



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Improving sample-efficiency of model-free reinforcement learning algorithms on image inputs with representation learning

A systematic analysis of leveraging state representation learning for more efficient model-free reinforcement learning on image-based problems

Master's thesis in Computer science and engineering

Marko Guberina, Betelhem Dejene Desta

MASTER'S THESIS 2022

Improving sample-efficiency of model-free reinforcement learning algorithms on image inputs with representation learning

A systematic analysis of leveraging state representation learning for more efficient model-free reinforcement learning on image-based problems

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 on image inputs with representation learning

A systematic analysis of leveraging state representation learning for more efficient model-free reinforcement learning on image-based problems

Marko Guberina, Betelhem Dejene Desta

© Marko Guberina, Betelhem Dejene Desta, 2022.

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 L^AT_EX

Gothenburg, Sweden 2022

Improving sample-efficiency of model-free reinforcement learning algorithms on image inputs with representation learning

A systematic analysis of leveraging state representation

learning for more efficient model-free reinforcement learning on image-based problems

Marko Guberina, Betelhem Dejene Desta

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Reinforcement learning struggles to solve control tasks on directly on images. Performance on identical tasks with access to the underlying states is much better. One avenue to bridge the gap between the two is to leverage unsupervised learning as a means of learning state representations from images, thereby resulting in a better conditioned reinforcement learning problem. Through investigation of related work, characteristics of successful integration of unsupervised learning and reinforcement learning are identified. We hypothesize that joint training of state representations and policies result in higher sample-efficiency if adequate regularization is provided. We further hypothesize that representations which correlate more strongly with the underlying Markov decision process result in additional sample-efficiency. These hypotheses are tested through a simple deterministic generative representation learning model trained with image reconstruction loss and additional forward and inverse auxiliary losses. While our algorithm does not reach state-of-the-art performance, its modular implementation integrated in the reinforcement learning library Tianshou enables easy use to reinforcement learning practitioners, and thus also accelerates further research. In our tests we limited ourselves to Atari environments and primarily used Rainbow as the underlying reinforcement learning algorithm.

Keywords: sample-efficient reinforcement learning, state representation learning, unsupervised learning autoencoder

Acknowledgements

We give special thanks to our supervisor Divya Grover, examiner Claes Strannegård and everyone else at Chalmers who made this work possible.

Marko Guberina, Betelhem Dejene Desta, Gothenburg, August 2022

Contents

List of Figures	xiii
-----------------	------

List of Tables	xv
----------------	----

1	Introduction	1
1.1	What is reinforcement learning?	1
1.2	Why is reinforcement learning interesting?	2
1.3	Why learn from pixels?	2
1.4	Efforts to make reinforcement learning more efficient	3
1.4.1	Utilizing a world model	3
1.4.2	Utilizing state representations	4
1.5	Goal of the thesis	4
1.5.1	Hypothesis	4
1.5.2	Contributions	5
1.6	Outline	5
2	Reinforcement learning	7
2.1	Problem setting	7
2.1.1	Bandit problems	7
2.1.2	Markov Decision Processes	8
2.2	Key concepts in reinforcement learning	9
2.2.1	Policy	9
2.2.2	Goal of reinforcement learning	10
2.2.3	Value functions	10
2.3	Classes of reinforcement learning algorithms	11
2.3.1	Policy gradient algorithms	11
2.3.1.1	Baselines	12
2.3.1.2	Off-policy gradients	13
2.3.1.3	Advanced policy gradients	14
2.3.2	Actor-critic algorithms	14
2.3.3	Value function methods	16
2.3.3.1	Dynamic programming	16
2.3.3.2	Fitted value iteration	17
2.4	Deep reinforcement learning with Q-functions	18
2.4.1	Double Q-networks (DDQN)	20
2.4.2	Q-learning with multi-step returns	20

2.4.3	Prioritized replay	20
2.4.4	Dueling Network	21
2.4.5	Noisy Nets	21
2.4.6	Integrated Agent:Rainbow	21
2.5	Problems with RL	21
3	State representation learning	23
3.1	Representation learning in general	24
3.1.1	Generative models	25
3.1.1.1	Probabilistic models	25
3.1.1.2	Directed graphical models	25
3.1.1.3	Directly learning a parametric map from input to representation	25
3.1.2	Discriminative models	26
3.1.3	Common representation learning approaches	26
3.1.3.1	Deterministic autoencoders	26
3.1.3.2	Variational autoencoders	27
3.1.3.3	Deterministic autoencoder regularization	28
3.2	Representation models for control	28
3.2.1	Autoencoder	29
3.2.2	Forward model	30
3.2.3	Inverse model	30
3.2.4	Using prior knowledge to constrain the state space	31
3.2.5	Using hybring objectives	32
3.3	Model-based reinforcement learning	32
4	Related Work	33
4.1	Reinforcement learning on Atari	33
4.2	Efforts in increasing efficiency in Atari	34
4.3	State representation learning for efficient model-free learning	34
4.3.1	Deterministic generative models	35
4.3.2	Stochastic generative models	35
4.3.3	Discriminative models	36
5	Methods	37
5.1	Formulating our hypotheses	37
5.2	Our approach	38
5.3	Environment and Preprocessing	39
5.4	Module implementation	40
5.4.1	Tianshou	40
5.4.2	Trainer	41
5.4.3	Implementing state representation learning in Tianshou	41
5.5	Network architectures	42
5.6	Hyperparameters	43
6	Results	45
6.1	Effectiveness of pretrained encoders	45

6.2	Effect of varying network sizes on Rainbow	46
6.3	Effectiveness of continuous updates	46
6.4	Effectiveness of regularization	46
6.5	Effectiveness of forward prediction	46
7	Discussion	53
7.1	Differences between features and states	53
7.2	Reconstruction loss	54
7.3	Importance of representing dynamics	55
7.4	Not all regularization is the same	55
8	Conclusion	57
	Bibliography	59
A	Appendix 1	I

List of Figures

2.1	Conceptual schematic of reinforcement learning.	7
2.2	Schematic of a Markov chain.	8
2.3	Schematic of a Markov decision process.	8
2.4	Schematic of a partially observable Markov decision process.	9
2.5	Schematic of a Markov decision process with a policy π	10
3.1	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).	30
3.2	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.	31
3.3	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).	31
5.1	Schematic of the feature extractor neural network parameter space. Since unsupervised learning converges much faster, it constricts the search space for the features extracted through reinforcement learning. This constriction is enforced joint training of both unsupervised state representation and reinforcement learning.	38
5.2	Tianshou concepts	41
6.1	The graphs of potential efficiency gains. As can be observed from the graphs, if training is started using an encoder which was already trained using only reinforcement learning better results are achieved more quickly. Of course, this does not encompass all potential benefits — unsupervised learning could make the problem easier overall and thereby allow for both even faster learning and higher final scores.	48

6.2	Runs with Rainbow only, but with different encoder sizes. As can be seen, some games strongly benefit from having a larger encoder, while learning fails on others.	49
6.3	Effectiveness of parallel training. Overall the results are negative, some slightly and others strongly. Interestingly, using a pretrained encoder has a negative effect in this case. We suspect that this is due to the fixation on a particular local minimum caused by reconstruction loss. Importantly, the negative effect is present despite regularization maintaining a fixed latent space over a relatively large number of epochs (5-10). Results tend to be worse in games where MSE loss is less suitable and in games with larger visual complexity.	50
6.4	Comparison of using and L2 latent space regularization and data augmentation in parallel training. L2 regularization helps overall because it helps fix the latent representations in place, thereby stabilizing the overall training process. Data augmentation on the other hand clearly hurt across the board.	51
6.5	Comparison between latent representation trained to compress the current frames and those trained to perform one-step forward predictions. Despite having a significantly higher reconstruction error, encoder trained to perform forward prediction perform better.	52

List of Tables

5.1	Table of reinforcement learning related hyperparameters.	43
-----	--	----

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 through 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 2, 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 viewed 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 learn 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 reinforce-

ment 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

While there is case to be made that reinforcement learning is a step toward artificial general intelligence, there are also more immediate applications. Due to generality of reinforcement learning and to advances in computer hardware, reinforcement learning offers a promising avenue toward solving decision and control problems which have not been solved through other, more direct methods. One of these is the problem of robotic grasping. Humans and other animals have an intuitive understanding of physics which they leverage for object manipulation. On the other hand, to program a robot to do the same, exact physics equations need to be provided so that the robot's actions may be calculated. Owing to the complexity of contact dynamics and the inability to precisely measure the points of contact, this is often impossible to do. By learning through trial and error and by leveraging the strong interpolation capabilities of neural networks, such an "intuition" may be learned. Furthermore, traditional optimization methods require rich objective functions at every iteration step, while reinforcement learning can handle "sparse" rewards — objective functions which equate to 0 at nearly all points of the domain.

1.3 Why learn from pixels?

Recent success in reinforcement learning 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. 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.

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

The problem is that pixel-based observations are much higher-dimensional than the actual states. This makes the learning problem dramatically more difficult, both because of its higher dimensionality, but also because it adds the state inference problem on top of control problem. In a lot of cases a problem which reinforcement learning algorithms are able to solve with direct state access is unsolvable with only pixel-based observations. In the cases where it is possible, the training time is much longer because much more samples are required. This is problematic because reinforcement learning is rather inefficient as it is. The high number of required samples in particular prohibits its use outside of simulated environments, while learning on agents in the real world is the ultimate goal in most practical applications.

1.4 Efforts to make reinforcement learning more efficient

1.4.1 Utilizing a world model

An important classification of reinforcement learning algorithms 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. Thus, they learn the dynamics of the environment and use that knowledge to choose actions which lead the agent to states with high reward. Both 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 results in 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.

Unfortunately, the model-based 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.

1.4.2 Utilizing state representations

In the case of image-based observations, an alternative to using model-based methods is to additionally deal only with the problem of extracting states from images. This can be done by using auxiliary unsupervised representation learning goals. More concretely, the idea is to utilize learning signals other than the reward signal to make the model-free learning more sample-efficient. In particular, this amounts to learning a latent representation of the environment, i.e. finding a lower-dimensional embedding of the observations, and learning a policy in this space.

The benefits of this approach are two-fold:

- It is known that, in general, lower-dimensional optimization problems are easier to solve than higher-dimensional ones.
- Empirical findings show that when reinforcement algorithms have direct state access, they learn much faster and often achieve better final results.

Since inferring states from observations is not directly related to the reward, unsupervised learning techniques should aid in feature extraction and thus make learning more sample-efficient. This is the approach investigated in this thesis. Since the goal is not to learn the dynamics of the environment, but simply to find an equivalent, but lower-dimensional representation of it, it is this approach should not suffer from the problems faced by model-based approaches.

1.5 Goal of the thesis

Given the previous discussion, the goal of the thesis may be presented: the idea is to investigate how unsupervised learning techniques can be combined with model-free algorithms in order to increase their sample-efficiency. To make this a concrete and manageable goal, we constrain ourselves to the problem of learning from images and to problems with discrete action spaces 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 Atari-57 games [Bel+13]. Of course, we are not the first to suggest such an approach. An overview of the field is provided in 4.

1.5.1 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 3.2, 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 hypotheses based on those properties and perform tests on a simple implementation. Here we offer our hypotheses:

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.

1.5.2 Contributions

As already mentioned, our main goal is not to produce a state-of-the-art algorithm, but rather to find and investigate 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 hypotheses which illuminate the problem and pave the way 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.

1.6 Outline

The rest of this text is organized as follows. We begin by describing the basics of reinforcement learning in 2. Here the problem setting and basic concepts are covered. The main classes of reinforcement learning are introduced in 2.3. Having the basics of reinforcement learning established, in 2.4 we introduce the reinforcement

learning algorithm we use in our implementation and discuss common reinforcement learning problems in 2.5. Following the reinforcement learning discussion, we turn our attention to unsupervised representation learning on images in 3.1 and discuss common state representation learning approaches for control problems in 3.2. Following the background, discuss related work in 4. 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 5.1 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 hypotheses.

2

Reinforcement learning

2.1 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 approaches: 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

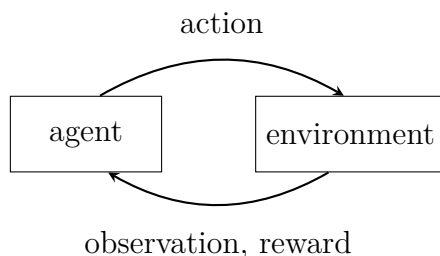


Figure 2.1: Conceptual schematic of reinforcement learning.

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 help introduce reinforcement learning formally, first the simplest possible problem to which reinforcement learning is the best solution is described.

2.1.1 Bandit problems

Consider the following 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 agent 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. However, the agent does not have access to that information and hence it can not effectively **exploit** the environment to obtain the highest rewards. Instead, it is forced to **explore** the environment in order to learn the probability distribution of the reward. In this problem, this amounts to pulling different levers and recording the received payoffs.

The key issue is thus to balance exploitation and exploration. It is compounded by the fact that the agent is given only a finite amount of time, or a finite number of lever pulls in this case. While one could leverage Bayesian statistics to construct an optimal solution to this simplest formulation of the bandit problem, this quickly becomes intractable as the complexity of the problem is increased. Namely, the described problem is stationary, in the sense that past actions do not influence the state of the world: the slot machines do not change as different levers are pulled. Said another way, the k -bandit problem has a single **state**. This is of course not the case in most problems of interest. To model the agent's effect on the environment, additional structure needs to be introduced. This is done in the following sections.

2.1.2 Markov Decision Processes

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 2.2.

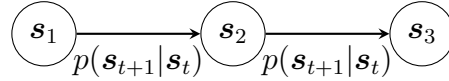


Figure 2.2: Schematic of a Markov chain.

Formally, a Markov chain \mathcal{M} is defined by its state space \mathcal{S} with discrete or continuous state $\mathbf{s} \in \mathcal{S}$ and the transition operator \mathcal{T} . The notation \mathbf{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 $p(\mathbf{s}_{t+1}|\mathbf{s}_t)$, let $\mu_{t,i} = p(\mathbf{s}_t = i)$ and $\mathcal{T}_{i,j} = p(\mathbf{s}_{t+1} = i|\mathbf{s}_t = j)$. Then $\vec{\mu}_t$ is a vector of probabilities and $\vec{\mu}_{t+1} = \mathcal{T}\vec{\mu}_t$. Importantly, \mathcal{T} is linear.

To model the agent's actions, actions are added as priors to state transition probabilities in the Markov chain. With this and the definition of the reward function, a Markov decision process is constructed. It's schematic can be seen in 2.3.

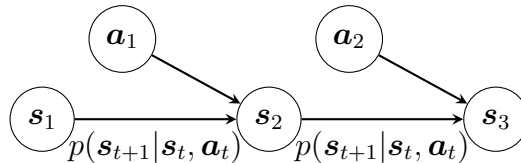


Figure 2.3: Schematic of a Markov decision process.

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 $\mathbf{a} \in \mathcal{A}$ 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 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 $\mathbf{o} \in \mathcal{O}$ denote the discrete or continuous observations and the emission probability \mathcal{E} which describes the probability $p(\mathbf{o}_t | \mathbf{s}_t)$ of getting the observation \mathbf{o}_t when in state \mathbf{s}_t . The schematic can be seen in 2.4.

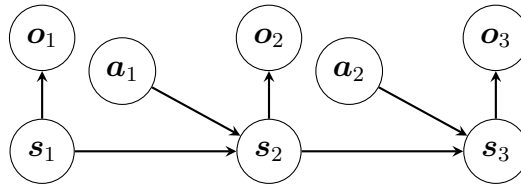


Figure 2.4: Schematic of a partially observable Markov decision process.

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 explicitly referenced in sketches and written out in the equations (most often using just the Markov decision process).

2.2 Key concepts in reinforcement learning

2.2.1 Policy

With the problem space being formally defined, definitions which will allow the construction of a reinforcement learning algorithm may be introduced. The reinforcement learning problem can be defined in finite or infinite time horizons. Different environments usually naturally fall in either category. In order 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 2.5.

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 , the notation π_θ is used.

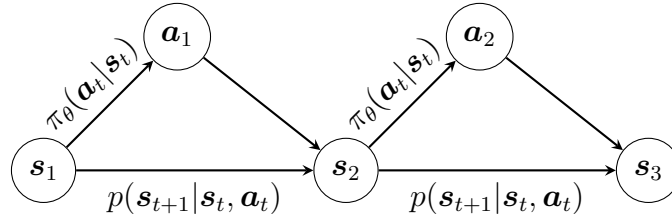


Figure 2.5: Schematic of a Markov decision process with a policy π .

2.2.2 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, the probability of observing a trajectory of states and actions can be written as:

$$\underbrace{p_\theta(s_1, a_1, \dots, s_T, a_T)}_{p_\theta(\tau)} = p(s_1) \prod_{t=1}^T \underbrace{\pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)}_{\text{Markov chain on } (s, a)} \quad (2.1)$$

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

$$p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p((s_{t+1}|(s_t, a_t))\pi_\theta(a_{t+1}|s_{t+1}) \quad (2.2)$$

With this, a formal definition of the goal of reinforcement learning can be given. It is to find policy parameters θ^* such that:

$$\theta^* = \operatorname{argmax}_{\theta} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] \quad (2.3)$$

$$= \operatorname{argmax}_{\theta} \sum_t^T \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)} [r(s_t, a_t)] \quad (2.4)$$

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 plays a role in modelling because usually it makes sense to value immediate rewards more. It is important to note that the *expected* sum of rewards is maximized. This makes the goal a smooth and differentiable function of the parameters, which means gradient descent can be employed 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 2.3.1, while additional concepts required by other classes of algorithms will be introduced in the following subsection.

2.2.3 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 :

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T \mathbb{E}_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t] \quad (2.5)$$

thus denoting the expected total reward from taking \mathbf{a}_t in \mathbf{s}_t . **Value functions** map states to the expected sum of rewards under policy π_θ :

$$V^\pi(\mathbf{s}_t) = \sum_{t'=t}^T \mathbb{E}_{\pi_\theta} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t] \quad (2.6)$$

The connection between the two is the following:

$$V^\pi(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi(\mathbf{s}_t, \mathbf{a}_t)} [Q^\pi(\mathbf{s}_t, \mathbf{a}_t)] \quad (2.7)$$

With these definitions, the reinforcement learning objective may also be written as:

$$\mathbb{E}_{\mathbf{s}_1 \sim p(\mathbf{s}_1)} [V^\pi(\mathbf{s}_1)] \quad (2.8)$$

2.3 Classes of reinforcement learning algorithms

2.3.1 Policy gradient algorithms

Policy gradients are derived by directly solving for the reinforcement learning objective using gradient descent, where the derivative is taken with respect to the policy parameters. To do so, the reinforcement learning objective needs to be evaluated. To make this a bit easier to follow, we introduce a notational shorthand:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \underbrace{\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]}_{J(\theta)} \quad (2.9)$$

$J(\theta)$ is estimated by making rollouts from the policy. Simply put, the agent collects experience under the current policy, and the average return is used as the estimate. In the equation below i is the sample index. i, t denotes the t^{th} timestep in the i^{th} sample:

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

Simplifying the notation further, we get:

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

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

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau \quad (2.12)$$

Since the goal of this text is just to introduce the necessary concepts and algorithms, the derivation(s) will be omitted. We encourage the interested reader to consult the literature [SB18; Lev21] to find more information.

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:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim p_{\theta}(\tau)} \left[\left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \right) \left(\sum_{t=1}^T r(\mathbf{s}_t, \mathbf{a}_t) \right) \right] \quad (2.13)$$

It can be evaluated through sampling:

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

The estimated gradient can be used to perform gradient ascent. This is the backbone of the REINFORCE algorithm, also known as “vanilla policy gradient”:

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t)$ by running the policy
2. use the samples to estimate the gradient of the objective:
 $\nabla_{\theta} J(\theta) \approx \sum_i \left(\sum_t \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$
3. update the policy function by performing a step of gradient ascent:
 $\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta)$

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.

2.3.1.1 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:

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

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

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 in 2.3.2.

2.3.1.2 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 as it would not produce those rollouts:

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

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:

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

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

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

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

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.

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

To get this equation, the factor $\frac{\pi_{\theta'}(\mathbf{s}_{i,t})}{\pi_{\theta}(\mathbf{s}_{i,t})}$ had to be 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} . The justification for this can be found in the previously referenced literature. What is important in the context of the thesis is that we will be interested only in off-policy methods as they are inherently more sample-efficient and our goal is to increase the sample-efficiency of reinforcement learning.

2.3.1.3 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.

2.3.2 Actor-critic algorithms

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

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

Simply put, in the policy gradient method a single-run Monte-Carlo (MC) estimate is used to estimate the return. This causes high variance, while incurring no bias. Another option is to try to estimate the full expectation $\hat{Q}_{i,t} \approx \sum_{t'=t}^T \mathbb{E}_{\pi_{\theta}} [r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t]$. 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 the “reward to go” estimator can generalize between states, it will be able to produce 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**. If the correct Q-function was available (i.e. not the estimate, but the actual values), the policy gradient estimate could be improved by using it both to estimate the return and as a baseline:

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

$$b_t = \frac{1}{N} \sum_i Q(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (2.25)$$

However, having a baseline that depends on actions leads to bias. Thus a state-dependent baseline is employed:

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

Since the value function 2.6 gives the expected return of the average action, it is possible to calculate how much better a certain action is by subtracting its Q-value 2.5 for the value function. The result is called the **advantage function**:

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

¹In 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.

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

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

The value function can be estimated through samples:

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

After collecting many such samples

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

the value function can be fitted through supervised regression with the following loss:

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

This process can be sped up with bootstrapped estimates:

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

This will further reduce variance, but again increase bias.

Fortunately, the trade-off between bias and variance can be tuned. 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 be used:

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

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):

$$\hat{A}_{GAE}^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{n=1}^{\infty} (\gamma\lambda)^{n-1} r(\mathbf{s}_{t+n}, \mathbf{a}_{t+n}) + \gamma \hat{V}_\theta^\pi(\mathbf{s}_{t+1}) - \hat{V}_\theta^\pi(\mathbf{s}_t) \quad (2.34)$$

where the factor λ controls the weight of future values.

Combining this into an iterative algorithm, and fixing the issues of naive implementations results in the following algorithm:

Actor-critic algorithm template

1. take action $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{s})$, observe transition $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$ and store it in the replay buffer \mathcal{R}
2. sample a batch $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ from buffer \mathcal{R}
3. update the Q-value estimator \hat{Q}_θ^π by using the target:
 $y_i = r_i + \gamma \hat{Q}_\theta^\pi(\mathbf{s}'_i, \mathbf{a}'_i) \forall \mathbf{s}_i, \mathbf{a}_i$
4. compute the policy gradient estimate with:
 $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(\mathbf{a}_i^\pi | \mathbf{s}_i) \hat{Q}^\pi(\mathbf{s}_i, \mathbf{a}_i^\pi)$, where $\mathbf{a}_i^\pi \sim \pi_\theta(\mathbf{a}|\mathbf{s}_i)$
5. update the policy function by performing a gradient step:
 $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

2.3.3 Value function methods

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

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

which would yield the highest expected return. In other words, if the advantage function is known, the policy would be reduced to the argmax operation. This approach has roots in dynamic programming.

2.3.3.1 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. 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 Markov decision process is assumed. Dynamic programming can be applied to continuous problems as well, but exact solution exist only in special cases.

For brevity, dynamic programming will not be fully introduced here. The interested reader is referred to [SB18]. Here only the broad idea will be introduced in order to make value iteration intuitive in its own right. A problem is said to have optimal substructure if an optimal solution to it can be constructed from optimal solutions of its subproblems. Value functions have this property: for a single well-defined problem, nearby optimal states are not related to optimality of distant states. In other words, the value of one state is related only to the value of states to which there are transitions from it. This principle is used to derive the **Bellman equation**. In problem with a finite number of states, it can be shown that iterating between evaluating the value function with the Bellman equation and updating the policy based on the value function leads to the optimal value function and policy.

In particular, the bootstrap update for the value function is:

$$V^\pi(\mathbf{s}) \leftarrow E_{\mathbf{a} \sim \pi(\mathbf{a}|\mathbf{s})} [r(\mathbf{s}, \mathbf{a}) + \gamma E_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{a}, \mathbf{s})} [V^\pi(\mathbf{s}')]] \quad (2.36)$$

where $V^\pi(\mathbf{s}')$ is the current estimate (initially set to whatever). With 2.35 we can construct the policy iteration algorithm:

Policy iteration

1. evaluate $V^\pi(\mathbf{s})$ with 2.36
2. set $\pi \leftarrow \pi'$

Upon inspection of the argmax in the advantage function, by skipping the policy update, this produce can be further simplified into the **value iteration** algorithm: ²

Value iteration

1. set $Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E[V(\mathbf{s}')]]$
2. set $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$

2.3.3.2 Fitted value iteration

As mentioned, and as should be clear from the equations, 2.3.3.1 works only on problems with a finite number of states. While real-valued problems can be partitioned into discrete ones and limited in scope, for most interesting problems this results in an intractable algorithm. However, function approximation, in particular nonlinear function approximation through neural networks can greatly bolster the capacity of value iteration. Another benefit of such an approach is that it is naturally adaptable to being off-policy. This is because after collecting samples the goal is to fit the value function, namely the Q-function, to the gathered data. In particular, the algorithm template is: ³

Fitted Q-iteration algorithm

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using a policy based on the value function
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$

By itself, this algorithm does not encourage exploration. This is usually fixed by using the **epsilon-greedy** policy:

$$\pi(\mathbf{a}_t | \mathbf{s}_t) = \begin{cases} 1 - \epsilon & \text{if } \mathbf{a}_t = \operatorname{argmax}_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ \frac{\epsilon}{|\mathcal{A}| - 1} & \text{otherwise} \end{cases} \quad (2.37)$$

In this case ϵ is often set to be large in the beginning of training and is decreased over time.

Unfortunately, by using nonlinear function approximators for value functions the convergence guarantees from the finite setting are lost. To see why, we first introduce the Bellman operator:

$$\mathcal{B} : \mathcal{B}V = \max_{\mathbf{a}} r_{\mathbf{a}} + \gamma \mathcal{T}_{\mathbf{a}} V \quad (2.38)$$

where $r_{\mathbf{a}}$ is the stacked vector of rewards of all states for action \mathbf{a} , and $\mathcal{T}_{\mathbf{a}, i, j} = p(\mathbf{s}' = i | \mathbf{s} = j, \mathbf{a})$ is the matrix of transitions corresponding to the action \mathbf{a} . V^* is now a *fixed point* of \mathcal{B} , meaning that if it is recovered the optimal policy is obtained:

$$V^*(\mathbf{s}) = \max_{\mathbf{a}} r(\mathbf{s}, \mathbf{a}) + \gamma E[V^*(\mathbf{s}')], \text{ so } V^* = \mathcal{B}V^* \quad (2.39)$$

²Here the notation was simplified for readability.

³A particular implementation called DQN will be discussed later.

It's possible to show that V^* always exists, is unique and corresponds to the optimal policy. This is because it can be proven that \mathcal{B} is a *contraction*. This means that for any V, \bar{V} :

$$\|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty \leq \underbrace{\gamma}_{\text{gap always gets smaller by } \gamma \text{ w.r.t. } \infty\text{-norm}} \|V - \bar{V}\|_\infty \quad (2.40)$$

However, if nonlinear function approximation is used the situation changes. Namely, in the second step of 2.3.3.1:

$$V' \leftarrow \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(\mathbf{s}) - (\mathcal{B}V)(\mathbf{s})\|^2 \quad (2.41)$$

where Ω is the hypothesis space (ex. the space of all weights of employed neural network architectures). V' will be a projection of $\mathcal{B}V$ back to Ω . Let us introduce an operator for this projection:

$$\Pi : \Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(\mathbf{s}) - V(\mathbf{s})\|^2 \quad (2.42)$$

So the fitter value iteration algorithm is:

1. $V \leftarrow \Pi \mathcal{B}V$

and here \mathcal{B} is a contraction (w.r.t. ∞ -norm (“max” norm)), Π is a contraction w.r.t. l_2 -norm (Euclidean distance), but $\Pi \mathcal{B}$ is not a contraction of any kind! The same holds for fitted Q-iteration, but we withhold further analysis for sake of brevity. Thus fitted value iteration does not converge. Additionally, it is interesting that Q-learning is not in fact a derivative of the Q-function:

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \underbrace{\left[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}_i', \mathbf{a}_i') \right]}_{\text{no gradient through target value}} \right) \quad (2.43)$$

As can be seen, the target Q-values themselves depend on Q-values. Despite these sad theoretical results, fitted value iteration algorithms work surprisingly well in practise. In fact, this work uses fitted value iteration algorithms, namely DQN and Rainbow, which will be introduced next.

2.4 Deep reinforcement learning with Q-functions

In this section we finally introduce reinforcement learning algorithms which are built upon in the thesis. As the section title suggests, these algorithms utilize (deep) neural networks. While simple feed-forward networks with 2-3 hidden layers suffice for many control problems, additional preceding convolutional layers are required for problems with image inputs. For problems with long-term temporal dependencies, networks with recurrent layers are often used. We are particularly concerned with the DQN algorithm [Mni+13] and a series of improvements to it which culminated in their combination which is named the Rainbow algorithm. There are multiple

reasons why Rainbow was chosen. Firstly, it and other fitted value iteration algorithms are the best suited ones for problems with discrete action spaces to which we restricted ourselves. Secondly, leveraging state representation learning is most suitable to fitted value iteration algorithms and critics in actor-critic algorithms because they are more stable than policy gradients. As shall be discussed later, stability of state representation learning and of reinforcement learning is necessary for successful simultaneous learning of both.

The on-policy version of Q-learning suffers from overfitting to local transitions. The problem is further exacerbated because the target values change through time. While this problem can be ameliorated by using parallel workers, a better solution is to use a **replay buffer**. A replay buffer is simply an array which keeps track of sampled transitions. Minibatch updates can be constructed by sampling independent and identically distributed transitions from the buffer, thereby remove the problem of overfitting to local transitions. Continuous replacement of old samples with new ones ensures continual learning as new transitions are collected (the initial epsilon-greedy policy often can not sample all transitions because it can not reach states with high rewards). Q-learning then becomes:

Q-learning with a replay buffer

repeat until a satisfactory result is reached:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to \mathcal{B}

repeat K times:

2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ in i.i.d. fashion from \mathcal{B}

3. update network weights:

$$\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) (Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}')])$$

This algorithm still suffers from the fact that Q-learning is not gradient descent and that it tries to converge to a moving target (which is local overfitting). One technique to ameliorate this is to use **target networks**. The idea is to collect transitions with one network, called the target network, and apply updates to another. For practical reasons, the two are the same network, where the target network is periodically updated with the weights of the other network. This can be done by simply copying the weights or doing it more smoothly via ex Polyak averaging. With this we get the classic DQN algorithm [Mni+13]:

“Classic” DQN

1. take some action \mathbf{a}_i , observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ and add it to \mathcal{B}
2. sample a mini-batch $(\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j)$ from \mathcal{B} uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}')$ using the *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j) (Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ' : copy ϕ every N steps

Here the other network is updated at every sample, i.e. $K = 1$ because that was the setting in the original paper, but of course it could have any other value.

2.4.1 Double Q-networks (DDQN)

Empiric evidence shows that Q-networks trained with DQN overestimate returns, i.e. the true Q-function. The reason for this is the following one:

$$\text{target value } y_j = r_j + \gamma \underbrace{\max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)}_{\text{herein lies the problem}} \quad (2.44)$$

The explanation goes as follows. Consider two random variables, X_1 and X_2 , and let them represent a true value obscured by some noise. Provably,

$$E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2]) \quad (2.45)$$

The relation to Q-learning is the following. If we imagine that $Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ is not perfect because it has added noise, we get exactly the situation in the inequality — the max over the actions and the expectation over it will lead to systematic overestimation. Thus $\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}')$ *overestimates* the next value. Note that $\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}') = Q_{\phi'}(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}', \mathbf{a}'))$. If noise in the action selection mechanism and noise in the value evaluation mechanism are decorrelated, the problem will go away. In other words, getting both actions and values from $Q_{\phi'}$ needs to be avoided. This can be done by utilizing another network. For practical reasons, the target network and the updating network can serve this purpose. Although, not theoretically ideal, this solution works well in practise. Thus the only difference to the 2.4 is to change the error calculation in step 3 into

$$y = r + \gamma Q_{\phi'}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi}(\mathbf{s}', \mathbf{a}')\right) \quad (2.46)$$

2.4.2 Q-learning with multi-step returns

As discussed in 2.3.2, n-step returns 2.33 offer a better balance better bias and variance than either single-step bootstrap estimates or Monte-Carlo estimates. They can of course be used in Q-learning as well:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{\mathbf{a}_{j,t+N}} Q_{\phi'}(\mathbf{s}_{j,t+N}, \mathbf{a}_{j,t+N}) \quad (2.47)$$

2.4.3 Prioritized replay

As mentioned in 2.4, the samples for minibatches are sampled uniformly from the buffer. Depending on the size of the buffer, this slows down progress as it takes time for newer transitions to be incorporated into Q-function estimation. Alternatively, newer samples could be given priority by being sampled with a higher probability, or, better still, also proportionally to the size of the TD error:

$$p_t \propto \left| r_{t+1} + \gamma_{t+1} \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}_{t+1}, \mathbf{a}') - Q_{\phi}(\mathbf{s}_t, \mathbf{a}_t) \right|^\omega \quad (2.48)$$

where ω is a hyper-parameter determining the shape of the distribution.

2.4.4 Dueling Network

Dueling Network was designed for value based learning, this architecture separates the representation of state-value and state-dependent action advantages without supervision[6]. It consists of two streams that represent 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].

$$Q(s, a; \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a; \theta, \alpha) \quad (2.49)$$

2.4.5 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.

2.4.6 Integrated Agent:Rainbow

In the Rainbow architecture [Hes+18] 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 S_{t+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 S_{t+n} selected according to the online network as the bootstrap action $a \cdot t + n$, and evaluating such action using the target network.

2.5 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 is 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.

3

State representation learning

As discussed in 1.3, 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.

Extracting lower-dimensional information from data in order to extract meaningful information from it is an established problem in machine learning broadly, and is referred to as **representation learning** in that context. Clearly, learning to extract stateful information from image observations can be viewed as a subset of representation learning and in this context it is referred to as **state representation learning**. One important aspect of representation learning is that the representations can be abstract and only implicitly represent the data in question. Such representations can be learned in an unsupervised manner: the goal is to learn latent representations which have useful features, and as such they can be directly optimized with regards to these features and not to conform to some explicit semantic form. The purpose of this chapter is to briefly discuss representation learning in general terms, and then to investigate how it can be applied to the problem of learning state representations from images for control problems. Importantly, the problem of learning a model which can be used to achieve control through planning will not be discussed, although there are similarities between the two.

The fundamental reason why this is a promising proposition is the fact that the learning signal generated from for example image reconstruction loss is substantially stronger than the reward signal, especially in settings with sparse rewards where the reward signal is not present most of the time. This thus means that the state representation will be learned quickly compared to policy learning. Hence they can be leveraged to aid the sample-efficiency in of reinforcement learning, despite their training starting at the same time as that of the policy.

3.1 Representation learning 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. This mapping should also meaningfully generalize well on new data. The following list, introduced in [BCV13], summarizes different assumptions that can be made on the data to be represented. These priors thus translate themselves as desirable characteristics of learned representations.

1. *Smoothness*: the learned mapping f is such that $x \approx y$ generally implies $f(x) \approx f(y)$.
2. *Multiple explanatory factors*: generally, there are several different underlying factors which are the cause of the observed data. The learned representations should be able to distinguish between these different factors. In other words, these causes should ideally be disentangled features in representation space. The existence of different underlying causes is, by itself, an assumption made on the observed data.
3. *A hierarchical organizations of explanatory factors*: The learned abstractions should relate to each other in a hierarchical fashion. This concerns, for example, the assumption that different layers of convolutional neural networks should embed progressively finer features of images.
4. *Semi-supervised learning*: for inputs X and target Y to be predicted, learning $P(X)$ should help learning $P(Y|X)$ because features of X should help explain Y . This implies that unsupervised pre-training of networks should benefit supervised learning tasks because the features learned through unsupervised learning should help explain the supervised learning task.
5. *Shared factors across tasks*: moreover, the features learned on X should help in learning different supervised predictions Y' .
6. *Manifolds*: if it is assumed that the probability mass concentrates in regions with much smaller dimensionality than the data itself, then the learned representations should have smaller dimensionality to exploit this assumption.
7. *Natural clustering*: further, it is assumed that different values of categorical variables are associated with separate manifolds. This too should be evident in the learned representations.
8. *Temporal and spatial coherence*: consecutive or spatially nearby observations tend to be associated with the same value of relevant categorical concepts or result in small surface move on the surface of the manifold
9. *Sparsity*: implies either that many features are 0, or that the features are insensitive to small changes in x
10. *Simplicity of factor dependencies*: ideally factors are related to each other linearly, or otherwise in a simple fashion.

The process of extracting representations from observations, or inferring latent variables in a probabilistic view of a dataset, is often called **inference**. Models used for representation learning can be categorized as either **generative** or as **discriminative** models.

Generative models learn representations by modelling the data distribution $p(\mathbf{x})$. Such models can therefore generate realistic examples of the data they represent. They can be used for downstream tasks by evaluating the conditional distribution $p(y|\mathbf{x})$. This is done via Bayes rule.

Discriminative models instead model the conditional distribution $p(y|\mathbf{x})$ directly. Discriminative modelling consists first of inference that extracts latent variables $p(\mathbf{v}|\mathbf{x})$, which are then used to make downstream decision from those variables $p(y|\mathbf{v})$.

The benefit of discriminative models is that the expensive process of learning $p(\mathbf{x})$ is avoided. That however makes them 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.

3.1.1 Generative models

3.1.1.1 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.

3.1.1.2 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. Hence, they can be conceived them as causal models, where h activations cause the observed x , making h nonindependent. This makes recovering the posterior $p(h|x)$ intractable.

3.1.1.3 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.

3.1.2 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 \mathbf{x} , you need to define a similarity distribution which allows you to sample a positive input $\mathbf{x}^+ \sim p^+(\cdot|\mathbf{x})$ and a data distribution for a negative input $\mathbf{x}^- \sim p^-(\cdot|\mathbf{x})$, with respect to an input sample \mathbf{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} . These transformations include for example cropping, resizing, blurring, color distortion or perspective distortion and their combinations. A $(\mathbf{x}_q, \mathbf{x}_k)$ pair of query and key views is positive when these 2 views are created with different transformations on the same image, i.e. $\mathbf{x}_q = t(\mathbf{x})$ and $\mathbf{x}_k = t'(\mathbf{x})$, and is negative otherwise. A feature encoder $e(\cdot)$ then extracts feature vectors from all augmented data samples $\mathbf{v} = e(\mathbf{x})$. This is usually ResNet, in which case $\mathbf{v} \in \mathcal{R}^d$ is the output of the average pooling layer. Each \mathbf{v} is then fed into a projection head $h(\cdot)$ made up of a small multi-layer perceptron to obtain a metric embedding $\mathbf{z} = h(\mathbf{v})$, where $\mathbf{z} \in \mathcal{R}^{d'}$ with $d' < d$. All vectors are then normalized to be unit vectors. Then you take a batch of these metric embedding pairs $\{(\mathbf{z}_i, \mathbf{z}'_i)\}$, with $(\mathbf{z}_i, \mathbf{z}'_i)$ being the metric embeddings of $(\mathbf{x}_q, \mathbf{x}_k)$ 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:

$$\mathcal{L}_i = -\log \frac{\exp(\mathbf{z}_i^T \mathbf{z}'_i / \tau)}{\sum_{j=0}^K \exp(\mathbf{z}_i \cdot \mathbf{z}'_j) / \tau} \quad (3.1)$$

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

3.1.3 Common representation learning approaches

3.1.3.1 Deterministic autoencoders

Deterministic autoencoders are generative models.

$$h^{(t)} = f_{\theta}(x^{(t)}) \quad (3.2)$$

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:

$$\mathcal{J}_{AE}(\theta) = \sum_t L(x^{(t)}, g_{\theta}(f_{\theta}(x^{(t)}))) \quad (3.3)$$

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

3.1.3.2 Variational autoencoders

Variational autoencoders marry graphical models and deep learning. The generative model is a Bayesian network of form $p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$, or in the case of multiple stochastic layers, a hierarchy such as: $p(\mathbf{x}|\mathbf{z}_L)p(\mathbf{z}_L|\mathbf{z}_{L-1}) \cdots p(\mathbf{z}_1|\mathbf{z}_0)$. Similarly, the recognition model is also a conditional Bayesian network of form $p(\mathbf{z}|\mathbf{x})$ which can also be a hierarchy of stochastic layers. Inside each conditional may be a deep neural network, e.g. $\mathbf{z}|\mathbf{x} \sim f(\mathbf{x}, \epsilon)$ with f being the neural network mapping and ϵ 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:

$$p_{\theta}(\mathbf{x}_1, \dots, \mathbf{x}_M) = \prod_{j=1}^M p_{\theta}(\mathbf{x}_j | Pa(\mathbf{x}_j)) \quad (3.4)$$

where $Pa(\mathbf{x}_j)$ 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:

$$\boldsymbol{\eta} = \text{NeuralNet}(Pa(\mathbf{x})) \quad (3.5)$$

$$p_{\theta}(\mathbf{x} | Pa(\mathbf{x})) = p_{\theta}(\mathbf{x} | \boldsymbol{\eta}) \quad (3.6)$$

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

$$q_{\phi}(\mathbf{z}|\mathbf{x}) \approx p_{\theta}(\mathbf{z}|\mathbf{x}) \quad (3.7)$$

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

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = q_{\phi}(\mathbf{z}_1, \dots, \mathbf{z}_M|\mathbf{x}) = \prod_{j=1}^M q_{bm\phi}(\mathbf{z}_j | Pa(\mathbf{z}_j), \mathbf{x}) \quad (3.8)$$

This can also be a neural network. In this case, parameters ϕ include the weights and biases, ex.

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = \text{EncoderNeuralNet}_{\phi}(\mathbf{x}) \quad (3.9)$$

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma})) \quad (3.10)$$

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*.

3.1.3.3 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 denoised 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.

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 \mathbf{x} to the mean $\mu_\phi(\mathbf{x})$ and variance $\sigma_\phi(\mathbf{x})$ in the latent space. The input to D_θ is then the mean $\mu_\phi(\mathbf{x})$ augmented with Gaussian noise scaled by $\sigma_\phi(\mathbf{x})$ 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:

$$\mathcal{L}_{\text{RAE}} = \mathcal{L}_{\text{REC}} + \beta \mathcal{L}_z^{\text{RAE}} + \lambda \mathcal{L}_{\text{REG}} \quad (3.11)$$

where \mathcal{L}_{REG} represents the explicit regularizer for D_θ , and $\mathcal{L}_z^{\text{RAE}} = \frac{1}{2} \|\mathbf{z}\|_2^2$, which is equivalent to constraining the size of the learned latent space, which is needed to prevent unbounded optimization. One option for \mathcal{L}_{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 $\mathcal{L}_{\text{REG}} = \mathcal{L}_{L_2} = \|\theta\|_2^2$. Alternative options include having a gradient penalty and spectral normalization.

3.2 Representation models for control

In state representation learning 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 $\tilde{\mathbf{s}}_t \in \tilde{\mathcal{S}}$ of dimension K with characteristics similar to those of true states $\mathbf{s}_t \in \mathcal{S}$. In particular, the representation is a mapping of the history of observation to the current state: $\tilde{\mathbf{s}}_t = \phi(\mathbf{o}_{1:t})$. Actions $\mathbf{a}_{1:t}$ 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; $\hat{\mathbf{o}}_t$ denotes the reconstruction of \mathbf{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.

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 [Les+18] overview paper. In this section state representation learning for control in general is discussed. as this will allow for a broader contextualization of our own work.

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.¹ 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.

3.2.1 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 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

$$\mathbf{s}_t = \phi(\mathbf{o}_t; \theta_\phi) \quad (3.12)$$

$$\hat{\mathbf{o}}_t = \phi^{-1}(\mathbf{s}_t; \theta_{\phi^{-1}}) \quad (3.13)$$

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

¹The term autoencoder is overloaded in this case.

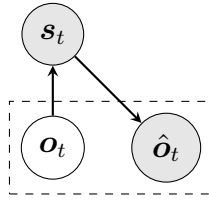


Figure 3.1: Auto-encoder: learned by reconstructing the observation (one-to-one). The observation is the input and the computed state is the vector at the auto-encoder’s bottleneck layer, i.e. is the output of the encoder part of the auto-encoder network. The loss is calculated between the true observation and the reconstructing observation (which is obtained by passing the observation through both the encoder and the decoder).

3.2.2 Forward model

The auto-encoder does not encode dynamic information. Since that information is necessary for control, usually a few consecutive 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 train a model to predict future state representations. A model can also be trained on 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

$$\hat{\mathbf{s}}_{t+1} = f(\tilde{\mathbf{s}}_t, \mathbf{a}_t; \theta_{\text{forward}}) \quad (3.14)$$

The forward model can be constrained to have linear transition between $\tilde{\mathbf{s}}_t$ and $\tilde{\mathbf{s}}_{t+1}$, 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.

3.2.3 Inverse model

The introduction of 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 the 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.

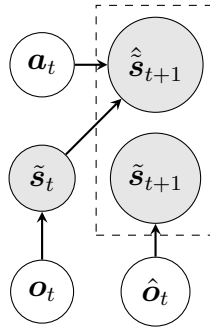


Figure 3.2: Forward model: predicting the future state from the state-action pair. The loss is computed from comparing the predicted state against the true next state (the states being the learned states). This can also be done directly by predicting the next observation and comparing against it.

Formally, we can write this as:

$$\hat{a}_t = g(\tilde{s}_t, \tilde{s}_{t+1}; \theta_{\text{inverse}}) \quad (3.15)$$

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.

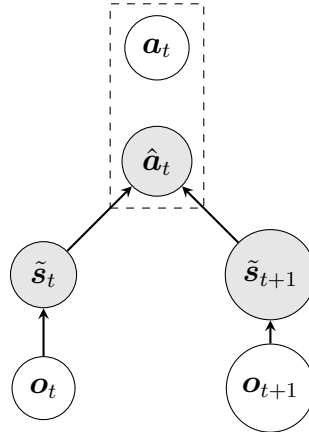


Figure 3.3: Inverse model: predicting the action between two consecutive states. The loss is computed from comparing the predicted action between two consecutive states against the true action that was taken by the agent between those two states. (the states being the learned states).

3.2.4 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 consecutive observation are drastically different. This is called the slowness principle.

3.2.5 Using hybring objectives

The approaches outlined thus far can be combined into hybrid approaches, for example [Wat+15].

3.3 Model-based reinforcement learning

Like the name suggest, in model-based reinforcement learning a “world model” is learned. While there exists a whole spectrum of methods between pure model-free and model-based ones, the key distinguishing feature of model-based methods is that the learned model is used to *plan* actions. In this case, the task of reinforcement learning in the narrow sense is to learn the values of different states. This then enables calculation of trajectories toward states with high rewards. In model-free methods, only the following action is selected at on iteration of the process because only the transition reward is learned and states these transitions lead to are unknown (not explicitly modelled).

4

Related Work

As said in the introduction, the goal of the thesis is to use state representation learning to increase the efficiency and final 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 continuous 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.

4.1 Reinforcement learning on Atari

Started with [Mni+13]. We already discussed [Hes+18]. Agent 57 [Bad+20] was the first deep RL agent that outperforms the standard human benchmark on all 57 Atari games. It was built 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 policies 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 only 100000 transitions.¹ This equates to 2.5 hours of real time.

4.2 Efforts in increasing efficiency in Atari

At the moment of writing, to the authors knowledge, the most efficient algorithm is [Ye+21] which is based on MuZero [Sch+20a] 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 3.1.3.3, 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.

4.3 State representation learning for efficient model-free learning

Auxiliary losses may be used in a myriad of different ways to help reinforcement learning. 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 reasonable to assume that this happened because a novel state has been encountered. This means that the loss

¹This 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.

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.

4.3.1 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 [LR10]. 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 be used. In [Yar+19], this was solved by employing the regularizations introduced in [Gho+19], which were already discussed in 3.1.3.3.

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.

4.3.2 Stochastic generative models

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 2.1.2, the formal setting for reinforcement learning is the Markov decision process which is stochastic. 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 [Lee+20], (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].

4.3.3 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 [Rak+21], (TODO:check these 2) is used in [Ana+19] and [Maz+20] is to use mutual information. A particularly promising avenue is to learn discriminative representation models through bootstrapping as introduced in [Gri+20]. This has been employed to learn state representations in [Sch+20b], and in [Mer+22a] 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 [Lee+20].

5

Methods

5.1 Formulating our hypotheses

Early attempts to integrate unsupervised state representation learning with reinforcement learning did not yield the expected results. One of the reasons for this was due to the fact that joint training of both algorithms did not work due to the instability caused by different training objectives. Before discussing how to solve this issue, we first need to elaborate why joint training is beneficial. In most recent work, convolutional neural networks are used to extract features from images. This includes reinforcement learning on images: convolutional layers precede linear layers. The convolutional layers, with possible addition of some of the linear layers, constitute the “feature extraction” portion of the policy networks which learn from images. In all of the related work which utilizes state representation learning and is discussed above unsupervised learning is used to train the feature extractor portion of the reinforcement learning network. Because unsupervised learning tasks are not trained to extract states in a supervised manner (because the states are inaccessible), the features they learn will not exactly correspond to true states nor will they be able to exactly learn the state transition dynamics. This of course does not mean that approximate state representation can not be sufficiently accurate however. For example, in [Sto+21] the feature extractor trained with augmented temporal contrast (a novel unsupervised learning task introduced in the work) was, on some problems, able to learn features which resulted in sample-efficiency close to training on true states.

On the other hand, by training the policy network to maximize reward, the feature extraction portion of the network will implicitly learn to extract features which are good enough to choose reward-maximizing actions. Thus, logically speaking, the feature extractor trained with just the reinforcement learning signal has to be able to extract stateful information for the images, otherwise the agent would not perform well. However, because the reward signal is sparse and contains only indirect information about state, the feature extractor, which contains most of the policy network parameters in simple problems, learns slowly.

This leads us to our first hypothesis 1. Assuming that the learned representations correspond to the images well, thus also meaning that they do not destroy stateful information, then they could be interpreted as partial extractions of states from images. Reinforcement learning feature extraction should then learn more efficiently on top of those representations due to the fact that the search space has been constricted. We illustrate this assumption in 5.1. Given the fact that neural networks

have to be overparametrized in order to learn using stochastic gradient descent and their generalization capabilities, the feature extractors for the unsupervised learning and reinforcement learning tasks can be the same network. Furthermore, owing to learning properties of stochastic gradient descent, networks need to be updated with samples taken from all available information. In other words, they “forget” information they have learned unless it continues to be provided in subsequent gradient updates. For this reason we suspect that continuing with unsupervised learning updates even after the unsupervised learning loss initially diminishes will be necessary to ensure the constriction described above. Following this results in joint training of state representation learning and reinforcement learning, which is why we hypothesize that it will yield higher sample-efficiency than other training approaches.

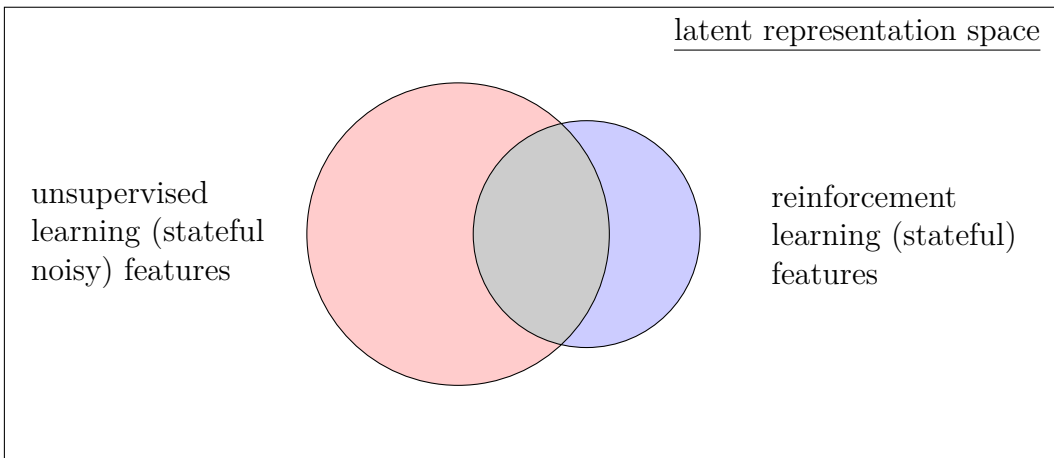


Figure 5.1: Schematic of the feature extractor neural network parameter space. Since unsupervised learning converges much faster, it constricts the search space for the features extracted through reinforcement learning. This constriction is enforced joint training of both unsupervised state representation and reinforcement learning.

Because the objectives are different and because unsupervised learning is much faster, we hypothesize that regularization is necessary 3. Additionally, we hypothesize that representations which better represent true states and state transition dynamics, i.e. the underlying Markov decision process will better condition the reinforcement learning feature extraction task 2.

5.2 Our approach

As stated previously, the goal is to learn effective state representations while training the policy. We opt for a deterministic generative model to learn state representations with, specifically a deterministic autoencoder trained with image reconstruction loss. This is done because performance of a generative model is easier to analyse than discriminative ones. The reinforcement learning algorithm shares its encoder with the encoder for state representation learning (they are the same network, with different heads attached). Our main inspiration for this foundational approach comes from [Yar+19]. However, since we elect to work in discrete action environments,

we choose [Hes+18] as the underlying reinforcement learning algorithm. Since we are not trying to achieve state-of-the-art results, we could have elected another algorithm, but this allowed us to more easily compare our results to those applicable in 4, and also to have faster training times which made experimentation easier. We introduce changes informed by 4 in order to test our hypothesis. Specific papers will be referenced when appropriate. In particular this equates to the following:

1. We test hypothesis 1 with four different training modes.
 - (a) Only the reinforcement learning algorithm is trained. This serves as the control case.
 - (b) The autoencoder is pretrained, fixed in place, and reinforcement learning algorithm is trained on top of the unsupervised learning features without the ability to update the encoder. This roughly tells us the quality of features (state representations) obtained through unsupervised learning alone.
 - (c) Same as before, but now we update the encoder with reinforcement learning loss. This serves as the control case for the following mode.
 - (d) We jointly train the encoder both with unsupervised learning and with reinforcement learning loss. The encoder continues to be updated even after the unsupervised learning loss diminishes.
2. We test hypothesis 2 with the following additional losses.
 - (a) Instead of doing unsupervised learning to reconstruct the given observations, we also pass it action and predict the following observations. Since we can not pass the actions before they have been selected by the policy network, they are passed in the decoder (the generative part of the autoencoder). This thus constitutes forward prediction in pixel space.
 - (b) Finally, we add inverse dynamics loss on top of the reconstruction, forward predictive and reinforcement learning loss. As shown in [Pat+17], this should encourage the state representation to focus the parts of the observation related to the agent and thus further condition state representations. As argued in [Rak+21], this loss alone can not be represent the MDP so it is not considered in isolation.
3. We test hypothesis 3 with the following regularization techniques.
 - (a) Following the analysis carrier out in [Yar+19], we begin by using regularization techniques introduced in [Gho+19]. Since we found that joint training of state representations and reinforcement learning does not work without at least L2 latent space loss, we use it all joint training tests.
 - (b) Informed by regularization effects of denoising autoencoders, and by great success achieved with the random shift augmentation in [KYF20; Yar+21] and other recent work, we too use random shift augmentation on observations in unsupervised learning updates. This serves as our test.

5.3 Environment and Preprocessing

We perform a comprehensive evaluation of our proposed method on the Arcade Learning Environment [Bel+13], 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 down sampled to a 84 x 84 picture and transforming their RGB representation to gray-scale. This is standard practice in the field and it makes the training faster without fundamentally simplifying the problem.

We constrict ourselves to games in which there is no partial observability and where the reward is relatively dense. This is because we are not focusing on exploration which is necessary to solve some of the games. Methods known to use which formulate intrinsic rewards are compatible with our approach and can be in fact integrated with it like in [Mer+22b].

5.4 Module implementation

5.4.1 Tianshou

We implement our approach as a modular extension in [Wen+21]. The purpose of this choice is to make our work easily accessible and extendable. Tianshou is an actively updated reinforcement learning library and offers the highest number reinforcement learning algorithms on the market. It is based on Python and PyTorch. It is well documented, provides tests and type hints. It offers gym wrappers for the most popular reinforcement learning benchmarks, including Atari. It supports vectorization for all environments.

The guiding principle of Tianshou is to abstract the reinforcement learning problem as much as possible because good abstractions are the basis of modularity. This is done by splitting reinforcement learning algorithms into the following modules:

- `Batch`
- `ReplayBuffer`
- `Policy`
- `Collector`
- `Trainer`

These modules interact in the way depicted in ??.

`Batch` is designed to store and manipulate “hierarchical named tensors”. Hierarchical named tensors are a set of tensors whose name forms a hierarchy. In essence, they are nested dictionaries.

There are 7 reserved keys in `Batch`:

1. `obs` — observation at step t
2. `act` — action at step t
3. `rew` — reward at step t
4. `done` — done flag at step t
5. `obs_next` — observation at step $t + 1$

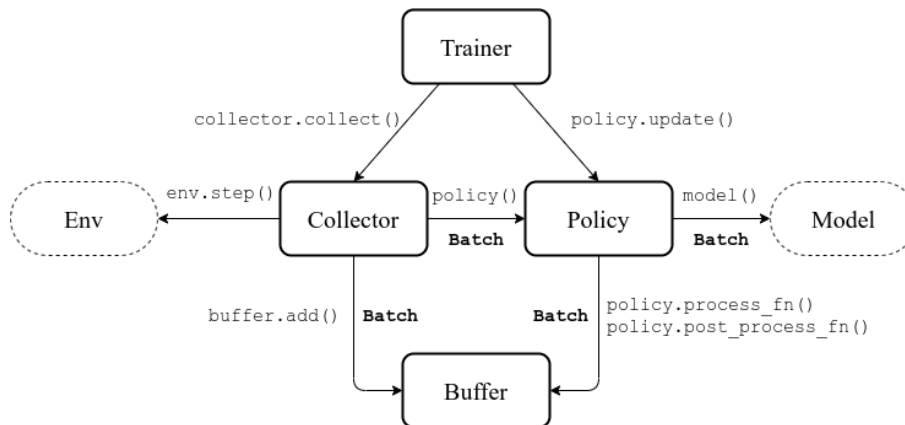


Figure 5.2: Tianshou concepts

6. `info` — info at step t
7. `policy` — data computed by policy at step t

`ReplayBuffer` stores experience data. It's purpose is to manage `Batch`. All data is stored in a circular queue.

There are different classes of policies, but all policies must inherit from `BasePolicy`. Typical function are the following ones:

1. `__init__()`
2. `forward()` — compute action with given observation
3. `process_fn()` — pre-process data from the replay buffer
4. `learn()` — update policy with a given batch of data
5. `post_process_fn()` — pre-process data from the replay buffer
6. `update()` — this one does it all: samples from buffer, pre-processes data (ex. computing the n-step return), learn from data and post-processes the data (ex. update the prioritized replay buffer)

`Collector`'s task is to interact with the environment and store the observed transitions in `ReplayBuffer`.

5.4.2 Trainer

's task is to balance environment interactions by calling on `Collector` and updating the agent by calling on `Policy`.

5.4.3 Implementing state representation learning in Tianshou

Surprisingly, Tianshou lacks state representation learning algorithms which is why believe it could benefit from our work. We implemented state representation learning as a `Policy` which takes a reinforcement learning `Policy` as an argument. This makes our implementation truly modular and we had to change less than 5 lines DQN implementation for it to work. We believe that this was due to a minor design

error in n-step return implementation, rather than our lack of foresight. Apart from this we used Tianshou’s design to our advantage. By having the state representation learning wrapped around reinforcement learning, we manipulate the batches sent to the reinforcement learning algorithm and thus keep it fully encapsulated. Data augmentation is implemented passing appropriate preprocessing functions to the `Trainer`. The user only needs to ensure matching network dimensions to utilize state representation learning algorithms and pass the appropriate flag to use the one we implemented.

5.5 Network architectures

As will be further discussed in 6, an encoder just large enough to solve the reinforcement learning problem by itself does not have the capacity to (also) solve the unsupervised pixel reconstruction problem. The forward prediction problem is more difficult still. In our experiments we used 3 encoders in total, while keeping the fully connected perceptron for reinforcement learning the same. All decoders are simply mirror images of the corresponding encoders, where 2-dimensional convolutional layers are replaced with 2-dimensional deconvolutional layers, and the order of the layers is flipped.

The smallest encoder is the one used in [Mni+15]. It has only convolutional layers. We will denote a layer’s configuration as the following tuple: (number of input channels, number of output channels, kernel size, stride, padding). Our smallest network is thus of the architecture: (number of stacked frames, 32, 8, 4, 0), (32, 64, 4, 2, 0), (64, 64, 3, 1, 0). The output of the encoder is flattened and passed to the fully connected reinforcement learning layers. We use rectified linear units as activation functions after each layer.

In general, and for our purposes in particular, the smaller the bottleneck layer of an autoencoder, the better it performs its job of dimensionality reduction. For some games we were able to obtain the bottleneck of length 50, while for more visually complex games we used a bottleneck of length 256. We are sure that the bottleneck layer could be made smaller with more advanced training techniques, but this would further complicate our method and most likely increase the number of parameters which would negatively impact the performance of reinforcement learning. Furthermore, since our goal is state representation learning and not perfect reconstruction, we believe that it is better to opt for a smaller network which compresses well enough. Having that said, we would ideally also test the network architecture and training procedure proposed in [Oh+15]. The architecture of the two bigger encoders differ only in the size of the bottleneck layer. In our experiments we refer to the length of the bottleneck layer as **features dimension**. Their convolutional layers are the following ones. (number of stacked frames, 32, 3, 2, 0), (32, 32, 3, 2, 0), (32, 32, 3, 2, 0), (32, 32, 3, 2, 0). These convolutional layers are now followed by a linear layer of the size ($32 \times 35 \times 35$, features dimension). We use rectified linear units as activation functions after each layer but the last which is a hyperbolic tangent function.

The both Q and V networks consist of 2 fully connected linear layers with sizes of (features dimension, 512) and (512, number of actions * number of atoms).

5.6 Hyperparameters

As our goal was to investigate the effectiveness of leveraging unsupervised learning for reinforcement learning, we decided to change the underlying reinforcement learning algorithm as little as possible. This way we also test the effectiveness of our method as a module which is meant to be used as an extension to the underlying algorithm. Of course, ideally we would investigate the interplay between the two losses to see whether particular configurations work better than others, but we had to focus on testing our hypothesis, albeit in a somewhat limited way. The reinforcement learning related hyperparameters can be found in 5.1.

Table 5.1: Table of reinforcement learning related hyperparameters.

parameter	value
epsilon test	0.005
epsilon train	1.0
epsilon train final	0.05
buffer size	100000
learning rate	0.0001
gamma	0.99
number of atoms	51
v_{min}	-10.0
v_{max}	10.0
noisy std	0.1
alpha	0.5
beta	0.4
beta final	1.0
beta anneal steps	5000000
n-step	3
target update frequency	500
epoch	50
steps per epoch	100000
steps per collect	10
updates per step	0.1
batch size	32
number of workers for training	10
number of workers for testing	10
number of stacked frames	4

Alternatively, we could have opted for hyperparameters suggested by [VHA19] which are tailored for sample-efficiency. We chose not to due to the fact that, while they are more sample-efficient, result in longer training times. Since we are not interested in achieving state of the art results nor are we thoroughly experiment with reinforcement learning related hyperparameters, we chose to perform more tests by selecting a less computationally taxing set of hyperparameters.

6

Results

Due to computational constraints, we constrain ourselves to 7 Atari games. Both DQN and Rainbow perform well by themselves on the chosen games. However, the games vary in visual complexity and control problem difficulty. In particular, the selected games are:

1. Breakout
2. Enduro
3. Ms Pacman
4. Pong
5. Qbert
6. Seaquest
7. Space Invaders

Because we can only indirectly gauge the effect of various metrics through final algorithm performance, the primary metric of interest is obtained return in relation to the number of training iterations. Due to different reward scaling, we keep results on different games in separate graphs. We further divide the results into those pertaining to different hypotheses in order to avoid line clutter. As a final note, the results for each particular setting are **single-runs**. Generally speaking, this is not adequate due to high pseudorandom number generator seed dependence and noisiness of reinforcement learning in general. Depending on the problem, 3-10 runs are averaged, or the best one is selected, in order to tackle this problem.

6.1 Effectiveness of pretrained encoders

To begin, we first need to estimate the possible sample-efficiency gains. This is done by comparing returns per training step between only reinforcement learning and only reinforcement learning where training starts with a encoder trained from the first reinforcement learning-only run. We will use runs with reinforcement learning only as the baseline for all cases. Because we chose games which can be essentially solved with only reinforcement learning, the encoder of a network trained with reinforcement learning should serve as an ideal pretrained encoder.

As can be observed in 6.1, using a pretrained encoder can be both beneficial or detrimental, but it does not seem particularly important, especially long-term. We used the smaller encoder for runs consisting only of Rainbow and we used the same architecture for the run with an encoder pretrained with Rainbow. Notably, some games systematically suffer when reconstruction loss is applied in any way, in particular Breakout . The reason is that MSE loss is not well suited for small details,

be they critical or not. In Breakout the ball is not well represented, as can be seen in reconstructions.¹ This and other problems with the method will be further discussed in 7.

6.2 Effect of varying network sizes on Rainbow

As mentioned in 5.5, we use a larger encoder when using reconstruction loss. Thus we need to observe the effect of using different encoder sizes reinforcement learning alone. We present this separately in 6.2 to avoid clutter.

As can be seen in 6.2, some games benefit greatly from having a large encoder (ex. Seaquest), while learning completely fails on others (ex. Breakout). Despite rising the variance of the final result, the effect seems to be neutral overall.

6.3 Effectiveness of continuous updates

Having seen the effect of having pretrained encoders, we turn the attention to the effect of continuous updating of the encoder with reconstruction loss. The results are shown in 6.3.

6.4 Effectiveness of regularization

Before continuing the discussion, the effectiveness of regularization needs to be discussed. We present runs with both data augmentation and L2 regularization, runs with either one or the other and runs without neither in 6.4.

As can be seen in the figure 6.4, regularization can both benefit and hurt learning. L2 regularization forces the latent space to be distributed more closely to the origin. While somewhat arbitrary, we checked that it substantially helps in maintaining the same latent vectors for the same inputs across updates. This was measured by observing the reconstruction quality over encoders and decoders saved at different epochs.

Data augmentation seems to only add noise and therefore hurt learning by destabilizing it. This might be due to our particular implementation, but it is more likely due to the fact that the encoder produces different outputs for the same, but differently augmented input.

6.5 Effectiveness of forward prediction

Finally, we compare the effectiveness of forward prediction, or the importance of dynamics for state representations. The results can be observed in 6.5.

Despite fairly poor reconstruction capabilities and an error larger by an order of magnitude, state representation trained with forward prediction are overall more

¹Interestingly, the same is not the case in Pong because there most of the background is black, thereby making the ball a relatively big source of reconstruction error in comparison.

effective than simple compression. As with all results, the effects vary from game to game.

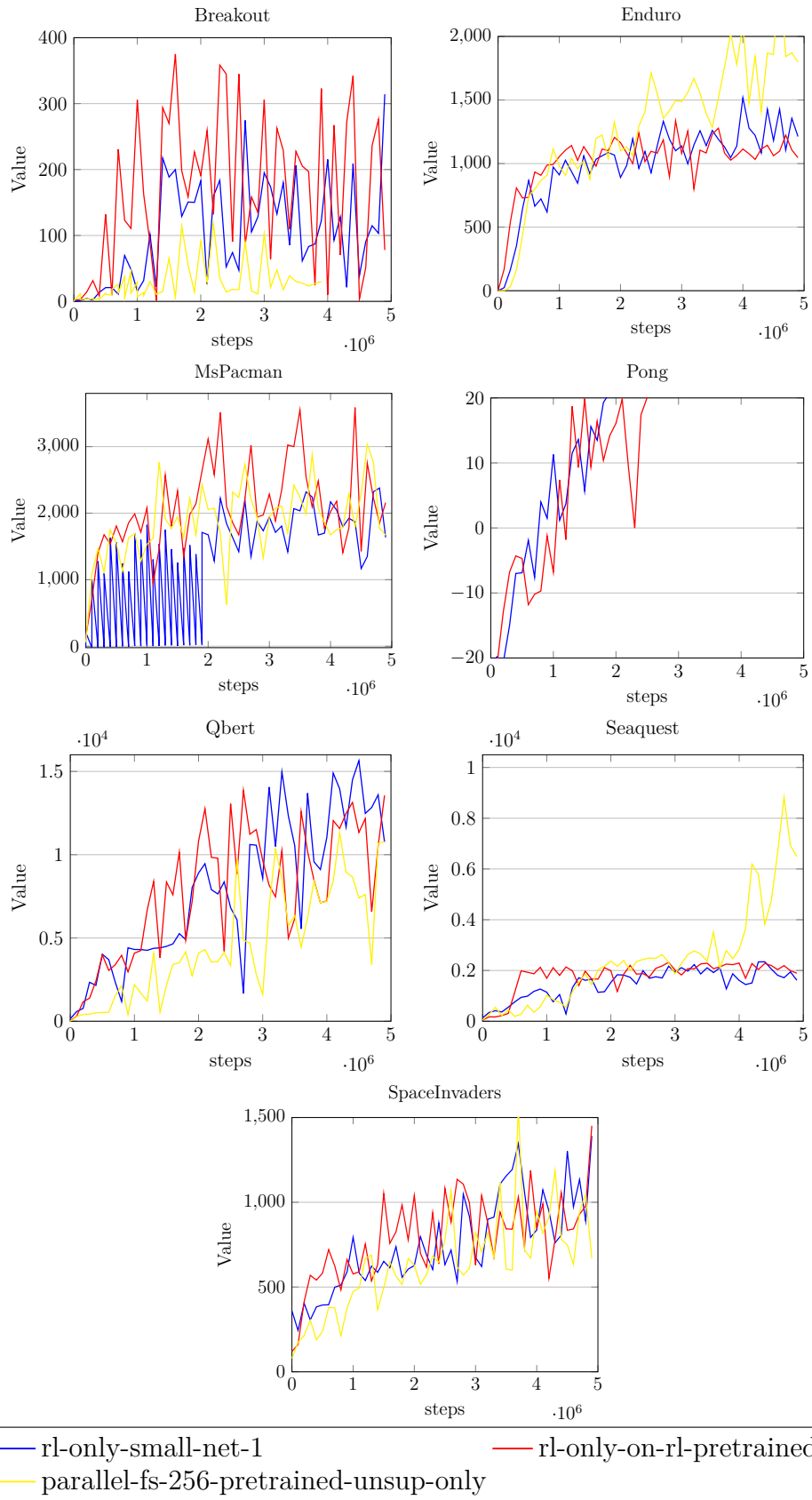


Figure 6.1: The graphs of potential efficiency gains. As can be observed from the graphs, if training is started using an encoder which was already trained using only reinforcement learning better results are achieved more quickly. Of course, this does not encompass all potential benefits — unsupervised learning could make the problem easier overall and thereby allow for both even faster learning and higher final scores.

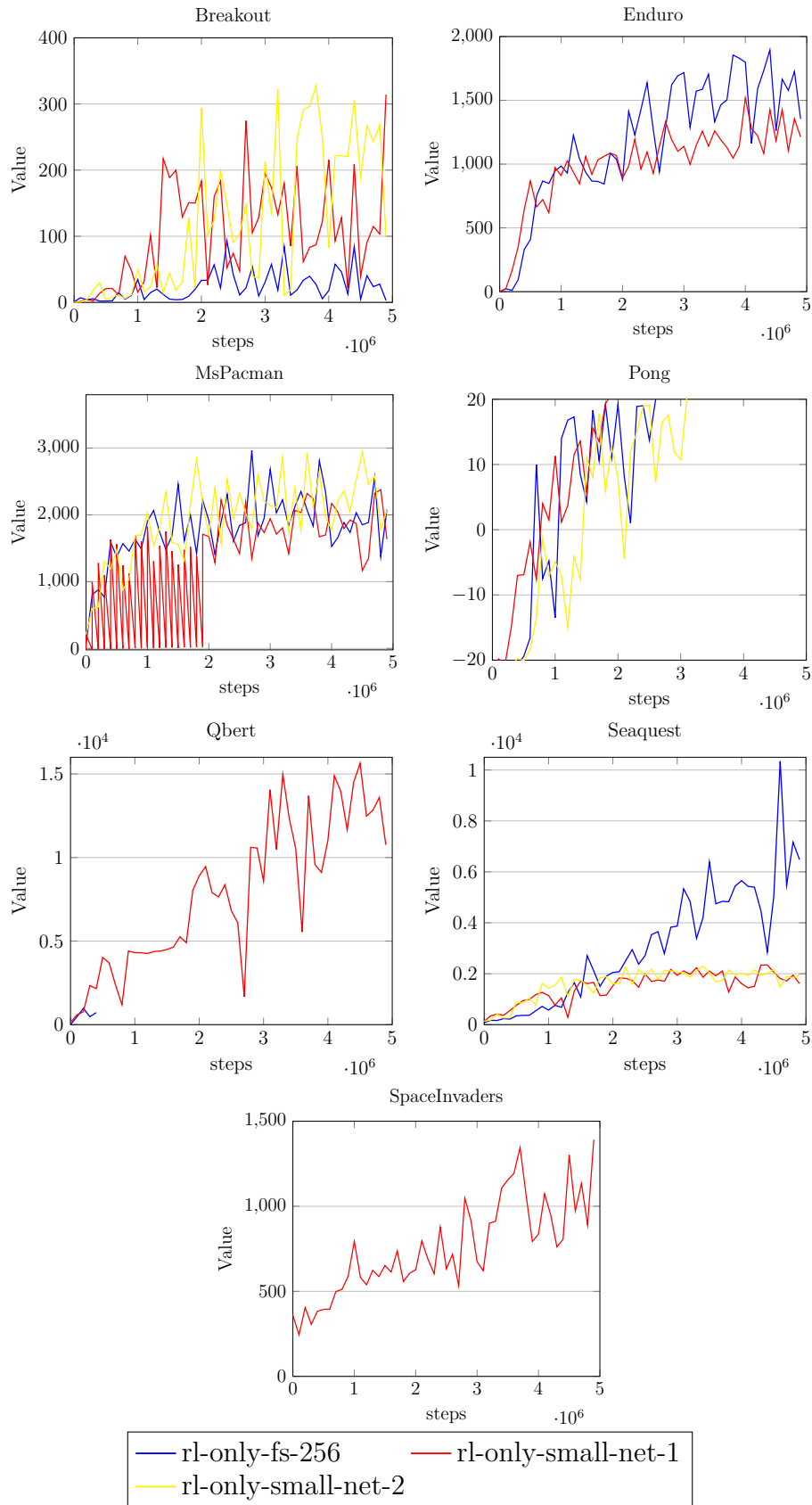


Figure 6.2: Runs with Rainbow only, but with different encoder sizes. As can be seen, some games strongly benefit from having a larger encoder, while learning fails on others.

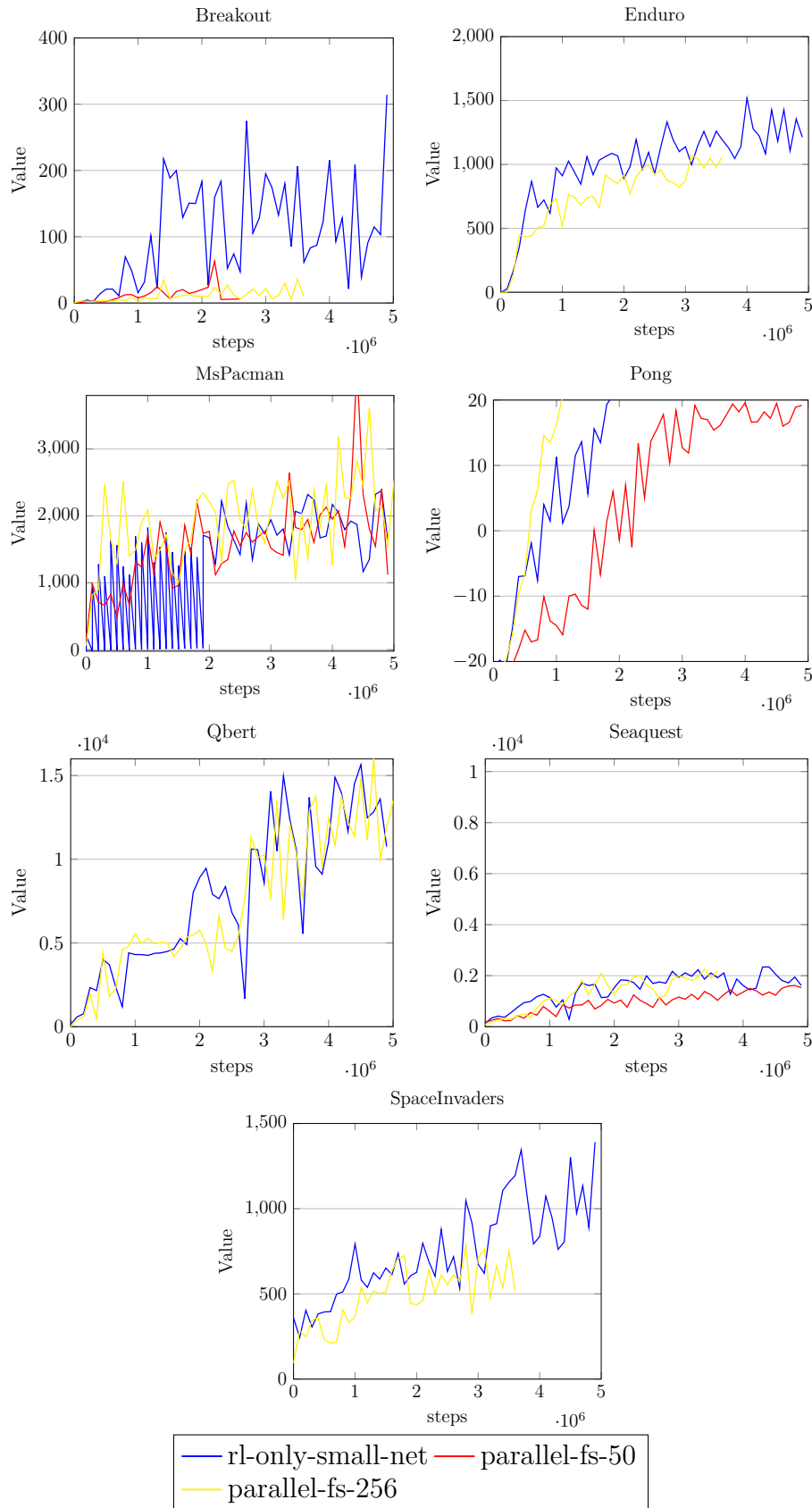


Figure 6.3: Effectiveness of parallel training. Overall the results are negative, some slightly and others strongly. Interestingly, using a pretrained encoder has a negative effect in this case. We suspect that this is due to the fixation on a particular local minimum caused by reconstruction loss. Importantly, the negative effect is present despite regularization maintaining a fixed latent space over a relatively large number of epochs (5-10). Results tend to be worse in games where MSE loss is less suitable

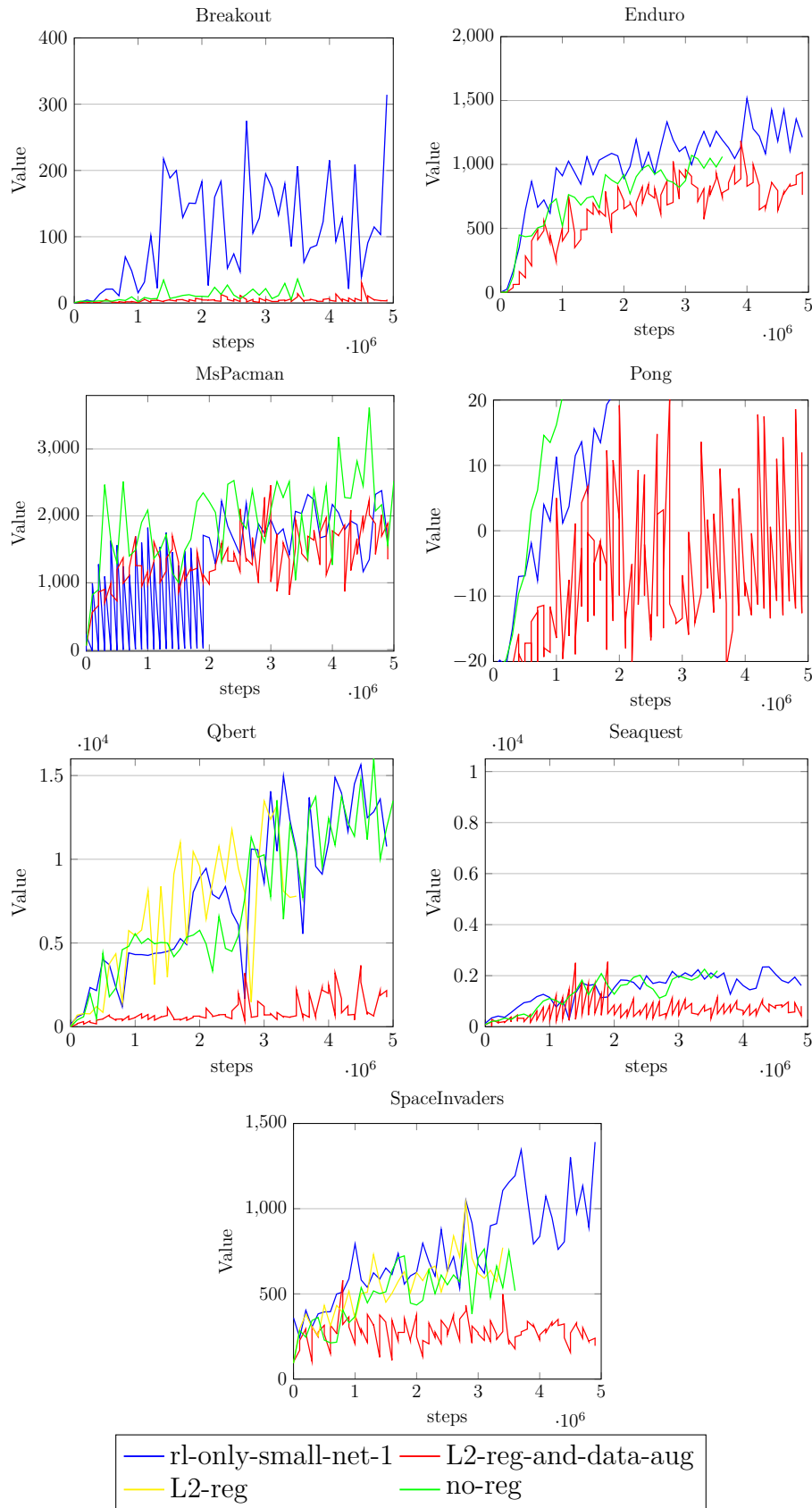


Figure 6.4: Comparison of using and L2 latent space regularization and data augmentation in parallel training. L2 regularization helps overall because it helps fix the latent representations in place, thereby stabilizing the overall training process. Data augmentation on the other hand clearly hurt across the board.

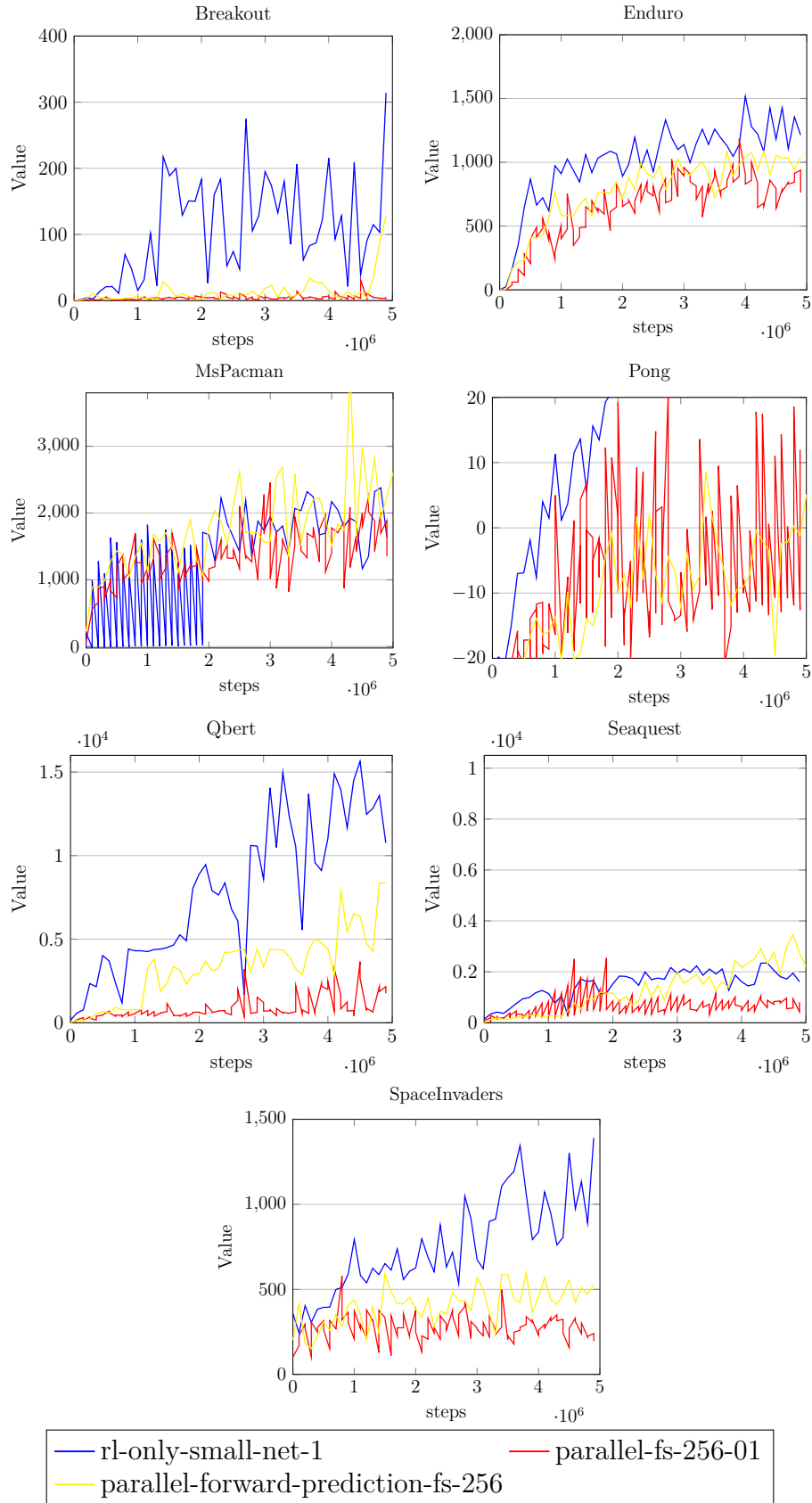


Figure 6.5: Comparison between latent representation trained to compress the current frames and those trained to perform one-step forward predictions. Despite having a significantly higher reconstruction error, encoder trained to perform forward prediction perform better.

7

Discussion

Overall, the results are both disappointing and inconclusive. Both intuitively and by being informed with the relevant literature, we expected stronger results. In this section we explore possible explanations for the results and use them to suggest promising avenues for further research. We break our discussion of the results into answering the following questions:

1. What are the differences between features and states and how important are they to the final performance?
2. Why is reconstruction loss a particularly bad at representing stateful information?
3. What could be the characteristics of more successful approaches to using unsupervised learning for state representation learning?
4. What role does regularization play in reinforcement learning, unsupervised learning and their combination?

Given the generality of our hypothesis, we of course can not answer these questions fully, but we can give hints and suggestions.

7.1 Differences between features and states

We believe that the central idea behind using unsupervised representation learning as state representation learning is the one illustrated in 5.1. If the hypothesis behind it is correct, then learning should be faster. This is certainly the case in numerous works which use shared layers for actors and critics in actor-critic methods. Of course, this only works on the condition that the inherent instabilities in training networks with different losses have been taken care of in one way or another. The specific differences between our features and states will be further discussed in the following subsections, while a more general discussion will be held here.

When looking at simple games such as Pong or Seaquest, one can clearly see what constitutes a state. The dynamic information primarily consists of the positions and velocities of objects in the image. More static information would be the types of objects (the player, projectiles, obstacles etc) and so on. For such simple problems, a resourceful engineering could easily craft representations consisting of at most a few dozen parameters. From our experiments with different autoencoder architectures and sets of hyperparameters, we could not obtain such low-dimensional representations through unsupervised learning. While deep learning experts could certainly tailor the various parameters and the training process for each specific game and thereby obtain lower dimensionality, the point of finding general purpose

reinforcement learning solutions would be lost. After all, one could (relatively) easily hand-craft near-perfect solutions for every Atari57 game. Thus we are left with the conclusion that even if unsupervised learning methods can help reinforcement learning, they by themselves are not the key to getting (super) human speed in acquiring knowledge in control problems.

We therefore propose two avenues toward highly sample-efficient reinforcement learning. One is to formulate model or state representation learning which is geared specifically toward learning dynamics and to make it more restrictive. In other words to find more inspiration in works such as [pilco]. This way single problems can be quickly learned from scratch. Another is quite the opposite and more inspired by humans. Humans generalize across all of their experiences, which is what enables them to quickly make sense of novel situations. Similarly, very large neural network models could also leverage such broad representations to learn quickly. An interesting start would be to see whether a single network could be trained to play most Atari games, and other games as well and learn new games quicker than starting from scratch by leveraging its implicit representations.

There is another important broad conclusion to be drawn from the gap between our expectations and our results. We expected the results to be better because our vague sense of what is a feature learned through either unsupervised or reinforcement learning led us to believe that they are related. This would probably not have happened if we had a better understanding of what these features really are. Hence we believe that having tools to investigate the nature of features obtained through deep learning would help tremendously in designing unsupervised learning techniques for state representation learning. Thus better understanding and explainability of neural networks would help not only in safety of systems relying on it, but also in algorithm design — just looking at the final result and other indirect metrics is not good enough.

7.2 Reconstruction loss

Using pixel-reconstruction loss in general, and MSE loss in particular is a very poor choice for state representation learning. A good reconstruction contains plenty of non-stateful information, while a poor one loses stateful information while still keeping a lot of non-stateful information. The reason for this is that lowering pixel-reconstruction loss does not directly incentivize learning about states. We believed that simply lowering dimensionality would make the reinforcement learning problem easier, but this is clearly not the case unless non-stateful information is removed. Loosely speaking, with encoders trained via pixel-reconstruction the entropy remains roughly the same when seen through the lens of reinforcement learning. As already mentioned, parallel training objectives make the training process more unstable which hinders progress. Hence the representation being learned needs to be good enough to overcome this negative effect and better still in order to obtain tangible benefits.

Furthermore, having a generative model significantly increases the wall-clock training time (200-400%) due to the networks needing more parameters and due to the existence of the decoder. For this reason discriminative models should be preferred

for state representation learning and as seen in 4 this is a clear trend in recent years. Generative models still have their place in model-based approaches of course.

7.3 Importance of representing dynamics

Forward prediction in pixel-space showed an edge over just compression. While theoretically sound, this is somewhat surprising given the fact that the reconstruction error was over a magnitude larger than in the case of just compression. This clearly shows the importance of directly incentivizing learning stateful information. There are numerous improvements which could be made to improve the quality of predictions:

- learning to predict further into the future with the help of curriculum learning should increase the accuracy on shorter time scale
- more specialized architectures like using temporal-convolutions, having separate parameters for each action in the decoder,...

We refrain from suggesting further improvements because as discussed in 7.2, we believe that reconstruction loss is a bad choice for state representation learning and that an all together different approach should be taken, ex. self-predictive bootstrapped latent representations.

7.4 Not all regularization is the same

As shown in 6.4, improper regularization hurts learning, while meaningful regularization helps. In our context regularization serves to stabilize the learning process. This mainly refers to preventing the latent representations to shift over time, i.e. to produce the same latent representations when given the same frames. Since L2 regularization constricts the manifold of latent representations, it serves this purpose well.

In our implementation, data augmentation seems to only introduce noise because it makes the encoder produce different latent vectors when given the same frame. Interestingly, it had a positive effect on learning in runs with 10 times more frequent network updates (compared to environment steps), but as we performed these runs on a limited number of games this claim can not be further corroborated with substantial evidence.

8

Conclusion

The idea behind this thesis is simple and straightforward: using autoencoders trained with pixel-reconstruction loss to lower the dimensionality of images should make the reinforcement learning problem on images easier. As it turns out, this is not necessarily the case. It, although unremarkably, works on visually simple games like Pong where all important details can be captured by MSE loss, i.e. by formulating pixel reconstruction loss on a pixel-by-pixel basis. Simultaneously, the same method completely stunts learning on games such as Breakout where the ball, a crucial element of the state, is captured poorly.

We conclude that pixel reconstruction loss should be avoided for the purposes of state representation learning as it does not directly incentivize learning stateful information, but only serve to lower dimensionality, sometimes even at the expense of stateful information. This is most exemplified by the fact that forward prediction in pixel space supports learning better, despite the fact that it maintains an order of magnitude bigger error, just because it more strongly encourages embedding of stateful information (velocities in particular). Given the listed reasons, and the gap between our expectations and the obtained results, we recommend more research into the nature of features obtained through both unsupervised and reinforcement learning so that beneficial combinations can be deduced, rather than guessed. On a shorter time scale, we recommend discriminative models as both seem more promising and are faster in terms of wall-clock time.

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.
- [Bel+13] Marc G Bellemare et al. “The arcade learning environment: An evaluation platform for general agents”. In: *Journal of Artificial Intelligence Research* 47 (2013), pp. 253–279.
- [BCV13] Yoshua Bengio, Aaron Courville, and Pascal Vincent. “Representation learning: A review and new perspectives”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.
- [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>.
- [Oh+15] Junhyuk Oh et al. “Action-conditional video prediction using deep networks in atari games”. In: *Advances in neural information processing systems* 28 (2015).
- [Wat+15] Manuel Watter et al. “Embed to control: A locally linear latent dynamics model for control from raw images”. In: *Advances in neural information processing systems* 28 (2015).
- [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.
- [Hes+18] Matteo Hessel et al. “Rainbow: Combining improvements in deep reinforcement learning”. In: *Thirty-second AAAI conference on artificial intelligence*. 2018.
- [Les+18] Timothée Lesort et al. “State representation learning for control: An overview”. In: *Neural Networks* 108 (2018), pp. 379–392.
- [SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [Ana+19] Ankesh Anand et al. “Unsupervised state representation learning in atari”. In: *Advances in Neural Information Processing Systems* 32 (2019).

- [Gho+19] Partha Ghosh et al. “From variational to deterministic autoencoders”. In: *arXiv preprint arXiv:1903.12436* (2019).
- [VHA19] Hado P Van Hasselt, Matteo Hessel, and John Aslanides. “When to use parametric models in reinforcement learning?” In: *Advances in Neural Information Processing Systems* 32 (2019).
- [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.
- [Gri+20] Jean-Bastien Grill et al. “Bootstrap your own latent—a new approach to self-supervised learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21271–21284.
- [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.
- [Lee+20] Alex X Lee et al. “Stochastic latent actor-critic: Deep reinforcement learning with a latent variable model”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 741–752.
- [Maz+20] Bogdan Mazoure et al. “Deep reinforcement and infomax learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 3686–3698.
- [Sch+20a] Julian Schrittwieser et al. “Mastering atari, go, chess and shogi by planning with a learned model”. In: *Nature* 588.7839 (2020), pp. 604–609.
- [Sch+20b] Max Schwarzer et al. “Data-efficient reinforcement learning with self-predictive representations”. In: *arXiv preprint arXiv:2007.05929* (2020).
- [Lev21] Sergey Levine. *Deep Reinforcement Learning course (CS 285) lectures*. Fall 2021. URL: https://www.youtube.com/playlist?list=PL_iWQ0sE6TfXxKgI1GgyV1B_Xa0DxE5eH.
- [Rak+21] Kate Rakelly et al. “Which Mutual-Information Representation Learning Objectives are Sufficient for Control?” In: *Advances in Neural Information Processing Systems* 34 (2021).
- [Sto+21] Adam Stooke et al. “Decoupling representation learning from reinforcement learning”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 9870–9879.
- [Wen+21] Jiayi Weng et al. “Tianshou: A Highly Modularized Deep Reinforcement Learning Library”. In: *arXiv preprint arXiv:2107.14171* (2021).

- [Yar+21] Denis Yarats et al. “Mastering visual continuous control: Improved data-augmented reinforcement learning”. In: *arXiv preprint arXiv:2107.09645* (2021).
- [Ye+21] Weirui Ye et al. “Mastering atari games with limited data”. In: *Advances in Neural Information Processing Systems* 34 (2021).
- [Mer+22a] Astrid Merckling et al. “Exploratory State Representation Learning”. In: *Frontiers in Robotics and AI* 9 (2022).
- [Mer+22b] Astrid Merckling et al. “Exploratory State Representation Learning”. In: *Frontiers in Robotics and AI* 9 (2022). ISSN: 2296-9144. DOI: 10.3389/frobt.2022.762051. URL: <https://www.frontiersin.org/article/10.3389/frobt.2022.762051>.

A

Appendix 1

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.