to the process of learning a parametric mapping from raw input data domain to a feature vector or tensor, in the hope of capturing and extracting more abstract and useful concepts that can improve performance of downstream tasks. Often this includes dimensionality reduction. The goal of representation learning is for this mapping to meaningfully generalize well on new data. Before introducing types of representation models, we first need to define the characteristics a good representation of data needs to have in general. These principles and trade-offs between their relative priorities guide the model design. The following list summarizes different desirable characteristics:

1. *smoothness*: f s.t. $x \approx y$ implies $f(x) \approx f(y)$
2. *multiple explanatory factors* a.k.a. disentangling features
3. *semi-supervised learning*: for input Z and target Y, learning $P(X)$ helps learning $P(Y|X)$ because features of X help explain Y
4. *shared factors across tasks*: like previous point, but also works for different Ys
5. *manifolds*: probability mass concentrates in regions with much smaller dimensionality than data itself
6. *natural clustering*: different values of categorical variables are associated with separate manifolds.
7. *temporal and spatial coherence*: consecutive or spatially nearby observations thend to be associated with the same value of relevant categorical concepts or result in small surface move on the surface of the manifold
8. *sparsity*: could mean often many features are 0. could also be that the features are insensitive to small changes in x
9. *simplicity of factor dependencies*: ideally factors are related to each other linearly, or otherwise simply

The process of extracting representations from observations, or inferring latent variables in a probabilistic view of a dataset, is often called **inference**. There are **generative** and **discriminative** models.
Generative models learn representations by modelling the data distribution $p(\boldsymbol{x})$. Such a model can generate realistic examples. Evaluating the conditional distribution $p(y|\boldsymbol{x})$ it done via Bayes rule.
Discriminative models model the conditional distribution $p(y|\boldsymbol{x})$ directly. Discriminative modelling consists of first the inference that extracts latent variables $p(\boldsymbol{v}|\boldsymbol{x})$ which are then used to make downstream decision from those variables $p(y|\boldsymbol{v})$.
The benefit of discriminative models are that you don't have to go through an expensive process of learning $p(\boldsymbol{x})$. That's also harder to evaluate. This is especially evident if you just want a lower dimensional distribution. In the context of reinforcement learning, the model-based approach benefits from generative models as they can be used to generate predictions which can then be used for planning. In the model-free approach, both discriminative and generative models may be used as predictions are not used.

2.6.2 Generative models

2.6.2.1 Probabilistic models

From the probabilistic modeling perspective, feature learning can be interpreted as an attempt to recover a parsimonious set of latent random variables that describe a distribution over the observed data. $p(x, h)$ is the probabilistic model over the joint space of latent variables h and observed data x. Feature values are then the result of an inference process to determine the probability distribution of the latent variables given the data, i.e. $p(h|x)$, a.k.a posterior probability. Learning is the finding the parameters that (locally) maximize the regularized likelihood of the training data.

2.6.2.2 Directed graphical models

Directed latent factor models separately parametrize $p(x|h)$ and the prior $p(h)$ to construct $p(x, h) = p(x|h)p(h)$. They can explain away: a priori independent causes of an event can become nonindependent given the observation of the event. Can conceive them as cause models, where h activations cause the observed x, making h nonindependent. This makes recovering the posterior $p(h|x)$ intractable.

2.6.2.3 Directly learning a parametric map from input to representation

The posterior distribution becomes complicated quickly. Thus approximate inference becomes necessary, which is not ideal. Also, depending on the problem, one needs to derive feature vectors from the distribution. If we want deterministic feature values in the end, we might as well go ahead and use a nonprobabilistic feature learning paradigm. Doing so is particularly desirable for representations for model-free reinforcement learning algorithms: since the data distribution is not explicitly used to make plans, the stochasticity inherent in statistical modelling hinders the ability of the reinforcement learning algorithm to use those representations.

2.6.3 Discriminative models

In discriminative modelling the data distribution is not directly represented. Instead, it is implicit in the representation space. One way to learn discriminative models is through contrastive representation learning. Intuitively, it's learning by comparing. So instead of needing data labels y for datapoints $\boldsymbol{x}$, you need to define a similarity distribution which allows you to sample a positive input $\boldsymbol{x}^+ \sim p^+(\cdot|\boldsymbol{x})$ and a data distribution for a negative input $\boldsymbol{x}^- \sim p^-(\cdot|\boldsymbol{x})$, with respect to an input sample $\boldsymbol{x}$. "Similar" inputs should be mapped close together, and "dissimilar" samples should be mapped further away in the embedding space.
Let's explain how this would work with the example of image-based instance discrimination. The goal is to learn a representation by maximizing agreement of the encoded features (embeddings) between two differently augmented views of the same images, while simultaneously minimizing the agreement between different images. To avoid model maximizing agreement through low-level visual cues, views from the same image are generated through a series of strong image augmentation methods. Let $\mathcal{T}$ be a set of image transformation operations where $t, t' \sim \mathcal{T}$ are two different transformations sampled independently from $\mathcal{T}$. There transformations include ex. cropping, resizing, blurring, color distortion or perspective distortion. A $(\boldsymbol{x}_q, \boldsymbol{x}_k)$ pair of query and key views is positive when these 2 views are created with different transformations on the same image, i.e. $\boldsymbol{x}_q = t(\boldsymbol{x})$ and $\boldsymbol{x}_k = t'(\boldsymbol{x})$, and is negative otherwise. A feature encoder $e(\cdot)$ then extracts feature vectors from all augmented data samples $\boldsymbol{v} = e(\boldsymbol{x})$. This is usually ResNet, in which case $\boldsymbol{v} \in \mathcal{R}^d$ is the output of the average pooling layer. Each v is then fed into a projection head $h(\cdot)$ made up of a small multi-layer perceptron to obtain a metric embedding $\boldsymbol{z} = h(\boldsymbol{v})$, where $\boldsymbol{z} \in \mathcal{R}^{d'}$ with $d' < d$. All vectors are then normalized to be unit vectors. Then you take a batch of these metric embedding pairs $\{(\boldsymbol{z}_i, \boldsymbol{z}'_i)\}$, with $(\boldsymbol{z}_i, \boldsymbol{z}'_i)$ being the metric embeddings of $(\boldsymbol{x}_q, \boldsymbol{x}_k)$ of the same image are fed into the contrastive loss function which does what we said 3 times already. The general form of popular loss function such as InfoNCE and NT-Xent is:

$$\mathcal{L}_i = -\log \frac{\exp(\boldsymbol{z}_i^T \boldsymbol{z}'_i/\tau)}{\sum_{j=0}^{K} \exp(\boldsymbol{z}_i \cdot \boldsymbol{z}'_j)/\tau} \tag{2.40}$$

where τ is the temperature parameter. The sum is over one positive and K negative pairs in the same minibatch.

2.6.4 Common representation learning approaches

2.6.4.1 Deterministic autoencoders

Deterministic autoencoders are generative models. TODO: move text about them here.

$$h^{(t)} = f_\theta(x^{(t)}) \tag{2.41}$$

There's also the reconstruction $r = g_\theta(h)$, used for the reconstruction error $L(x, r)$ over the training examples. Autoencoder training boils down to finding θ which minimizes:

$$\mathcal{J}_{AE}(\theta) = \sum_t L(x^{(t)}, g_\theta(f_\theta(x^{(t)}))) \tag{2.42}$$

One can tie the weights between the encoder and the decoder (i.e. make the same ones, just reversed).

2.6.4.2 Variational autoencoders

TODO: delete half of this or add the other half Variational autoencoders marry graphical models and deep learning. The generative model is a Bayesian network of form $p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})$, or in the case of multiple stochastic layers, a hierarchy such as: $p(\boldsymbol{x}|\boldsymbol{z}_L)p(\boldsymbol{z}_L|\boldsymbol{z}_{L-1})\cdots p(\boldsymbol{z}_1|\boldsymbol{z}_0)$. Similarly, the recognition model is also a conditional Bayesian network of form $p(\boldsymbol{z}|\boldsymbol{x})$ which can also be a hierarchy of stochastic layers. Inside each conditional may be a deep neural network, e.g. $\boldsymbol{z}|\boldsymbol{x} \sim f(\boldsymbol{x}, \boldsymbol{\epsilon})$ with f being the neural network mapping and $\boldsymbol{\epsilon}$ a noise random variable. Its learning algorithm is a mix of classical (amortized, variational) expectation maximization, but with the reparametrization trick ends up backpropagating through the many layers of the deep neural networks embedded inside it.
We can parametrize conditional distributions with neural networks. VAEs in particular work with *directed* probabilistic models, also know as *probabilistic graphical models* (PGMs) or *Bayesian networks*. The joint distribution over the variables of such models factorizes as a product of prior and conditional distributions:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M) = \prod_{j=1}^{M} p_{\boldsymbol{\theta}}(\boldsymbol{x}_j|Pa(\boldsymbol{x}_j)) \tag{2.43}$$

where $Pa(\boldsymbol{x}_j)$ is the set of parent variables of node j in the directed graph. For root nodes the parents are an empty set, i.e. that distribution is unconditional. Before you'd parametrize each conditional distribution with ex. a linear model, and now we do it with neural networks:

$$\boldsymbol{\eta} = \text{NeuralNet}(Pa(\boldsymbol{x})) \tag{2.44}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|Pa(\boldsymbol{x})) = p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{\eta}) \tag{2.45}$$

To solve intractabilities, we introduce a parametric *inference model* $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. This model is called the *encoder* or *recognition model/* $\boldsymbol{\phi}$ are called the *variational parameters*. They are optimized s.t.:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \approx p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) \tag{2.46}$$

Like a DLVM, the inference model can be almost any directed graphical model:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_M|\boldsymbol{x}) = \prod_{j=1}^{M} q_{bm\phi(\boldsymbol{z}_j|Pa(\boldsymbol{z}_j),\boldsymbol{x})} \tag{2.47}$$

This can also be a neural network. In this case, parameters $\boldsymbol{\phi}$ include the weights and biases, ex.

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = \text{EncoderNeuralNet}_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{2.48}$$

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma})) \tag{2.49}$$

Typically, one encoder is used to perform posterior inference over all of the datapoints in the dataset. The strategy used in VAEs of sharing variational parameters across datapoints is also called *amortized variational inference*.

2.6.4.3 Deterministic autoencoder regularization

Autoencoders may be employed not only just to learn representations, but to perform additional auxiliary tasks. One such task is denoising: provided a noisified input at the encoder, the decoder outputs a denoisified image as output. Importantly, while this training process results in a denoising autoencoder, it also regularizes the autoencoder. Regularization not only helps with preventing overfitting, but also produces better representations as it encourages smoothness and spatial coherence of when learning. The same result can be accomplished by other data augmentation techniques like random cropping.
TODO: reword so that the spice flows.
Learning VAEs from data poses unanswered theoretical questions and considerable practical challenges. This work proposes a generative model that is simpler, deterministic, easier to train, while retaining some VAE advantages. Namely, the observation is that sampling a stochastic encoder in Gaussian VAE can be interpreted as injecting noise into the input of a deterministic decoder.
The encoder deterministically maps a data point $\boldsymbol{x}$ to the mean $\mu_\phi(\boldsymbol{x})$ and variance $\sigma_\phi(\boldsymbol{x})$ in the latent space. The input to D_θ is then the mean $\mu_\phi(\boldsymbol{x})$ augmented with Gaussian noise scaled by $\sigma_\phi(\boldsymbol{x})$ via the reparametrizing trick. Authors argue that this noise injection is a key factor in having a regularized decoder (noise injection as a mean to regularize neural networks is a well-known technique). Thus training the RAE involves minimizing the simplified loss:

$$\mathcal{L}_{\mathrm{RAE}} = \mathcal{L}_{\mathrm{REC}} + \beta\mathcal{L}_{\boldsymbol{z}}^{\mathrm{RAE}} + \lambda\mathcal{L}_{\mathrm{REG}} \tag{2.50}$$

where $\mathcal{L}_{\mathrm{REG}}$ represents the explicit regularizer for D_θ, and $\mathcal{L}_{\boldsymbol{z}}^{\mathrm{RAE}} = \frac{1}{2}||\boldsymbol{z}||_2^2$, which is equivalent to constraining the size of the learned latent space, which is needed to prevent unbounded optimization. One option for $\mathcal{L}_{\mathrm{REG}}$ is Tikhonov regularization since it is known to be related to the addition of low-magnitude input noise. In this framework this equates to $\mathcal{L}_{\mathrm{REG}} = \mathcal{L}_{L_2} = ||\theta||_2^2$. There's also the **gradient penalty** and **spectral normalization**.

2.6.5 Representation models for control

With representation in learning introduced in general, we can now introduce four different strategies for learning latent space models for control: the autoencoder, the forward model, the inverse model and the model with prior. These models refer to portions of the control problem they are modelling.[2] They can be both discriminative and generative models. In the figures below, the white nodes are inputs and the gray nodes are outputs. The dashed rectangles are fitted around variables with which the loss is calculated.

2.6.6 Autoencoder

The idea behind the autoencoder is to just learn a lower-dimensional embedding of the observation space. This should make the learning problem easier due to

[2]The term autoencoder is overloaded in this case.

the dimensionality reduction. The auto-encoder may be trained to denoise the observations by passing an observation with artificially added noise to the encoder, but then calculating the reconstruction loss on the image without the added noise. Formally this can be written as

$$\boldsymbol{s}_t = \phi(\boldsymbol{o}_t; \theta_\phi) \tag{2.51}$$
$$\hat{\boldsymbol{o}}_t = \phi^{-1}(\boldsymbol{s}_t; \theta_{\phi^{-1}}) \tag{2.52}$$

where θ_ϕ and $\theta_{\phi^{-1}}$ are the parameters learned for the encoder and decoder respectively.

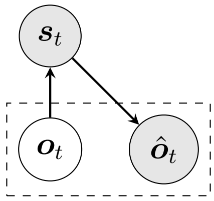


Figure 2.5: Auto-encoder: learned by reconstructing the observation (one-to-one). The observation is the input and the computed state is the vector at the auto-encoder's bottleneck layer, i.e. is the output of the encoder part of the auto-encoder network. The loss is calculated between the true observation and the reconstructing observation (which is obtained by passing the observation though both the encoder and the decoder).

2.6.7 Forward model

The auto-encoder does not encode dynamic information. Since that information is necessary for control, usually a few consequtive observations (or their embeddings) are stacked and passed to the reinforcement learning algorithm. This way the information about the dynamics is implicitly provided. While doing so works, it could be made more efficient by embedding the dynamic information as well. One way to achieve this is to trained a model to predict future state representations. A model can also be observations directly, of course provided that the network in question has a bottleneck layer from which the learned representations can be extracted. Since learning on sequential information is difficult and would also benefit from lowering the dimensionality, learning a forward model can be done in two steps: first, learning an auto-encoder to embed individual frames and then learning a predictive model in the embedded space. In the schematic we show the case where predictions are learned from embeddings because it is the structurally more complex scheme. Formally, we have

$$\hat{\tilde{\boldsymbol{s}}}_{t+1} = f(\tilde{\boldsymbol{s}}_t, \boldsymbol{a}_t; \theta_{\text{forward}}) \tag{2.53}$$

The forward model can be constrained to have linear transition between $\tilde{\boldsymbol{s}}_t$ and $\tilde{\boldsymbol{s}}_{t+1}$, thereby imposing simple linear dynamics in the learned state space. Depending on the problem, if this is done well enough, learning a control law can be avoided and instead schemes like model-predictive control can be employed.

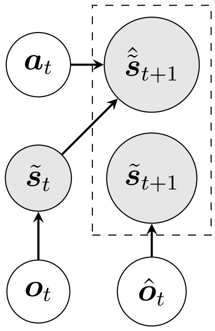


Figure 2.6: Forward model: predicting the future state from the state-action pair. The loss is computer from comparing the predicted state against the true next state (the states being the learned states). This can also be done directly by predicting the next observation and comparing against it.

2.6.8 Inverse model

The introducing predictions solves the problem of not embedding the dynamic information. However, not all information in the observation is relevant for control. Consider a computer game where images feature decorative backgrounds — those decorations are irrelevant for playing the game well. If the reconstruction loss is computed from entire observation, that information is also carried over into the embedded space. However, if the model is trained to predict actions, it is only incentivised to use information which the agent can affect. Thus, due to less information being required, the inverse model should produce a more compact embedding. Formally, we can write this as:

$$\hat{\boldsymbol{a}}_t = g(\tilde{\boldsymbol{s}}_t, \tilde{\boldsymbol{s}_{t+1}}; \theta_{\text{inverse}}) \tag{2.54}$$

If the inverse model is neural network, we can recover the embedding by discarding the last few layers and use their outputs to produce the embeddings.

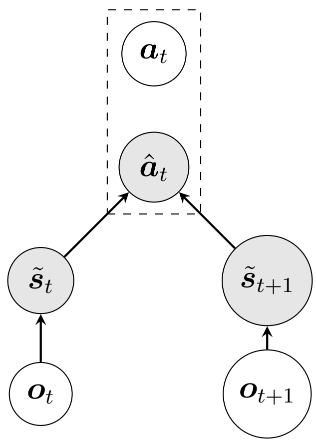


Figure 2.7: Inverse model: predicting the action between two consecutive states. The loss is computer from comparing the predicted action between two consecutive states against the true action that was taken by the agent between those two states. (the states being the learned states).

2.6.9 Using prior knowledge to constrain the state space

Of course, not everything need be learned in every problem. While in general hand-engineered features are worse than learned ones, there are other ways to provide prior knowledge to the learning system. For example, convolutional neural network by their architecture encode the fact that nearby pixels are related. In the SRL context we already mention the possibility of constraining the model to linear transitions, but there are other available techniques like for example constraining temporal continuity or the principle of causality. Furthermore, priors can be defined as additional objectives or loss functions. For example, additional loss can be provided if embeddings from consequtive observation are drastically different. This is called the slowness principle.

2.6.10 Using hybring objectives

The approaches outlined thus far can be combined into hybrid approaches. TODO: throw a reference or two from the overview paper you're going over, for example embed2control.

2.7 Model-based reinforcement learning

TODO Introduce just the idea for the sole purpose of showing why we aren't doing model-based reinforcement learning, but instead opting for model-free with state representation learning.

3
Related Work

As said in the introduction, the goal of the thesis is to use state representation learning to increase the efficiency and finals results of model-free reinforcement learning. We are now ready to discuss the specifics of our approach. Firstly, we limit ourselves to image observations and discrete action spaces. In particular, we limit ourselves to Atari57 games as they are common benchmarks in the field for discrete action spaces. As shall be seen in the following text, a lot of recent work in state-representation learning for model-free reinforcement learning has been done in robotics problems with continous action spaces, for example [Yar+19]. Importantly, since we are concerned with finding ways to make reinforcement learning more sample-efficient, we will be using only off-policy algorithms.

Secondly, we are particularly interested in the problem of simultaneous training of the state representations and the policy. The reason for this is that two-step training is often not available because not all state transitions can be observed beforehand. This state of affairs is the natural setting for problems where reinforcement learning is a good solution: the problems where exploration is necessary due to either the high complexity of the dynamics or unanticipatable events. Parallel training of the state representations and the policy necessitates instability in policy training due to the fact the state estimations change even for same observations as the state representation are learned. Hence, related work that focuses on solving or at least ameliorating this issue is of particular importance to our work.

Finally, we want our method to be robust not just in the sense that it works across a wide array of problems, but in the sense that it can be easily added to a variety of reinforcement learning algorithms to a positive effect. In other words, it should function as a module which can be easily added to new algorithms. Furthermore, it should work well with other improvements as those suggested in some of the following related work. To set the context, we begin with by discussing prior work in the Atari environment.

3.1 Reinforcement learning on Atari

Started with [Mni+13]. We already discussed [Hes+18]. Agent 57 [Bad+20] was the first deep RL agent that out performs the standard human benchmark on all 57 Atari games. It was built on on top of the Never Give Up (NGU) agent which utilizes a model-based approach. It combines two ideas: first, the curiosity-driven exploration, and second, distributed deep RL agents, in particular R2D2. The agent was able to balance the learning of different skills that are required to perform well

on such diverse set of games: exploration and exploitation and long-term credit assignment. In order to achieve this a neural network was trained to parameterize a family of policies ranging from very exploratory to purely exploitative, by using adaptive mechanism polices were prioritized throughout the training process.
However, if we convert simulated time to real time, these algorithms can take up to 16000 hours to reach their final performance. Since the goal is not really to solve Atari games, but to find useful general purpose algorithms, the work is still ongoing. The new proposed benchmark is Atari100K: solving the games with with only 100000 transitions. [1] This equates to 2.5 hours of real time.

3.2 Efforts in increasing efficiency in Atari

At the moment of writing, to the authors knowledge, the most efficient algorithm is [**ye2021mastering**] which is based on MuZero [**schrittwieser2020mastering**] and is a model-based algorithm. However, the title of the most efficient algorithm often switches between a model-based algorithm, a model-free algorithm with state representation learning or similar approaches. We will not discuss model-based approaches, but will discuss some alternative ones as their techniques illuminate the problem.
In particular, this concerns using data-augmentation as a means to directly regularize reinforcement learning. This was first employed in [Las+20] and expanded in [KYF20] and [Yar+21]. In [Las+20], the observations are augmented before they are passed to the policy networks. As we discussed in 2.6.4.3, data-augmentation or noisifying input data functions as strong regularization to feature extractors. The same applies to feature extraction trained just from reinforcement learning. In [KYF20], the same observation is copied and augmented several times. All of these augmented version of the same image are passed through the policy network. The results are then averaged and provide a better estimates than those obtained by a single pass of either non-modified or augmented observation. Thus we may conclude that data augmentation provides benefits to both representation and reinforcement learning.
We now turn to discussing works which utilize unsupervised state representation learning to increase reinforcement learning efficiency.

3.3 State representation learning for efficient model-free learning

Auxiliary losses may be used in a myriad of different ways to help reinforcement learning. TODO check first 2. In for example [She+16], [Jad+16] or [Pat+17] the same models used for state representations as used to help guide exploration. When, for example, a trained forward predictive model incurs large error, it is

[1]This equates to 400000 frames because the standard is to repeat each action 4 times: this makes learning easier, but also makes sense because humans do not need such small reaction time to solve the games.

reasonable to assume that this happened because a novel state has been encountered. This means that the loss can be interpreted as "intrinsic reward" and be added to "extrinsic reward" provided by the environment, yielding an algorithm which encourages exploration.
Of interest to us is the use of auxiliary losses for state representation learning. The specific loss and how it's used depends on the chosen state representation model. In the following subsections some common approaches will be explored.

3.3.1 Deterministic generative models

Perhaps the simplest model to be used for state representation learning on images is an autoencoder trained on reconstruction loss. Using an autoencoder ensures spatial coherence. This idea has been introduced in [**lange2010deep**]. It did not get traction in reinforcement learning more broadly due to the fact that when the autoencoder is updated, the state representation changes. Unlike regularizing noise which reduces overfitting and incentivizes learning of desirable properties, this noise is destructive. It hinders the ability of the reinforcement learning algorithm to associate states with their values due to the fact that what it is given different numbers as the same state through the course of autoencoder training. To solve this problem, regularization needs to used. In [Yar+19], this was solved by employing the regularizations introduced in [**ghosh2019variational**], which were already discussed in 2.6.4.3.
A mayor flaw of this approach is the fact that reconstructive loss incentivizes reconstruction of the entire image which contains information irrelevant to the agent. This pertains backgrounds and other object which do not effect state transitions. This does not mean that the obtained representations are not better than raw images, but that they could be made better. Knowing this, we still opted for this approach due to its simplicity and easy of debugging.

3.3.2 Stochastic generative models

As discussed in 2.6.1, stochastic models can be used to generate predictions which can be used to plan and thus be used in model-based reinforcement learning. However, this does not mean that they can not be used to bolster model-free learning. As discussed in 2.1.3, the formal setting for reinforcement learning is the Markov decision process — a stochastic setting. Of course, the degree of stochasticity depends on the problem at hand, but given even in a fully deterministic setting stochastic models can be used to deal with epistemic uncertainty. This is further exacerbated in case of partial observability. In [**slac**], (approximate) variational inference is used to formulate the entire algorithm objective. First, control is formulated as an inference problem and is thereby embedded into the MDP graphical model. From this single graphical model of the problem, the variational distribution of action-dependent state transitions can be factorized into a product of recognition, dynamics and policy terms. As with most approaches which employ stochastic generative models, a variational autoencoder is used to represent the latent (representation) space. It should be noted that without this deep integration with the problem, which enables learning state representation and policies under a single objective, the stochasticity of state representations would hurt the performance of the algorithm. A detailed analysis of this issue can be found in [Yar+19].

3.3.3 Discriminative models

Because we are ultimately only interested in state representations, generative models are not required. Thus it is natural to opt for a discriminative model. Discriminative models can be trained in different ways. In [LSA20], contrastive loss is employed. Another common choice, theoretically investigated in [**rakelly2021mutual**], (TODO:check these 2) is used in [**anand2019unsupervised**] and [**mazoure2020deep**] is to use mutual information. A particularly promising avenue is to learn discriminative representation models though bootstrapping as introduced in [**grill2020bootstrap**]. This has been employed to learn state representations in [**schwarzer2020data**], and in [**merckling2022exploratory**] where the losses have also been used to incentivize exploration.
These approaches ameliorate problems found in approaches discussed so far: that they avoid both the stochasticity of stochastic generative models and the unnecessary features picked up through reconstruction loss. Learning state representations through bootstrapping is particularly interesting because it is rather flexible with its formulation. In both papers mentioned, the bootstrapping happens through self-predictive loss and is aided with inverse dynamics loss. It would be interesting to integrate this more deeply with an appropriate reinforcement learning algorithm, akin to how stochastic generative models are integrated in the MDP in [**slac**].

3.3.4 Rainbow stuff

TODO: move this to background As stated in section 2.4.1.6 Integrated agent was built by integrating the previous extensions of DQN in to one agent. Prioritized replay and multi-step learning were the two most crucial components. compared to the previous benchmarks rainbow was able to improve both data efficiency and final performance. Although it potentially improved the performance of the original DQN algorithm; it also inherits the disadvantage of DQN, such as excessive memory usage, learning instability, and can only be applied to a discrete action space[**investigationontheDeepLearningFramework**].
The use of deep Auto-Encoder Neural Networks in Reinforcement Learning is till in its early stage. The application of auto-encoders in dimensionality reduction has played a major role in reducing training time and data efficiency [**auto-encoderforEfficientEmbedd**
Introducing auto encoders in batch RL resulted in learning from raw pixels with out previously augmenting the data manually or prepossessing [LR10]; this closes the existing gap between the between the high dimensionality of visual observations and the low dimensionality of state spaces. Deep convolutional encoders can learn good representations,they require large amounts of training data which makes there application in control systems limited. In the latest work [Haa+18] a successful RL variant called SAC + AE was introduced. Prior to this agent, two-step training was proposed by (Lange & Riedmiller, 2010; Munk et al., 2016; Higgins et al., 2017b;

Zhang et al., 2018a; Nair et al., 2018; Dwibedi et al., 2018) due to suboptimal policies the performance of these agents was poor.SAC + AE was designed using parrallel training using an off-policy algorithm and add an auxiliary task with an unsupervised objective low samole effieciency of most of the previously stated arcitectures presented under Section 2.3.1 was implemented. In addition to successfully combining autoencoders with model-free RL in the off-policy setting ;it was proved that SAC+AE bit the current model-based agents with noisy observations

3.4 Formulating our hypothesis

they're obvious and stated already, TODO this later.

4
Methods

4.1 Problems to be tackled

As stated previously, the goal is to learn effective state representations while training the policy. We indentify the following obstacles obstructing this goal:

1. to learn effective state representations which make the whole process more efficient, the learning algorithm needs to be incentivesed to embedded information relevant to the agent while discarding irrelevant information
2. as new state representations are learned, the old representations need to change as little as possible so as not to compromize what the policy has learned

4.2 Hypotheses

We posit the following hypotheses about features which make state representation effective:

1. representations which encode the dynamic information such as velocities will perform better than those which do not
2. representations which are learned solely on observations and not both observations and the agent's actions will yield worse results
3. learning procedures which try to model only the information relevant to the agent will perform even better
4. sharing information between the policy and the state representations will be benefitial

To test whether the learning state representations helps we compare the results against training the policy directly on observations. To test the whether parallel training of the policy and the state representations hinders the learning process, we compare the results against the alternative training procedure of first learning the state representations, fixing them and then training the policy on these representations, i.e. the two-step training procedure. Finally, to test our hypotheses about the properties of state representations which yield higher effectiveness, we compare results of differently designed and trained state representations. In particular, we use an autoencoder which only reconstructs the observations given to it as the baseline case. To test whether embedding dynamic information helps, we train the same autoencoder to be a forward predictor. Its inputs are a number of consequtive observations and it's output is the subsequent observation. We do this with and without also adding the agents actions as the input. To test whether making dynamic information more explicit helps, in another version we also pass the differences of each two consequtive frames. To test whether incentivising the state representation model to only focus on the dynamic information relevant to the agent helps, we train an inverse dynamics model and use the embeddings it generates as state representations. Finally, to test whether use nonlinear function approximators to representant policies are causing problems, we also train a tabular policy.

4.2.1 Enviroment and Preprocessing

We perform a comprehensive evaluation of our proposed method on the Arcade Learning Environment (Bellemare et al., 2013), which is composed of 57 Atari games. The challenge is to deploy a single algorithm and architecture, with a fixed set of hyper-parameters, to learn to play all the games given embedded latent space representation of the environment from auto encoder and game rewards. This environment is very demanding because it is both comprised of a large number of highly diverse games and the observations are high-dimensional.
Working with raw Atari frames, which are 210 x 160 pixel pictures with a 128 color palette, is computationally expensive, therefore we do a basic preprocessing step to reduce the input dimensionality. The raw frames are preprocessed by down sampling to a 110 x 84 picture and transforming their RGB representation to gray-scale. Cropping an 84 x 84 rectangle of the image that nearly captures the playing area yields the final input representation to encoder part of the auto encoder.

4.2.2 Deep Auto-encoder and Model Architecture

The first step in the process is to collect data to train the auto-encoder. we run a data collection module to generate the 100000 frame for each stated under the result section. The raw images are transformed to tensors and then trained a variational autoencoder with the objective of re-constructing the original image fed to the network. The auto-encoder was trained for maximum 100 epochs. When reconstructing an image with a network bottleneck, the encoder is forced to compress the original image to a smaller dimensional vector in the latent space.
[By compressing the raw pixels environment to smaller dimensional vector in the latent space we aim to improve the training time it takes for the integrated Rl agent developed by [16] and the shift in the latent space representation and its impact on the RL agent learning performance.We show this in more detail in the following sections.]

4.2.3 Training the RL Agent

In this paper we integrate auto encoder with integrated agent called Rainbow[12],the selection of this architecture was based it's ability to out perform all the previous architecture. our main focus was to experiment with the latent space representation of the environment.To address this we set up two experiment architectures one two step training and parallel training.

Two step Training: First, we train the auto encoder with the prepossessed data and the weights of the encoder are saved in file for training the agent. In this set

up the auto encoder is not updated such that we have a static representation of the environment.
second,we train the integrated agent with the results from the auto encoder.

Parallel Training: In this set up the agent is trained using the dynamic state representation from the encoder as we train both the auto encoder and the integrated agent in parallel. This introduces stochasticity of the environment to the training and in real life control system we believe that this will set a new paradigm on the use RL.

5

Results

state the Experiments

5.0.1 Two Step Training

5.0.2 Parallel Training

6
Conclusion

You may consider to instead divide this chapter into discussion of the results and a summary.

6.1 Discussion

6.2 Conclusion

Bibliography

[LR10] Sascha Lange and Martin Riedmiller. "Deep auto-encoder neural networks in reinforcement learning". In: *The 2010 international joint conference on neural networks (IJCNN)*. IEEE. 2010, pp. 1–8.

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: `http://arxiv.org/abs/1312.5602`.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: `http://dx.doi.org/10.1038/nature14236`.

[Jad+16] Max Jaderberg et al. "Reinforcement learning with unsupervised auxiliary tasks". In: *arXiv preprint arXiv:1611.05397* (2016).

[She+16] Evan Shelhamer et al. "Loss is its own reward: Self-supervision for reinforcement learning". In: *arXiv preprint arXiv:1612.07307* (2016).

[Pat+17] Deepak Pathak et al. "Curiosity-driven exploration by self-supervised prediction". In: *International conference on machine learning*. PMLR. 2017, pp. 2778–2787.

[Haa+18] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[Hes+18] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Yar+19] Denis Yarats et al. "Improving sample efficiency in model-free reinforcement learning from images". In: *arXiv preprint arXiv:1910.01741* (2019).

[Bad+20] Adrià Puigdomènech Badia et al. "Agent57: Outperforming the atari human benchmark". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 507–517.

[KYF20] Ilya Kostrikov, Denis Yarats, and Rob Fergus. "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels". In: *arXiv preprint arXiv:2004.13649* (2020).

[LSA20] Michael Laskin, Aravind Srinivas, and Pieter Abbeel. "Curl: Contrastive unsupervised representations for reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5639–5650.

[Las+20] Misha Laskin et al. "Reinforcement learning with augmented data". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 19884–19895.

[All+21] Arthur Allshire et al. "Laser: Learning a latent action space for efficient reinforcement learning". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 6650–6656.

[Yar+21] Denis Yarats et al. "Mastering visual continuous control: Improved data-augmented reinforcement learning". In: *arXiv preprint arXiv:2107.09645* (2021).

A

Appendix 1

(UNFINISED)
//UPDATE THE FORMAT LATER

1. Agent: It is an assumed entity which performs actions in an environment to gain some reward.
2. Environment (e): A scenario that an agent has to face.anything the agent cannot change arbitrarily is considered to be part of the environment.
3. Reward (R): An immediate return given to an agent when he or she performs specific action or task.
4. State (s): State refers to the current situation returned by the environment.
5. Policy (π): It is a strategy which applies by the agent to decide the next action based on the current state.
6. Value (V): It is expected long-term return with discount, as compared to the short-term reward.
7. Value Function: It specifies the value of a state that is the total amount of reward. It is an agent which should be expected beginning from that state.
8. Model of the environment: This mimics the behavior of the environment. It helps you to make inferences to be made and also determine how the environment will behave.
9. Model based methods: It is a method for solving reinforcement learning problems which use model-based methods.
10. Q value or action value (Q): Q value is quite similar to value. The only difference between the two is that it takes an additional parameter as a current action.