CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

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

a

Master's thesis in Computer science and engineering

NAME FAMILYNAME

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

Master's thesis 2022

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

a

NAME FAMILYNAME

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

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

Supervisor: Name, Department
Advisor: Name, Company or Institute (if applicable)
Examiner: Name, Department

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

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2022

Improving sample-efficiency of model-free reinforcement learning algorithms by learning latent space representations
a
NAME FAMILYNAME
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Abstract text about your project in Computer Science and Engineering.

Keywords: Computer, science, computer science, engineering, project, thesis.

Acknowledgements

Here, you can say thank you to your supervisor(s), company advisors and other people that supported you during your project.

Name Familyname, Gothenburg, May 2022

Contents

List of Figures

List of Tables

1
Introduction

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 existance of an agent which observes its environment and interacts with it though its own actions. "Error" implies that the agent has a goal it tries to achieve and that it does not know how to achieve it (in the most effective manner). What it can do is take different actions and appraise them according to how closely they lead the agent toward its goal, thereby observing the quality of those actions. By repeatedly exploring the effects of various sequences of actions, the agent can find, i.e. learn, the sequence of actions which lead to its goal. Here it is important to discuss what a goal is. To formalize the process outlined above, one needs to describe it in purely mathematical terms. Thus, among other things, the goal needs to be described numerically. To achieve that, the notion of a reward function is used: it maps every state of the environment to a number which denotes its value called the reward. The state of the environment to which the highest reward is ascribed is then the goal. A more general description of the goal of reinforcement learning is to maximize the sum of rewards over time. Formalization of the entire process will be carried out later in the text, while here only the most important concepts will be outlined. One of these is the trade-off between "exploration" and "exploitation." To learn just by trial and error implies learning from experience. This means that the agent can not know how a certain strategy fares unless it collects experiences which come by following said strategy. Thus in order to find a good strategy, usually refered to as a "policy", the agent needs to produce various different strategies and observe their results until it find a promising one. The process of finding different strategies and experimenting with new random behavior is called exploration. Likewise, the process of repeating a good strategy is called exploitation. Due to the curse of dimensionality, [1] in complex multi-dimensional domains it is impossible to test but a miniscule proportion of all possible strategies. Because of this, the problem of effective exploration and the trade-off between it and exploitation is a fundamental problem in reinforcement learning.

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

[1] The curse of dimensionality refers to the exponential rise of possible configurations of the problem with the number of dimensions.

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

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 perfomance 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. [2] These recent success were kick-started by Deep Q-Networks (DQN) algorithm which, by utilizing convolutional neural network, crucially 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 usually impossible to get full access to the state of the environment. The state then needs to be inferred from obsevations such as those from cameras. [3] Due to incredible results achieved in simulated environments, reinforcement learning holds the promise of solving many incredibly important engineering problems, for example robotic manipulation and grasping. Having that said, there exists a large gap between simple simulated environments and the real world, and many improvements to the current state-of-the-art algorithms are required to bridge that gap. To explain the approach investigated in this thesis, a bit more context is needed.

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

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

[3] 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 implicitely hold enough information about the true state of the world for successful goal completetion.

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 implies a harder learning problem: they need to implicitely learn those dynamics while only being provided the reward signal. This makes them much less sample-efficient. By contrast, model-based algorithms are of course more sample-efficient. Furthermore, the plan generated from the learned model can be utilized to interpret the agent's actions which in turn leads to many further benefits such as the ability to guarantee outcomes in safety-critical operations. Unfortunatelly, the twin learning objective of learning the best action-choosing policy to maximize the reward over time, and the learning of the model results in fundamental training instabilities which usually results in worse final performance. In simple terms, the reason behind this is the following one: in the beggining 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 inacessible 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.

Given the previous discussion, the goal of the thesis may be presented: the idea is to combine the sample-efficiency of model-based approaches with the flexibility of model-free methods. Another way to describe the same is to say that we want to utilize learning signals other than the reward signal to make the model-free learning more sample-efficient. In particular, we want to learn a latent representation of the environment, i.e. to find lower-dimensional embeddings of the environment, and learn a policy in this space. To make this a concrete and managable goal, we constrain ourselves to the problem of learning from images in particular. To be able to compare our results to those of other researchers, we will test our algorithms on the standard benchmark tasks in the field, namely Atari57 games. The potential benefits of the proposed approach are two-fold: firstly, we know that in general lower-dimensional optimization problems are easier to solve than higher-dimensional ones. Secondly, it is known that when algorithms learn with direct state access, they learn much faster and often achieve better final results. The main reason behind this is that images are much higher-dimensional than underlying states, and this is self-evident in the case of Atari games. Since inferring states from observations is not directly related to the reward, we expect that using unsupervised learning techniques will aid in feature extraction and thus make learning more sample-efficient. Furthermore, since the goal is not to learn the dynamics of the environment, but simply to find an equivalent, but lower-dimensional representation of it, we expect that this approach won't suffer from the problems faced by model-based approaches.

Of course, we are not the first to suggest such an approach, but we haven't found a systematic analysis of it accross an array of model-free algorithms. Related work will be discussed in its own chapter.
In total, our contributions will be the following:

1. A systematic analysis of utilizing autoencoders for learning lower-dimensional representations for greater sample-efficiency in combination with different reinforcement learning algorithms.
2. Ideally, finding algorithms which achieve the same or better performance with the autoencoder as without it, but doing so more efficiently.
3. Defining a clear direction for further research based on the previous two contributions.

2

Background

2.1 Introduction to reinforcement learning

2.1.1 Problem setting

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

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

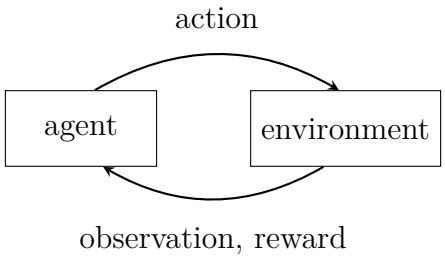


Figure 2.1: Conceptual schematic of reinforcement learning.

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

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

2.1.2 Bandit problems

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

Table 2.1: Li Zhou ,relationship among bandit, MDP, RL, and decision theory

$Learnmodeloutcome$	Mutlti armed bandits	RL
$Givenmodelofstocasticoutcomes$	Decision Theory	MDP
	Actions dont change state of the world	Actions change state

2.1.3 Markov Decision Processes

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

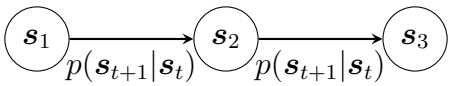


Figure 2.2: Schematic of a Markov chain.

Formally, a Markov chain $\mathcal{M}$is the defined by its state space $\mathcal{S}$ with discrete or continous state $\boldsymbol{s} \in \mathcal{S}$ and the transition operator $\mathcal{T}$. The notation $\boldsymbol{s}_t$ denotes the state at time t and it is a vector of real numbers. The transition operator allow for a succinct description of environment dynamics. For a transition probability $p(s_{t+1}|s_t)$, let $\mu_{t,i} = p(\boldsymbol{s}_t = i)$ and $\mathcal{T}_{i,j} = p(\boldsymbol{s}_{t+1} = i|\boldsymbol{s}_t = j)$. Then $\overrightarrow{\mu}_t$ is a vector of probabilities and $\overrightarrow{\mu}_{t+1} = \mathcal{T}\overrightarrow{\mu}_t$. Importantly, $\mathcal{T}$ is linear.

To model the agent's actions, we simply augment the Markov chain by adding actions as priors to state transition probabilities and defining the reward function, thereby constructing a Markov decision process. It's schematic can be seen in **??**.

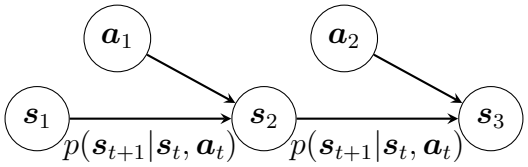


Figure 2.3: Schematic of a Markov decision process.

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 $\boldsymbol{a} \in \mathcal{A}$ is a continous or discrete action and r is the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. It should also be noted that now the transition operator is 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 $\mu_{t+1,i} = \sum_{j,k} \mathcal{T}_{i,j,k} \mu_{t,j} \xi_{t,k}$. Hence $\mathcal{T}$ remains its linearity.
Finally, partial observability also needs to be accounted for. To do so, a partially observable Markov decision process (POMDP) needs to be constructed. This is done by augmenting the Markov decision process to also include the observation space $\mathcal{O}$, where observations $\boldsymbol{o} \in \mathcal{O}$ denote the discrete or continous observations and the emission probability $\mathcal{E}$ which describes the probability $p(\boldsymbol{o}_t | \boldsymbol{s}_t)$ of getting the observation $\boldsymbol{o}_t$ when in state $\boldsymbol{s}_t$. The schematic can be seen in 2.4.

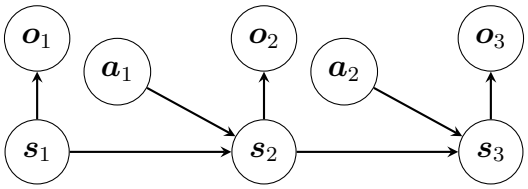


Figure 2.4: Schematic of a partially observable Markov decision process.

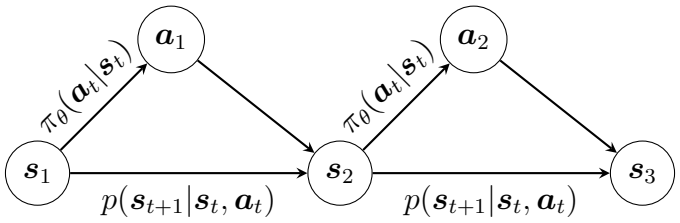
It is important to note that not all elements of POMDP are present in every problem: for example, the reward may be a determinstic function of the state and so on. In general through the text, to aid in simplifying notation, only the necessary elements will be explicity referenced in sketches and written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

2.1.4 Key concepts in reinforcement learning

2.1.4.1 Policy

With the problem space being formally defined, we may introduce definitions which will allow the construction of reinforcement learning algorithm. The reinforcement learning problem can be defined in finite or infinite time horizons. Different environments usually naturally fall in either category. For the agent to learn, it needs to be able to try out different actions from the same, or at least similar states. This is usually achieved by having the agent return to a set of starting states. The period between two such returns is called **an episode**. The agent selects actions based on

its **policy** π. The policy is a function which maps states to actions. The schematic showing it in the context of a Markov decision process is given in **??**.



The policy is a stochastic function. The intensity of stochasticity determines the trade-off between exploration and exploitation. To emphasize that the policy depends on some parameters θ, we usually write π_θ.

2.1.4.2 Goal of reinforcement learning

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

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

A bit more explicitely, we can a transition probaility as:

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

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

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

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

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

2.1.4.3 Value functions

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

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

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

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

The connection between the two is the following:

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

And we can also write the RL objective as:

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

2.2 Classes of reinforcement learning algorithms

2.2.1 Policy gradients

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

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

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

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

Simplifying the notation further, we get:

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

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

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

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

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

To evaluate the policy gradient we can sample:

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

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

REINFORCE algorithm:

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

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

2.2.1.1 Baselines

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

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

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

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

2.2.1.2 Off-policy gradients

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

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

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

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

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

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

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

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

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

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

2.2.1.2.1 Advanced policy gradient

The basic algorithm we have outline 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 algorithms such as PPO and TRPO, which will not be discussed here.

2.2.2 Actor-critic algorithms

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

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

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

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

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

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

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

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

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

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

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

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

The value function can be estimated through samples

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

After collecting many such samples

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

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

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

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

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

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

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

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

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

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

Actor-critic algorithm template

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

2.2.3 Value function methods

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

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

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

2.2.3.1 Dynamic programming

Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. They are of limited utility in reinforcement learing due to the perfect model requirement and their great computational expense, but are important theoretically — they provide an essential foundation for understanding the other methods. Usually a finite MDP is assumed. DP can be applied to continous problems as well, but exact solution exist only in special cases.
TODO:

1. Introduce value iteration in a tabular setting
2. introduce fitted value iteration
3. show that q-learning is off-policy
4. show that q-learning does not converge when using non-linear function approximation (introduce the bellman operator as well)

2.3 Deep Reinforcement Learning and DQN

(paraphrase)After deep learning has shown remarkable results in learning from raw pixel data in computer vision it's application has been adopted to solve many classical learning problems.The goal of Deep Reinforcement Learning is to connect reinforcement learning algorithm to deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates[2].

The use of label data and the assumption the distribution of data to be identical and independent through out training process in deep learning makes it complex to use it directly into reinforcement learning algorithms which must be able to learn from sparse ,noisy and delayed reward and highly correlated data.To address this issues and successfully apply deep learning to reinforcement learning [2] used experience replay mechanism[15] which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors.Convolutional neural network was used to learn successful control policies from raw video data in

complex reinforcement learning environment.The network is trained with a variant of Q-learning algorithm,with stochastic gradient descent to update the weights.
This architecture was based on Tesauro's TD-Gammon architecture [17] which updates the parameters of the network that estimates the value function directly from on-policy samples of experience. Similar to this approach the online network in DQN utilize a technique known as experience replay [18] where the agent's experiences at each time-step, $\rceil_t = (s_t, a_t, r_t, s_t + 1)$ is stored in a data set $\mathcal{D} = e1, ..., e_N$ pooled over many episodes into a replay memory. By drawing random samples from this pool of stored experiences the Q-learning is updated. After performing experience replay, the agent selects and executes an action according to an ϵ-greedy policy. The use of experience replay and target networks enables relatively stable learning of Q values, and led to super human performance on several Atari games.
The advantage of using Deep Q-learning over Q-learning includes allowing to have greater sample efficiency, reduced variance by randomising the sample bias and avoiding being stuck in local minimum.Drawback DQNs are it only handle discrete,low-dimensional action space.

2.3.1 Extension of DQN

Several Studies were performed to increase the performance of DQNs(Paraphrase).

2.3.1.1 Double Deep Q-networks: DDQN

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

2.3.1.2 Prioritized replay

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

2.3.1.3 Dueling Network

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

has a single Q-learning network with two streams that replace DQN architecture[3].

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

if needed TODO ADD EQUATION:factorization of action values

2.3.1.4 Multi-step learning

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

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

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

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

2.3.1.5 Noisy Nets

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

2.3.1.6 Integrated Agent:Rainbow

In the Rainbow architecture [16] tries combine the above six method stated above all together.Distributional loss (3) was replaced with a multi-step variant. The target distribution was constructed by contracting the value distribution in St+n according to the cumulative discount, and shifting it by the truncated n-step discounted return. multi-step distributional loss with double Q-learning by using the greedy action in St+n selected according to the online network as the bootstrap action $a \cdot t + n$, and evaluating such action using the target network.
TODO ADD EQUATION:target distribution

2.4 Problems with RL

ex. drl that matters paper

2.5 general computer vision stuff

2.6 general latent space learning

summarize this [Les+18]

2.7 Related Work

todo, not sorted: related to us

1. algs we use: [**erainbow**], [Haa+18]
2. (us) yarats improving sample efficiency [Yar+19], [All+21]
3. (us as inspiration) [Pat+17]
4. (not us but same high-level idea): data-augmentation like [Yar+21], contrastive loss like [LSA20], [Las+20], [Sha+21], [KYF20]
5. (not us, but things that can be done with image loss): [She+16]
6. (discuss some state of the art); [Bad+20]
7. (us, historic) deep Auto-Encoder Neural Networks in Reinforcement Learning [LR10]

sort these in the bellow two sections

2.7.1 Current state-of-the-art

2.7.2 Related to our work

3
Methods

In this section, we will explain the architecture of our auto-encoder and reinforcement learning algorithm. This includes a description of the environment and preprocessing in Section 3.0.1., the collection of training data for the auto-encoder and model architecture in section 3.0.2. and the training of the RL agent in Section 3.0.3.

3.0.1 Enviroment and Preprocessing

[!ADD MORE HERE]
We perform a comprehensive evaluation of our proposed method on the Arcade Learning Environment (Bellemare et al., 2013), which is composed of 57 Atari games. The challenge is to deploy a single algorithm and architecture, with a fixed set of hyper-parameters, to learn to play all the games given embedded latent space representation of the environment from auto encoder and game rewards. This environment is very demanding because it is both comprised of a large number of highly diverse games and the observations are high-dimensional.

Working with raw Atari frames, which are 210 x 160 pixel pictures with a 128 color palette, is computationally expensive, therefore we do a basic preprocessing step to reduce the input dimensionality. The raw frames are preprocessed by down sampling to a 110 x 84 picture and transforming their RGB representation to gray-scale.Cropping an 84 x 84 rectangle of the image that nearly captures the playing area yields the final input representation to encoder part of the auto encoder.

3.0.2 Deep Auto-encoder and Model Architecture

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

on the RL agent learning performance.We show this in more detail in the following sections.]

3.0.3 Training the RL Agent

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

Two step Training: First,we train the auto encoder with the prepossessed data and the weights of the encoder are saved in file for training the agent.In this set up the auto encoder is not updated such that we have a static representation of the environment.
second,we train the integrated agent with the results from the auto encoder.

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

4
Results

state the Experiments

4.0.1 Two Step Training

4.0.2 Parallel Training

5
Conclusion

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

5.1 Discussion

5.2 Conclusion

Bibliography

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

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

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

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

[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.

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

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

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

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

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

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

[Sha+21] Wenling Shang et al. "Reinforcement learning with latent flow". In: *Advances in Neural Information Processing Systems* 34 (2021).

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

A

Appendix 1

(UNFINISED)
//UPDATE THE FORMAT LATER

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