ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in
Artificial Intelligence in Industry

How Reinforcement Learning can improve Video Games Development: Dreamer and P2E Algorithms in the SheepRL Framework

CANDIDATE
Michele Milesi

SUPERVISOR
Prof. Michele Lombardi

CO-SUPERVISOR
MSc. Federico Belotti

Academic Year 2022/23

Session 2nd

"And in the naked light I saw
Ten thousand people, maybe more
People talking without speaking
People hearing without listening
People writing songs that voices never share
No one dared
Disturb the sound of silence"

To my family

Acknowledgements

I would like to thank my family and grandparents for always believing in me, supporting me, and never letting me lack anything.

Thanks to my friends for being part of my life, but most importantly, for accepting over 80-year-olds into the company.

I thank my classmates and friends, for sharing joys and labors in these two years spent together. Moments that I will always carry with me.

Special thanks to my Jedi master Federico, for constantly following me and teaching me so much in this experience, but especially for teaching me the art of the mustache.

Thanks to Lisa and Orobix, for placing trust in me, giving helpful advice and making me feel like one of them since I arrived.

I am thankful to Diego and Roberto for allowing me to be part of a wonderful reality where I have grown professionally.

Thank you to Virtus Gorle for being my second home.

Sommario

L'*Intelligenza Artificiale* (AI) nei videogiochi è un'area di ricerca di lunga data. Studia come utilizzare le tecnologie AI per ottenere prestazioni di livello umano quando si gioca. Da anni ormai, gli algoritmi di *Reinforcement Learning* (RL) hanno superato in performance i migliori giocatori umani nella maggior parte dei videogiochi. Per questo motivo è interessante investigare se il RL può essere ancora utilizzato nell'industria dei videogame oppure se il rapporto tra il RL e l'industria dei videogiochi debba rimanere puramente accademico.

Questo lavoro si concentra su due obiettivi primari nell'industria dei videogiochi: *(i)* Test e Debug: come il RL può essere utilizzato nell'industria dei videogiochi per scoprire bug latenti, valutare la difficoltà del gioco e perfezionare il design dei videogiochi? *(ii)* Creazione di *Personaggi non giocabili* (PNG): il RL è la strategia migliore per creare in modo efficiente i PNG o gli algoritmi di RL sono diventati troppo avanzati?

Questa tesi esplora la fattibilità dell'utilizzo dell'algoritmo Dreamer allo stato dell'arte per effettuare test automatizzati e per creare PNG per i videogiochi; inoltre, propone SheepRL un un framework open-source scalabile per l'esecuzione di esperimenti in modo distribuito.

Abstract

Artificial Intelligence (AI) in video games is a long-standing research area. It studies how to use AI technologies to achieve human-level performance when playing games. For years now, *Reinforcement Learning* (RL) algorithms have outperformed the best human players in most video games. For this reason, it is interesting to investigate whether RL can still be used in the video game industry or whether the relationship between RL and the video game industry should remain purely academic.

This work focuses on two primary objectives within the video game industry: *(i)* Testing and Debugging: how RL can be exploited in order to uncover latent bugs, assess game difficulty, and refine the design of the video game. *(ii) Non-Playable Characters* (NPC) Creation and Generalization: Is RL the best strategy to efficiently create NPCs or the RL algorithms have become too advanced?

This thesis explores the feasibility of using the state-of-the-art Dreamer algorithm in automated testing and NPCs creation for video games; in addition, it proposes SheepRL a scalable open-source framework for running experiments in a distributed manner.

Contents

List of Figures xv

List of Tables xxiii

Acronyms xxv

1 Introduction 1

1.1 Bridging AI and Neuroscience . . . 1

1.2 The Dreamer Algorithm: Imagination in RL . . . 2

1.3 SheepRL: A Scalable Framework for RL . . . 2

1.4 RL in the Video Game Industry . . . 2

1.5 Outline . . . 3

2 Reinforcement Learning 5

2.1 Overview . . . 5

2.2 Markov Decision Process . . . 7

2.3 Goal and Returns . . . 10

2.4 Values and Policy . . . 13

2.4.1 Optimality . . . 16

2.5 On-policy vs Off-policy . . . 20

2.6 Value Function Approximation . . . 23

2.7 Policy Gradients and Actor-Critic Methods . . . 25

2.7.1 Policy Gradient methods . . . 26

2.7.2 Actor-Critic Methods 28

3 State of the Art 31

3.1 Model-free Algorithms 31

3.2 Model-based Algorithms 34

4 Dreamer 37

4.1 Components 38

4.2 Dreamer V1 39

4.2.1 Dynamic Learning 40

4.2.2 Behaviour Learning 45

4.2.3 Actor 49

4.2.4 Environment Interaction 51

4.2.5 Buffer 52

4.2.6 Critical Aspects 53

4.3 Dreamer V2 57

4.3.1 Stochastic States 58

4.3.2 KL Loss 59

4.3.3 Straight-Through Gradients 60

4.3.4 Critic 60

4.3.5 Actor 61

4.3.6 Buffer 62

4.4 Dreamer V3 63

4.4.1 Differences with respect to Dreamer V2 63

4.5 Plan2Explore 70

4.5.1 Ensembles 70

4.5.2 Intrinsic Reward 71

4.5.3 Zero-shot vs Few-shot 72

5 Tools and Technologies 73

5.1 SheepRL . . . 74

5.1.1 Coupled vs Decoupled . . . 74

5.1.2 Environments . . . 77

5.1.3 Algorithms . . . 85

5.1.4 Buffers . . . 87

5.2 PyTorch Lightning, Fabric . . . 88

5.3 Hydra . . . 88

6 Experiments and Results 91

6.1 Correctness of Implementations . . . 92

6.1.1 DreamerV1 . . . 92

6.1.2 Dreamer-V2 . . . 94

6.1.3 Dreamer-V3 . . . 94

6.2 Generalization . . . 97

6.3 Automatic Testing . . . 99

7 Conclusions 103

7.1 Next Steps . . . 104

Bibliography 105

List of Figures

2.1 The interaction between the environment and the agent. The agent provides the state and the reward; the agent exploits the state to select the action to perform. Then the environment returns the next state and the next reward [55]. 9

2.2 The episodic tasks can be seen as continuing ones by adding an absorbing state after the terminal state, and a unique reflexive transition with zero reward. In this case, the episode ends at time step $T = 3$, and then the absorbing state, represented as a grey square, allows the episode to continue indefinitely in a fictitious way. Indeed, the return is equal whether we sum over the first $T = 3$ rewards or over the full infinite sequence, even if we introduce the discount. 13

2.3 Starting from the state s, the agent can select 3 different actions. For each action, the dynamics of the environment, represented by the transition probability function p, can lead to 2 different states, giving different rewards. This picture shows how the *Bellman equation* averages over all the possibilities, weighting each by the probability of occurring: the probability of selecting the action a (given by the policy π) and the probability that the next state is s' with reward r. 16

2.4 Diagrams of the optimal state-value function (left) and the optimal action-value function (right). On the left, the optimal state value is the maximum among the expected returns obtained from all the possible actions. On the right, the optimal action value is given by the weighted sum of the maximum expected return obtained from the next state pair (s', a'); the weights are given by the probabilities of occurring in state s' and selecting the action a', starting from (s, a). 18

2.5 How TD(n) targets are weighted in TD(λ)-returns: if $\lambda = 0$, then the overall update reduces to its first component, the one-step *Temporal Difference* (TD) update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the *Monte Carlo* target [55]. 25

2.6 Weighting given in the λ-return to each of the n-step returns [55]. 26

4.1 The structure of Dreamer-V1: on the left is the "dynamic learning", in which the agent learns the dynamics of the environment. In the center is "behavior learning", in which the agent learns how to optimally behave and learns to correctly estimate state values through imagination. On the right is the "environment interaction", in which the agent collects new data for training. 38

4.3 The RSSM is composed of the Recurrent Model, the Transition Model, and the Representation Model. These three models are iteratively used to learn the dynamics of the environment. First, the Recurrent Model computes the history of the episode, taking in input the previous history (i.e., the recurrent state h_{t-1}), the previous action a_{t-1} and the previous posterior s_{t-1}. Then, the transition model predicts the stochastic state (prior denoted with $\hat{s}_t$) from the new history computed by the recurrent model. Finally, the representation model computes the posterior state s_t from the history h_t and the observations o_t provided by the environment. 42

4.5 The Actor is responsible for selecting actions given latent states, indeed, it approximates the policy. Latent states are the only information it needs to choose actions. The stochastic state can be either the prior or the posterior. In the first case, the actor selects an action based on the imagined stochastic state; whereas, in the second case, it selects the action based on more precise information about the current state. 46

4.6 Imagination Phase: the agent starts from "real" latent states, i.e., posteriors concatenated with recurrent states (the history of the episode). It selects an action $\hat{a}_0$, then the RSSM computes the new recurrent state h_1 and the transition model computes the prior state $\hat{s}_1$. Then it iteratively performs these imagination steps up to a certain horizon H. The imagined trajectory will be used to learn the actor and the critic. 47

4.7 The Critic approximates the state-value function: given a latent state, it estimates the values of the values of that state. 48

4.8 The possible scenarios during sampling from the Sequential Replay Buffer: *(i)* On the top, the case in which the buffer is not full, so the sequence can start in the green area and continue in the grey one, without falling into the red area. *(ii)* On the bottom, the two cases where the buffer is full, on the left of the index `pos` there are new collected experiences, instead, on the right there are old experiences. It is possible to start sequences in the green area, so as to be sure not to mix new and old experiences into a single trajectory. . 54

4.9 This picture [27] shows the performance on the Atari environments, showing how Dreamer-V2 achieves superior performance compared to the best model-free algorithms and other model-based algorithms present at the time. Moreover, it shows how Dreamer-V2 is able to outperform humans in the Atari environments. 57

4.10 The world model of Dreamer-V2 [27]. It is possible to notice that it maintains the idea of Dreamer-V1: the observations are encoded and used to compute the posterior z_t. The prior $\hat{z}_t$ is computed from the history h_t (i.e., the recurrent model) and it is learned to be similar to the posterior. From the latent state (z_t concatenated with h_t), the observations are reconstructed $\hat{x}_t$ and the reward $\hat{r}_t$ is predicted. Finally, the posterior and the prior are now represented as 32 categoricals of 32 classes each. 58

4.11 The algorithm of the Straight-Through Gradients with Automatic Differentiation trick: the sample is extracted from the Categorical distribution, then the probabilities are computed. Finally, these probabilities are summed and subtracted from the sample. When we subtract the probabilities, we need to stop the gradients, to prevent them from nullifying those of the probabilities that were summed. 60

4.12 The dynamic and the behavior learning in Dreamer-V3: as in Dreamer-V2, the prior and the posterior are represented by various categorical distributions. The world model is used to learn the dynamics of the environment and it computes the latent states (z_t concatenated to h_t). The latent states are exploited to imagine future scenarios that are used to learn the optimal behavior and the approximated optimal state-value function in the imagined trajectories. 64

4.13 The *symlog* function: it compresses the magnitudes of both large positive and negative values, allowing to quickly move the network predictions to large values when needed. 65

4.14 The `uniform_mix` function applied to all categorical distribution, i.e., to the distribution of discrete/multi-discrete actions and to the distribution of the stochastic states (i.e., posterior and prior). The logits are first converted into probabilities, then the probabilities returned by the *Neural Network* (NN) are mixed with the uniform probabilities. Finally, the new computed probabilities are converted into *logits* and returned. 67

4.15 The Ensemble: it takes in input the latent state, and the previous action to estimate the next observation it will receive from the environment. The latent state is composed of the posterior s_t concatenated with the recurrent state h_t. Several ensembles are exploited to compute the novelty of a state: the greater the disagreement between them, the newer the state. Indeed, the more time a state is visited, the more trained the ensembles are to predict the next observation from that state, so, the more precise and similar will be their predictions. 71

5.1 The coupled version of the algorithms: several processes can be used to distribute training. All the processes contain both the environment and the agent, which means that different instances of the environment are created. Each process interacts with its local instances of the environment, and then each process executes the training loop: the parameters of the agent models are shared and synchronized thanks to Lightning Fabric. 75

5.2 The decoupled version of the algorithms: several processes can be spawned to distribute training (at least two processes). The zero-rank process (called *player*) contains m instances of the environment and performs the environment interaction. Whereas, all the other processes (called *trainers*) execute the training loop. The player interacts with the environment and collects experiences that are sent to the trainers. The trainers receive the data from the player and execute the training loop: again, Lightning Fabric synchronizes the weights of the models during training. At the end of the training loop, the rank-1 process (a trainer) sends the updated weights to the player. . . . 76

5.3 The walker walk environment provided by the DM Control Suite. The agent has to learn how to walk in the forward (right) direction by applying torques on the six hinges connecting the seven body parts. 79

5.4 The MsPacman Atari environment: the agent has to collect all of the pellets on the screen while avoiding the ghosts. 80

5.5 The CarRacing environment is the easiest control task to learn from pixels: a top-down racing environment. The agent has to learn to complete a lap as quickly as possible. 80

5.6 The "Dead or Alive ++" environment: the agent has to challenge different opponents. To advance to the next stage, he must defeat the same opponent twice (it has to win two rounds); when he passes the 8th stage, the agent has won the game. If he loses 2 games in the same stage, though, then the agent has lost and the episode ends. 81

5.7 The Crafter environment: the agent receives only the image as observation, it has to retrieve the information from it, including the life level, hunger level, and inventory. 82

5.8 The Navigate task of MineRL: the agent has to follow the compass to find the diamond in the environment, it is the light blue square located at the bottom left of the figure. 83

5.9 A part of the default configuration of the Dreamer-V3 algorithm. It is possible to notice that it is possible to define any type of parameter in a hierarchical manner, being able to assign the same value to a parameter as to another parameter. 89

6.1 Walker Walk: on the left the result obtained by the implementation of Dreamer-V1 in SheepRL (PyTorch); instead, on the right, the result published in [24] (TensorFlow). 93

6.2 MsPacman: on the left the result obtained by the implementation of Dreamer-V1 in SheepRL (PyTorch); instead, on the right, the result published in [24] (TensorFlow). As one can notice, we performed many fewer steps with respect to the original work because of the limited availability of resources. 93

6.3 The rewards obtained by Dreamer-V2 in the Walker Walk environment. . . 94

6.4 Pong: on the left the result obtained by the implementation of Dreamer-V2 in SheepRL (PyTorch); instead, on the right, the result published in [27] (TensorFlow). As one can notice, we performed many fewer steps with respect to the original work because of the limited availability of resources. 95

6.5 MsPacman: on the left the result obtained by the implementation of Dreamer-V3 in SheepRL (PyTorch); instead, on the right, the result published in [28] (TensorFlow). The graphs show a different number of steps, this is simply due to the measurement used on the x-axis: both experiments are performed with the same hyper-parameters and take the same number of steps. On the right, the steps correspond to the number of times the actor chooses an action (policy steps); while on the left, the steps correspond to the steps played in the environment. The values differ because of the action repeat parameter, set to 4; in fact, 400K/4 = 100K steps. 95

6.6 Boxing: on the left the result obtained by the implementation of Dreamer-V3 in SheepRL (PyTorch); instead, on the right, the result published in [28] (TensorFlow). The graphs show a different number of steps, this is simply due to the measurement used on the x-axis: both experiments are performed with the same hyper-parameters and take the same number of steps. On the right, the steps correspond to the number of times the actor chooses an action (*policy steps*); while on the left, the steps correspond to the steps played in the environment (*environment steps*). The values differ because of the action repeat parameter, set to 4; in fact, 400K/4 = 100K steps. 96

6.7 The results obtained by the SheepRL Dreamer-V3 (on the left) and the results obtained by its original implementation [28] (on the right) in the Crafter environment. The agent shows a strong ability to generalize (newly generated maps for each episode), to deal with partial observability (each input image reveals only a small part of the world), and to long-term reasoning and survival. 96

6.8 The reward obtained in the Navigate task. As it is possible to notice, Dreamer-V3 is able to follow the information of the compass, but it has some difficulties in collecting the diamond. Indeed, the reward shows that some episode ends with a cumulative reward greater than 100 (the reward has been collected) and other with a cumulative reward lower than 100. In most of the episodes, the agent receives a reward of (almost) 60, meaning that it has followed the information in the compass. 98
6.9 The portion of the Minecraft map explored by the P2E agent. The orange indicates the locations visited by P2E; whereas, the red part indicates the visited locations of the random agent. 100
6.10 The reward obtained by Dreamer-V3 in the "Dead or Alive ++" environment. 101
6.11 The actions distribution of Dreamer-V3 for completing the "Dead or Alive ++" game. As one can see, it is possible to win the game by repeatedly executing the same actions. 101

List of Tables

3.1 A survey of the SOTA algorithms in RL. 36

5.1 The environments available in SheepRL: the "Obs" column can be: Both (both pixels and vector observations), Depends (on the specific environment), Pixels (only images as observation), Vector (only vectors as observations). The "Action Space" column can be Continuous, Discrete, Multi-Discrete, or All (for all the action spaces). Finally, the "Category" column can be Depends (on the specific environment), Fully Described, or Partially Observable. . . 85

5.2 The algorithms in SheepRL. The *Type* column can be "Coupled", "Decoupled" or "Both"; the *Recurrent* one specifies whether or not the algorithm contains a recurrent model (such as LSTM or GRU); the *Observations* column can be "Image", "Vector" or "Both"; and the *Action Space* can be "Continuous", "Discrete", "Multi-Discrete" or "Both". 86

6.1 The generalization capabilities of Dreamer-V1 and SAC on the CarRacing environment. The track number 42 is the one on which the two algorithms were trained. The other three tracks have never been seen by the two agents. From the rewards, it is evident that Dreamer is still able to complete the lap quickly, while SAC struggles more. Considering that this environment is simple, we expect this difference to increase as the difficulty level rises. The underlined scores refer to the maximum score achieved by an algorithm on that particular track. 99

Acronyms

AI *Artificial Intelligence*

RL *Reinforcement Learning*

DRL *Deep Reinforcement Learning*

ML *Machine Learning*

SOTA *State of the Art*

MDP *Markov Decision Process*

MC *Monte Carlo*

TD *Temporal Difference*

BE *Bellman equation*

BOE *Bellman optimality equation*

NN *Neural Network*

DNN *Deep Neural Network*

RSSM *Recurrent State-Space Model*

MLP *Multilayer Perceptron*

LSTM *Long Short-Term Memory*

GRU *Gated Recurrent Unit*

SL *Sequence Length*

BS *Batch Size*

KL *Kullback-Leibler*

FIFO *First In First Out*

LN *LayerNorm*

EMA *Exponential Moving Average*

P2E *Plan2Explore*

ALE *Arcade Learning Environment*

NPC *Non-Playable Characters*

UTD *Update-to-Data*

CHAPTER 1

Introduction

The intersection of AI and the video game industry has given rise to a captivating and transformative realm where digital worlds are brought to life through the lens of *Machine Learning* (ML). Within this burgeoning field, RL stands as a pivotal force, promising not only to revolutionize gameplay but also to unlock novel avenues of exploration and innovation. This research embarks on a journey through this dynamic landscape, charting the course of the evolution of RL in the video game industry and examining its potential applications in testing, debugging, and the creation of NPCs through environment generalization.

1.1 Bridging AI and Neuroscience

At its core, RL draws inspiration from the intricacies of human learning and decision-making processes [55]. It is founded upon the principle of learning through interaction with an environment, wherein an agent undertakes actions to maximize cumulative rewards. This profound paradigm draws parallels with the neural mechanisms that underpin human cognition, making it an enticing avenue for study and application in fields as diverse as neuroscience and robotics.

The relationship between RL and neuroscience is particularly intriguing. The ability of RL agents to acquire optimal behaviors through latent imagination, wherein they simulate the consequences of actions and compute latent imagined states, mirrors the human capacity

for mental simulation, prediction, and planning. By bridging the gap between AI and neuroscience, RL presents opportunities for advancing AI-driven solutions.

1.2 The Dreamer Algorithm: Imagination in RL

At the heart of this exploration lies the Dreamer algorithm [24, 27, 28], a model-based RL algorithm that learns the optimal behavior through imagination. It enables RL agents to envision the consequences of their actions, fostering an iterative process of action refinement. This mechanism, reminiscent of human imagination, endows RL agents with the ability to learn optimal behaviors through a remarkable blend of trial and imagination.

1.3 SheepRL: A Scalable Framework for RL

Supporting this quest for the integration of RL into the video game industry is SheepRL, an open-source framework engineered to simplify experimentation and scalable application. In the world of game development, code comprehensibility and scalability are paramount. SheepRL addresses these concerns by providing a versatile platform that facilitates experiment management, code clarity, and extensibility. Furthermore, SheepRL goes a step further, offering the capability for distributed training of RL agents through the Lightning Fabric, thereby harnessing the power of parallel computing for accelerated learning.

1.4 RL in the Video Game Industry

This thesis embarks on a multifaceted exploration of the potential applications of RL within the video game industry. Specifically, it delves into the realms of testing and debugging, where RL can unveil latent bugs, assess game difficulty, and offer critical insights into game design. Additionally, it explores the creation of NPCs, a complex endeavor often fraught with resource-intensive processes. Here, the role of RL lies in generalizing environments to reduce the laborious training time required for NPC development.

As we traverse this exciting intersection of RL, video games, and computational imagination, we strive to unveil the transformative potential of RL while navigating the practical challenges that accompany its integration into the ever-evolving video game industry. This journey promises to illuminate new horizons and reshape the way we conceive and experience video games.

1.5 Outline

In Chapter 2 the main concepts of RL are presented to be able to understand the rest of the thesis; whereas in Chapter 3 the *State of the Art* (SOTA) RL algorithms are described, focusing on the advantages and disadvantages of each. In Chapter 4 Dreamer and *Plan2Explore* (P2E) [51] algorithms are analyzed in detail and the PyTorch implementations used for executing the experiments are presented; this implementation has been incorporated into the SheepRL framework, described in Chapter 5. Finally, in Chapter 6 experiments are presented and the results obtained are commented on; instead, in Chapter 7 conclusions about the work done are given.

CHAPTER 2

Reinforcement Learning

RL has always been associated with video games, this is because RL is perfectly suited to environments such as video games, where there is an explicit task to solve. It is precisely for this reason, in fact, that the online nature of RL lends itself perfectly to solving problems where there is a stated goal; learning, therefore, goal-oriented behaviors.

This chapter introduces RL and its related concepts. From the RL definition, passing to the *Markov Decision Process* (MDP), the definition of *policy*, and finishing with the difference between *on-policy* and *off-policy* algorithms, explaining the concepts of *Temporal Difference*, and *Advantages*. Finally, a general overview of the techniques used for solving RL problems, such as *action-values*, *policy gradient*, and *actor-critic* methods.

2.1 Overview

The RL is one of the ML paradigms; the goal is to learn what to do to maximize a numerical reward signal, thus we want to map situations to actions. It comprises two components that iteratively interact with each other: the *actor* and the *environment*. The actor is the learner and has to solve a specific goal by selecting actions ($A \in \mathcal{A}$), whereas the environment is everything the agent interacts with. At each time step ($t \in \mathbb{N}$), the agent selects actions and the environment responds to those actions and presents new situations to the agent (observations $O \in \mathcal{O}$), moreover, the environment provides the rewards ($R \in \mathcal{R} \subset \mathbb{R}$, i.e., the numerical

quantity the agent seeks to maximize). Thus, the agent affects the subsequent data it receives from the environment.

The iterative interaction between the agent and the environment forms a history, also known as *trajectory*:

$$\tau \doteq O_0,\ A_0,\ R_1,\ O_1,\ A_1,\ R_2,\ O_2,\ A_2,\ R_3,\ldots$$

Both the action selected by the agent and the observation and reward returned by the environment depend on the trajectory τ. Suppose now to have a trajectory τ_t up to a certain *time step* $t > 0$:

$$\tau_t \doteq O_0,\ A_0,\ R_1,\ O_1,\ A_1,\ R_2,\ldots,O_{t-1},\ A_{t-1},\ R_t, O_t$$

We can define the state $S_t \in \mathcal{S}$ as follows:

$$S_t \doteq f(\tau_t)$$

where f is any function of the trajectory.

RL is not a kind of *supervised learning* because the learner is not told which actions to take, but it has to find out which actions lead to the greatest reward. Moreover, someone can think that the RL is a form of *unsupervised learning* algorithm because it seeks to find out the optimal behavior without correct examples, but uncovering the hidden patterns in an agent's experience is not enough, indeed, by itself cannot address the RL problem of maximizing the reward signal. For this reason, the RL agents cannot be considered as *unsupervised learning* algorithms.

Since the goal of the agent is to maximize the numerical reward signal, it is important to select actions that affect not only the intermediate reward but also the following rewards, indeed, it may be more convenient to select an action that immediately gives the agent a lower reward with a good outlook for future rewards, rather than choosing an action that immediately gives a higher reward with a worse prospect of future rewards.

As mentioned before, the agent has to learn which actions to select to maximize the reward, i.e., mapping the situations to actions, in particular, the agent has to evaluate the actions taken rather than learn them from correct actions (*online* RL). This leads the agent to perform a *trial-and-error* search, leading to a key challenge in RL: the trade-off between *exploration* and *exploitation*. To get a high reward, the agent should select actions that it knows are effective for obtaining high rewards because it has already tried them in the past; but to find those actions, it needs to select actions it has never tried before. Thus, it has to trade-off between the exploration of new actions and the exploitation of known actions.

Another aspect to consider when discussing RL is the distinction between problems and solution methods. The *problem* is defined through an MDP that captures the most important aspects of the problem facing an agent interacting over time with its environment to achieve a goal. In particular, the agent should be able to perceive the state of the environment and take actions that affect the state of it, in order to solve its goal. This leads the MDPs to include three main aspects: the sensation of the state, the action, and the goal (or goals). However, the MDP will be explained in detail in Section 2.2.

2.2 Markov Decision Process

As introduced in the previous section, the interaction between the agent and environment forms a trajectory that can be used to compute states. There are two kinds of states: the *environment state* (S_t^e), i.e., the private state of the environment; and the *agent state* (S_t^a), which is the representation internal of the agent [52]. The first one is not visible to the agent, whereas the second one is any function of the trajectory τ_t.

Definition 2.1 (Markov State)**.** A state S_t is Markov if and only if

$$\Pr[S_{t+1} \mid S_t] = \Pr[S_{t+1} \mid S_0, S_1, \dots, S_t]$$

The environment state S_t^e is always a Markov state, whereas the agent state S_t^a could be a non-Markov state. This leads to different types of environments: *(i) fully described*

environments and *(ii) partially observable* environments. In the first case the observation is equal to the environment state that is equal to the agent state ($O_t = S_t^e = S_t^a$), whereas in the second case, the agent state is different from the environment state. The agent must construct its own state in one of the following three ways:

1. $S_t^a = \tau_t$, indeed, the trajectory is always Markov.

2. Use a probabilistic or Bayesian approach to estimate the agent state.

3. Use a recurrent NN.

An example of a fully described environment is “Chess”, the agent knows all the information about the position of its pieces and the position of the pieces of the opponent; whereas the game of “Poker” is an example of a partially observable environment, in which the agent knows only its own cards, it is not able to know the cards of the opponents.

An environment of a RL problem can be described either through a MDP or a *Partially Observable* MDP, if the environment is fully described or partially observable, respectively.

Definition 2.2 (Markov Decision Process)**.** A Markov Decision Process is a tuple

$$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$$

Where:

- $\mathcal{S}$ is a set of states.

- $\mathcal{A}$ is a set of actions.

- $\mathcal{P}$ is a state transition probability function:

$$\mathcal{P}_{ss'}^a = \Pr\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$$

- $\mathcal{R}$ is a reward function:

$$\mathcal{R}_s^a = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$

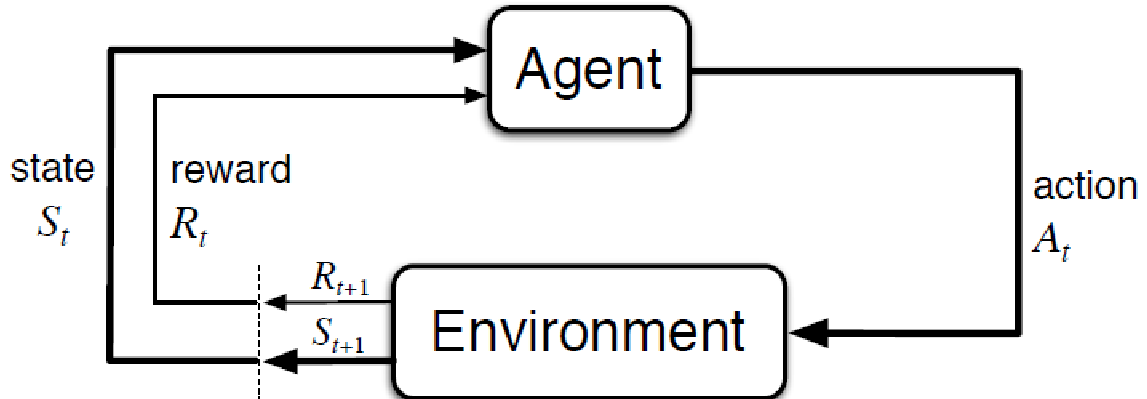


Figure 2.1 The interaction between the environment and the agent. The agent provides the state and the reward; the agent exploits the state to select the action to perform. Then the environment returns the next state and the next reward [55].

- $0 \leq \gamma \leq 1$ is the *discount factor* or discount rate[1].

The transition probability P and the reward functions together define the *dynamics* of the MDP. In other words, the MDP provides the next state and the reward given the previous state and the previous action. Thus, the MDP completely characterizes the dynamics of the environment: the transition probability function and the reward function are the two things that we need to solve any problem of learning goal-directed behaviors, indeed, each of these problems can be reduced to the exchange of three signals between the agent and the environment: the states (the information exploited by the agent to make decisions), the rewards (defines the agent's goal), and the actions (the choices of the agent). In Figure 2.1 it is possible to see how exactly the environment and the actor interact with each other.

Definition 2.3 (Dynamics of an MDP)**.** Given an MDP, its dynamics are defined by the function p:

$$p(s', r|s, a) \doteq \Pr\left\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\right\} \text{ for all } s', s \in \mathcal{S},\, r \in \mathcal{R},\, a \in \mathcal{A}\,.$$

[1]The reason why we need a discount factor will be better explained in the Section 2.3.

2.3 Goal and Returns

Further to my previous comments, the reward is the only learning signal the agent uses to solve the task. The two main characters of RL have to deal with the reward: *(i)* the environment has to define a proper learning signal in order to let the agent learn how to solve the task; whereas *(ii)* the agent has to interpret in the correct way the signal to find the optimal behavior.

In the case of the environment, a poorly designed reward most likely leads the agent to learn sub-goals or incomplete behaviors. For instance, suppose to train an agent to play chess. A good choice for the reward function could be the following one:

$$\mathcal{R}_1 = \begin{cases} +1 & \text{for} \quad \text{winning} \\ -1 & \text{for} \quad \text{losing} \\ 0 & \text{for} \quad \text{drawing and non-terminal moves} \end{cases}$$

In this case, we have designed the reward in a way that reflects our goal, i.e., to win at chess. In fact, the agent learns to maximize the reward, thus, to win the chess game. Let us take now another reward function:

$$\mathcal{R}_2 = \begin{cases} +1 & \text{for} \quad \text{winning} \\ +n & \text{for} \quad \text{taking opponent's pieces} \\ -1 & \text{for} \quad \text{losing} \\ 0 & \text{for} \quad \text{drawing and non-terminal moves} \end{cases}$$

Where $n > 0$ depends on the importance of the opponent's piece, in this second example, the agent could learn a strategy that seeks to take the opponent's pieces, without considering the goal of the game, i.e., to win the match. In particular, the agent is led to solve sub-goals, instead of achieving our goal. For this reason, the reward must be designed so that if the agent maximizes it, it also solves the task.

On the other hand, the agent should exploit the reward signal to achieve a goal. On one side it has to consider the next reward it will receive, however, it must also take into

consideration the influence it will have on subsequent rewards. In other words, the agent has to maximize the so-called *expected reward*.

Definition 2.4 (Expected Return)**.** Given a sequence of rewards $R_1, R_2, \ldots, R_T$, the expected return is defined as follows:

$$G_t \doteq R_t + R_2 + \cdots + R_T$$

Where:

- t is the index of the current time step.
- T is the final step time and can be either $T = +\infty$ or $T \in \mathbb{N}^+$
- $\mathbb{N}^+$ is the set of natural numbers, zero excluded (i.e., $\mathbb{N} \setminus \{0\}$).

In the case of $T = +\infty$ we are in a *continuing task*, otherwise, we are in an *episodic task*. The latter needs a *terminal state* that is the state in which the episode ends. This terminal state must be followed by a reset of the environment and a distribution of starting states (i.e., a distribution over the first state S_0). For episodic tasks, the interaction between the environment and the agent breaks naturally in *episodes*: each one starts from a starting state and ends in the terminal state; whereas, in continuing tasks, the environment-agent interaction does not break naturally in episodes, it simply continues without limit.

You can immediately see that for continuing tasks, the definition of expected return introduced before is not suitable, because it could easily be infinity. For this reason, we need a more robust definition to take into account future rewards. With this new concept, the agent seeks to select actions so that the sum of the *discounted* rewards it receives over the future is maximized.

Definition 2.5 (Expected Discounted Return)**.** Given a sequence of rewards $R_1, R_2, \ldots, R_T$, and a discount factor (or discount rate) $\gamma \in [0, 1]$ the expected return is defined as follows:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{T} \gamma^k R_{t+k+1}$$

Where, as usual, T can be either $T = +\infty$ or $T \in \mathbb{N}^+$

The discount rate determines the level of importance of future rewards, if $\gamma = 0$, then the agent is said to be "myopic" since it considers only the immediate reward. In this particular case, the agent selects the action A_t only by considering the next reward R_t. Unfortunately, this approach does not seem to work for general cases; on the contrary, it reduces access to future rewards, thus reducing the final return. On the other hand, as γ approaches 1, the return objective takes future rewards into account more strongly and the agent becomes more forward-looking.

Now that we have introduced the two types of tasks, it could be useful to uniform the two notations, indeed, it is useful to be able to refer to one or the other type of task, without necessarily having to change the notation or having to specify which task is being referred to. Let us consider the case of episodic tasks: in a single training, there could be more than one episode, so we should refer to each state by considering the time step t and the number of episodes i. So, we have to refer to $S_{t,i}$ the state at time step t of episode i. However, it turns out that when discussing episodic tasks, we rarely have to distinguish between different episodes. Instead, it is more useful to consider only one episode or state something that is true for all the episodes. For this reason, from this point forward, we will refer to episodic tasks by considering only a generic episode (i.e., we will write S_t to refer to $S_{t,i}$) [55].

To uniform the two notations, we still need a little more expediency: in episodic tasks, we sum a finite number of rewards, instead, in continuing tasks, we sum over an infinite number of terms. The solution is to add an *absorbing state* after the episode termination, with a unique reflexive transition that generates only zero rewards, as shown in Figure 2.2.

To resume, we can define the *expected discounted return* at time step t as:

$$G_t \doteq \sum_{k=0}^{T} \gamma^k R_{t+k+1}$$

With either $T = +\infty$ or $T \in \mathbb{N}^+$, and $\gamma \in [0,1]$, for both the episodic and continuing tasks.

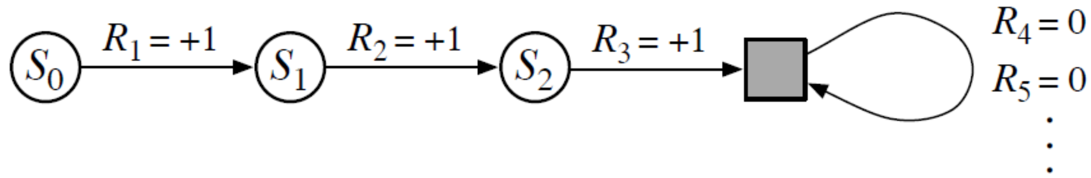


Figure 2.2 The episodic tasks can be seen as continuing ones by adding an absorbing state after the terminal state, and a unique reflexive transition with zero reward. In this case, the episode ends at time step $T = 3$, and then the absorbing state, represented as a grey square, allows the episode to continue indefinitely in a fictitious way. Indeed, the return is equal whether we sum over the first $T = 3$ rewards or over the full infinite sequence, even if we introduce the discount.

2.4 Values and Policy

Since the agent has to maximize the expected return, some questions arise: how can we estimate the expected return? How good a state is for the agent? Which are the best actions for a given state? We can answer these questions by introducing two fundamental concepts for RL: the *value functions* and the *policy*. The first ones are functions of states or state-action pairs, that given a state (or a state-action pair) estimate how good a state (or the state-action pair) is for the agent in terms of future rewards, i.e., in terms of expected return. Clearly, the expected return is influenced by the action the actor will take, so a value function is defined with respect to a particular way of acting, called policy.

Definition 2.6 (Policy)**.** A *policy* π is a mapping from states to probabilities of selecting each possible action at time step t. So, $\pi(a|b)$ is the probability of selecting the action $A_t = a$ in the state $S_t = s$ at time step t.

In particular, the policy $\pi(a|b)$ defines a probability distribution over $a \in \mathcal{A}$ for each possible state $s \in \mathcal{S}$. Now, we can formally define the value function of a state under a policy π.

Definition 2.7 (State-Value Function)**.** Given a policy π, the *value function* of a state s under the policy, denoted as $v_\pi(s)$, is the expected return when starting is s and following the policy π to select the subsequent actions.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1}|S_t = s\right], \text{ for all } s \in \mathcal{S} .$$

Where:

- $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows the policy π
- t is any time step.
- As usual, either $T = +\infty$ or $T \in \mathbb{N}^+$.

It is important to notice that the value of the state-value function for the final state, if any, is always zero.

Similarly to the state-value function, it is possible to define the *action-value* function, which for each state-action pair denotes the expected return. In other words, it denotes the expected return of taking an action a starting from the state s under the policy π.

Definition 2.8 (Action-Value Function)**.** Given a policy π, the value function of taking an action a in a state s under the policy, denoted as $q_\pi(s,a)$, is the expected return starting from s, taking the action a and thereafter following the policy π.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right], \text{ for all } s \in \mathcal{S}, a \in \mathcal{A} .$$

Now, that we have defined the value functions, we can introduce a fundamental property of them, very useful in RL. This is the recursive property of the value functions, called the *Bellman equation* (BE).

Definition 2.9 (Bellman equation for v_π)**.** For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}\left[G_t | S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s', r | s, a\right) \left[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s', r | s, a\right) \left[r + \gamma v_\pi(s')\right].
\end{aligned}
$$

Where:

- $a \in \mathcal{A}(s)$, i.e., the set of the actions that the agent can select when it is in the state s.
- $s' \in \mathcal{S}$.
- $r \in \mathcal{R}$.

As it is possible to notice in the above definition, the BE expresses the relationship between the value of a state and the value of its successor states: from a state s, by following a policy π, there are several successor states s'_i, depending on the transition probability function p (that defines the dynamics of the environment); each one leading a reward r_i; where $i \in \left[0, \sum_{a \in \mathcal{A}(s)} \sum_{(s',r) \in p(s,a)} 1\right) \subseteq \mathbb{N}$ is the index of the state-reward pair in which the agent can end up from the state s. In other words, the BE averages over all the possibilities, weighting each by the probability of occurring, as shown in Figure 2.3.

The *state-value* or the *action-value* functions can be computed in different ways: one of them is through *Monte Carlo* (MC) methods. MC methods consist of estimating v_π and q_π from experience: you take the mean of the actual return for each state encountered. This mean will converge to the real state-value $v_\pi(s)$, as the number of times the state s is encountered approaches infinity. Likewise, it is possible to hold the average of each state-action pair encountered; this average will tend to the value of $q_\pi(s,a)$, as the number of times in the state s the agent will take the action a approaches infinity.

The MC methods can be used when there is a suitable number of states or state-action pairs; if there are too many states (or state-action pairs), it is necessary to find another solution

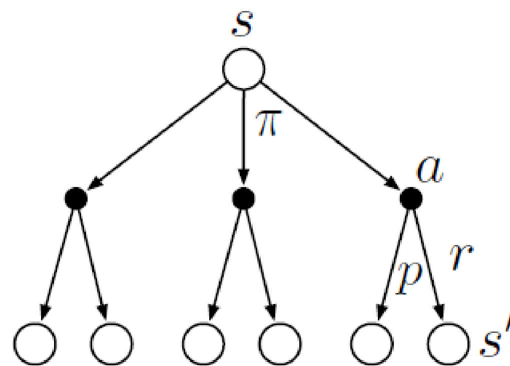


Figure 2.3 Starting from the state s, the agent can select 3 different actions. For each action, the dynamics of the environment, represented by the transition probability function p, can lead to 2 different states, giving different rewards. This picture shows how the *Bellman equation* averages over all the possibilities, weighting each by the probability of occurring: the probability of selecting the action a (given by the policy π) and the probability that the next state is s' with reward r.

to compute the value function. A possibility, that produces an accurate estimation of the v_π and q_π functions, is to parameterize them, with a number of parameters lower than the number of states.

2.4.1 Optimality

It is clear that the agent has to find the optimal policy, i.e., the policy to follow to get the highest reward. This optimal policy (π_*) is guaranteed to exist for an MDP. First of all, it is necessary to define a method to compare the policies and to establish which one is better. A practical way is to compare the expected returns (given by the value functions of the policies). In particular, if a policy π has a greater than or equal to expected return with respect to another policy π' for all the states $s \in \mathcal{S}$, then π is said to be better then or equal to the policy π'. So, value functions define a partial ordering over policies.

Definition 2.10 (Optimal Policy)**.** A policy π is said to be an *optimal policy* (π_*) if its expected return is greater than or equal to that of any other policy π' for all the states $s \in \mathcal{S}$.

$$\pi \geq \pi' \text{ iif } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S},\ \pi' \in \Pi\ .$$

Where Π is the set of all the set of all possible policies.

There could be more optimal policies that can be all denoted by π_* since they share the same state-value function, called *optimal state-value function* (v_*) and the *optimal action-value function* (q_*).

Definition 2.11 (Optimal State-Value Function)**.** Given the optimal policy π_*, the *optimal state-value function* v_* is defined as follows:

$$v_*(s) \doteq \max_{\pi} v_\pi(s) \text{ for all } s \in \mathcal{S} .$$

Definition 2.12 (Optimal Action-Value Function)**.** Given the optimal policy π_*, the *optimal action-value function* q_* is defined as follows:

$$q_*(s,a) \doteq \max_{\pi} q_\pi(s,a) \text{ for all } s \in \mathcal{S},\, a \in \mathcal{A} .$$

The optimal state-value (and the action-value) function must satisfy the self-consistency condition given by the BE: since we are referring to the optimal state-value function, the consistency of v_* can be written in a special form without reference to a specific policy.

Definition 2.13 (Bellman optimality equation)**.** Given an optimal policy π_* and its associated optimal state-value function v_*, the *Bellman optimality equation* (BOE) for v_* is defined as follows:

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \\
&= \max_a \sum_{s',r} p\left(s', r | s, a\right)\left[r + \gamma v_*(s')\right] .
\end{aligned}$$

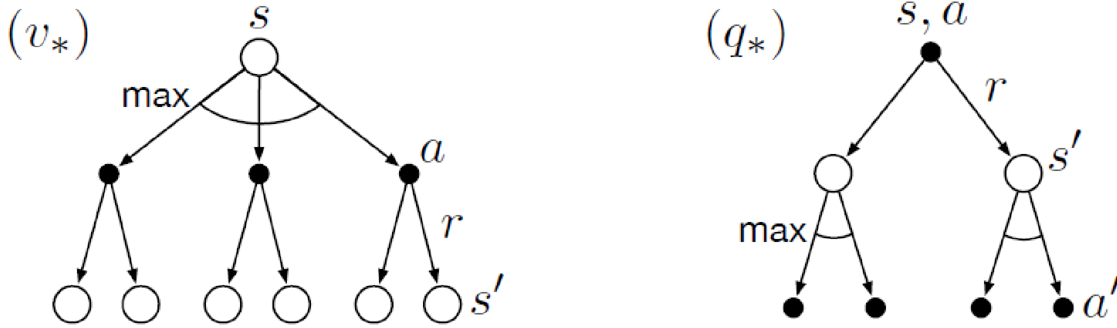


Figure 2.4 Diagrams of the optimal state-value function (left) and the optimal action-value function (right). On the left, the optimal state value is the maximum among the expected returns obtained from all the possible actions. On the right, the optimal action value is given by the weighted sum of the maximum expected return obtained from the next state pair (s',a'); the weights are given by the probabilities of occurring in state s' and selecting the action a', starting from (s,a).

As shown in Figure 2.4, the BOE for v_* expresses the fact that the value of a state must be equal to the expected return obtained with the best action from that state. On the other hand, the BOE for q_* can be defined in the following way:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a')|S_t = s, A_t = a\right] \\
&= \sum_{s',r} p\left(s',r|s,a\right)\left[r + \gamma \max_{a'} q_*(s',a')\right].
\end{aligned}
$$

If we know the optimal value function, then it is straightforward to obtain the optimal policy: for the state-value function v_*, the actions that appear best after a one-step search will be optimal actions. This means that any policy that acts *greedy* according to an optimal state-value function, then it is an optimal policy. Moreover, a policy that greedily follows the optimal action-value function q_* is an optimal policy too: for the action-value function, the choice of actions is even easier, indeed, with q_* we do not need to perform a one-step-ahead search, since we just need to find the action a which maximize the expected return $q_*(s,a)$ for all states $s \in \mathcal{S}$; it provides the expected return as a value that is locally and immediately available for each state-action pair. In addition, the action-value function allows the agent to

select optimal actions without knowing anything about the dynamics of the environment, so without knowing anything about any possible successor state.

It would be nice to be able to easily and quickly derive the optimal state-value or the optimal action-value functions, however in practice, it is not so straightforward. Let us take the state-value function v_π: for finite MDPs, it has a unique solution independent of the policy; in particular, the BOE is a system of n equations (one for each state, let us suppose n states) in n unknowns. If the dynamics of the environment ($p(s',r|s,a)$) are known, then it is possible to solve the BOE for v_* or q_*. If we could find a solution for the BOE, then we would solve the RL problem, but this solution is rarely applicable in practice; there are three main reasons: *(i)* The dynamics of the environment should be well-known; *(ii)* we should have enough computational resources to compute the solution of the BOE; and, *(iii)* the Markov property should be satisfied. For example, the game of backgammon satisfies the first and the last point aforementioned, but we do not have enough resources to compute the solution for the almost 10^{20} states of the game.

Given the difficulty in finding an exact solution to the BOE, we need to find other methods to approximate the optimal solution. When there are a limited number of states (or state-action pairs), we could use *tabular* methods, that use arrays or tables with one entry for each state (or state-action pairs). The idea is to iteratively update the value of each entry every time a state (or state-action pair) is encountered; when the number of times a state (or state-action pair) is encountered approaches infinity, then the value in its corresponding entry in the array (or table) tends to the value of the optimal state-value (or action-value) function. Instead, when the number of states is much greater, we need more compact parameterized functions to approximate the value functions: in our case, we will use *Deep Neural Network* (DNN) to approximate them.

Another aspect to take into account, especially in complex environments with a huge number of states, is that states have different probabilities of being encountered by the agent: there are states where the agent is more likely to visit and there are others where it is less probable. We prefer an approximation value function that is able to estimate better the values of states in which the probability to be visited is higher than states in which this probability is

lower. This is because the less probable states will have a low impact on the amount of reward the agent receives since they will be encountered few times or never. The online nature of RL facilitates the learning of optimal policies so as to put more effort into making good decisions for frequently encountered states, at the expense of less effort for less frequently encountered states. This aspect can easily be observed in tabular methods: the more a state is visited, the more accurate the value of that state is.

2.5 On-policy vs Off-policy

As mentioned above, most of the time it is not possible to obtain the exact solution of the BOE, so we need to find alternative methods that allow us to find the optimal policy. Given the online nature of RL, it is possible to alternate between collecting new data, i.e., exploring new states and proving other actions (interaction with the environment), and optimizing the policy.

In other words, we need to balance the *exploration* of new actions and the *exploitation* of the actions we consider better. A simple method to achieve an exploration-exploitation trade-off consists of using an ε-greedy policy.

Definition 2.14 (ε-greedy Policy)**.** Given an action-value function $q(s,a)$, an $\varepsilon \in (0,1)$, and m is the number of actions that the agent can select, then the ε-greedy policy π is defined as follows:

$$\pi(a|s) \doteq \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & \text{if } a^* = \arg\max_{a \in \mathcal{A}} q(s,a) \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases}$$

From the definition, it is possible to notice that there are no actions with zero probability to be selected. This ensures that the policy converges toward the near-optimal policy when the played steps approach infinity.

Theorem 2.1 (ε-greedy Policy Improvement)**.** For any ε-greedy policy π, the ε-greedy policy π' with respect to q_π is an improvement, $v_{\pi'} \geq v_\pi$.

$$
\begin{aligned}
q_\pi\left(s,\pi'(s)\right) &= \sum_{a\in\mathcal{A}} \pi'(a|s)q_\pi(s,a) \\
&= \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\varepsilon)\max_{a\in\mathcal{A}} q_\pi(s,a) \\
&\geq \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\varepsilon)\sum_{a\in\mathcal{A}} \frac{\pi(a|s)-\varepsilon/m}{1-\varepsilon} q_\pi(s,a) \\
&= \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(s,a) = v_\pi(s)
\end{aligned}
$$

Since the policy π' is the ε-greedy policy with respect to q_π. This means that it selects the action that maximizes the value (i.e., $a^* = \arg\max_{a\in\mathcal{A}} q(s,a)$) with probability $\frac{\varepsilon}{m} + 1 - \varepsilon$ and one of the other actions with probability $\frac{\varepsilon}{m}$.

Another aspect that must be taken into account is which policy is used for exploring and collecting the experiences and which one is used for playing greedily to maximize the reward, i.e., the policy to be learned. The policy used to generate behavior and collect experiences is called *behavior policy*, instead, the one to be learned is called *target policy*.

The easiest way is to consider the behavior and the target policies the same, in this way, the policy is used to collect experiences and then, on collected data, the policy and the value functions are updated. This process continues iteratively until convergence. This method is called *on-policy learning* because the two policies coincide, so the policy that gives behavior is updated during training.

Another strategy is to keep the target and behavior policies separate, in this way we can evaluate the policy target $\pi(a|s)$ to compute values $v_\pi(s)$ and $q_\pi(s,a)$ while following the behavior policy $\mu(a|s)$ [52]. This strategy (called *off-policy learning*) allows for several advantages over on-policies methods:

- We can use experiences collected from humans or other agents to train our agents.

- We can reuse the collected experiences, i.e., the ones generated from old policies $[\pi_0, \pi_1, \pi_2, \ldots, \pi_{t-1}]$, where t indicates the number of policy updates.

- We can learn the optimal policy while following a more exploratory policy, making sure we have tried all possible actions.
- Learn about multiple policies while following one single policy.

As one can imagine, off-policy learning is a little more complicated than on-policy learning. This is the reason why the off-policy approach generally has more variance and slower convergence than the on-policy one. The first requirement we have in off-policy learning is the assumption of *coverage*, referring to the fact that each action taken under the target policy π must be also taken, at least occasionally, under the behavior policy μ. This means that $\pi(a|s) > 0$ implies $\mu(a|s) > 0$, in order to be able to estimate values for π from μ. In this way, it is possible to make π a deterministic optimal policy, while μ remains stochastic and more exploratory, e.g., an ε-greedy policy.

To estimate the values for a target policy π from a behavior one μ, the off-policy methods use the *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. In particular, it is applied to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under π and μ policies, called the *importance sampling ratio* [55].

Definition 2.15 (Importance Sampling Ratio)**.** Given a starting state S_t, the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under any policy π is:

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T | S_t, A_{t:T-1} \sim \pi\} \\
&\quad = \pi(A_t|S_t) p(S_{t+1}|S_t, A_t) \pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=1}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)
\end{aligned}$$

where p is the state-transition probability function.

Thus the *importance sampling ratio* (i.e., the relative probability of the trajectory under the target and behavior policies) is defined as follows:

$$\rho_{t:T-1} \doteq \frac{\prod_{k=1}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k,A_k)}{\prod_{k=1}^{T-1} \mu(A_k|S_k)p(S_{k+1}|S_k,A_k)} = \prod_{k=1}^{T-1} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)} .$$

It can be seen that the dynamics of the environment defined by the MDP are canceled since they are identical at the numerator and denominator, so the importance sampling ratio depends only on the two policies, not on the MDP. In particular, the importance sampling ratio is used to "convert" the expected returns estimated with μ to the expected values for π:

$$\mathbb{E}\left[\rho_{t:T-1} G_t | S_t\right] = v_\pi(s) .$$

Where the expected returns G_t are obtained with the behavior policy μ.

2.6 Value Function Approximation

As anticipated, many problems have too many states in the MDP to be able to calculate the value exactly from the state-value functions or derive it by tabular methods, there would be too many states and/or actions to memorize, and learning would be too slow since we would have to visit all the states (or state-action pairs) a number of times tending to infinity. For instance, the game of backgammon has over 10^{20} states, whereas the "Computer Go" has about 10^{170} states.

In these cases, it is better to estimate values with *function approximation*:

$$\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$$

and/or

$$\hat{q}(s,a,\mathbf{w}) \approx q_\pi(s,a) .$$

Where $\mathbf{w}$ are the parameters used to approximate the state-value (or the action-value) function. The use of fewer parameters (than the number of states/state-action pairs) aims to be able to generalize from seen states to unseen states. There are three main types of value function approximations [52]:

- $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$, given a state s in input, the function approximation estimates the state-value function, so it outputs $\hat{v}(s,\mathbf{w})$.

- $\hat{q}(s,a,\mathbf{w}) \approx q_\pi(s,a)$., given a state s and an action a, the function approximation estimates the action-value function, so it outputs $\hat{q}(s,a,\mathbf{w})$.

- Given in input a state s, it estimates the action-value function for each possible action from the state s. So the output will be: $[\hat{q}(s,a_0,\mathbf{w}),\hat{q}(s,a_1,\mathbf{w}),\dots,\hat{q}(s,a_{m-1},\mathbf{w})]$. Where m is the number of actions.

One possibility is to use NN or DNN to approximate the state-value and action-value functions, so we can learn the parameters by Stochastic Gradient Descent. For instance, the goal could be to find the parameters $\mathbf{w}$ that minimizes the mean-squared error between the approximate value function $\hat{v}(s,\mathbf{w})$ and the true value function $v_\pi(s)$:

$$J(\mathbf{w}) = \mathbb{E}_{s\in\mathcal{S}}\left[(v_\pi(s) - \hat{v}(s,\mathbf{w}))^2\right] .$$

Since in RL, there is not any supervisor that gives us the true value function v_π, then we need to find a way to substitute it with a target for v_π because we do not have the real value function. One possibility is to substitute it with the return G_t, for instance, if we are using a MC algorithm [55]. Indeed, the main idea of MC methods is to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value.

Another possibility is to use the TD targets. TD learning is an ensemble of techniques that enables one to learn from incomplete episodes, by *bootstrapping*. It learns directly from experiences and does not need to know the MDP of the environment. Differently from the MC methods, the TD ones uses bootstrapped targets:

- $\text{TD}(0) \doteq G_{t:t+1} \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$. It is the simplest TD target (also known as *one-step* TD target), where it bootstraps one step ahead to estimate the expected return.

- $\text{TD}(n) \doteq G_{t:t+n+1} \doteq R_{t+1} + \gamma R_{t+2} + \cdots \gamma^n R_{t+n+1} + \gamma^{n+1}\hat{v}(S_{t+n+1},\mathbf{w})$; where n is the number of bootstrapping steps.

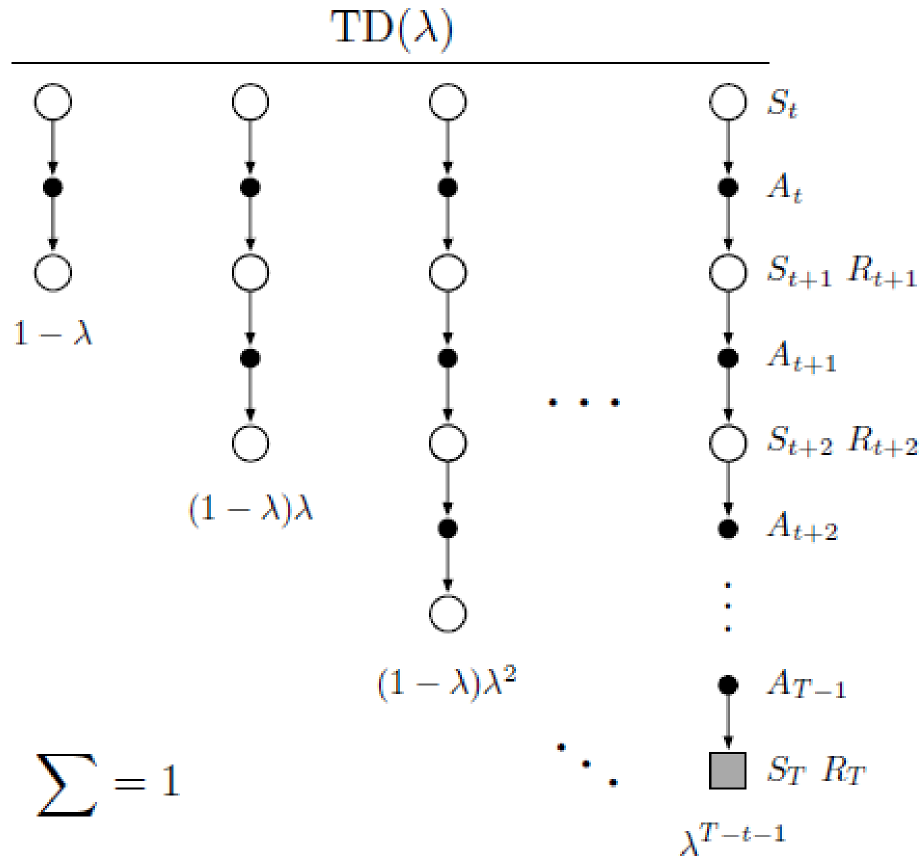


Figure 2.5 How TD(n) targets are weighted in TD(λ)-returns: if $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the *Monte Carlo* target [55].

- TD(λ) $\doteq (1-\lambda)\sum_{n=1}^{+\infty} \lambda^{n-1} G_{t:n}$, where $G_{t:n} =$ TD($n-1$) target. In Fiugre 2.5 and Figure 2.6 it is possible to graphically understand how the TD(λ) targets are defined.

2.7 Policy Gradients and Actor-Critic Methods

In the previous section, some examples of value function approximations were shown, in those cases the policy was not estimated but was rather derived from the state-value and action-value functions. Methods that estimate and learn optimal state-value and/or action-value functions without learning policy are called *action-value methods*.

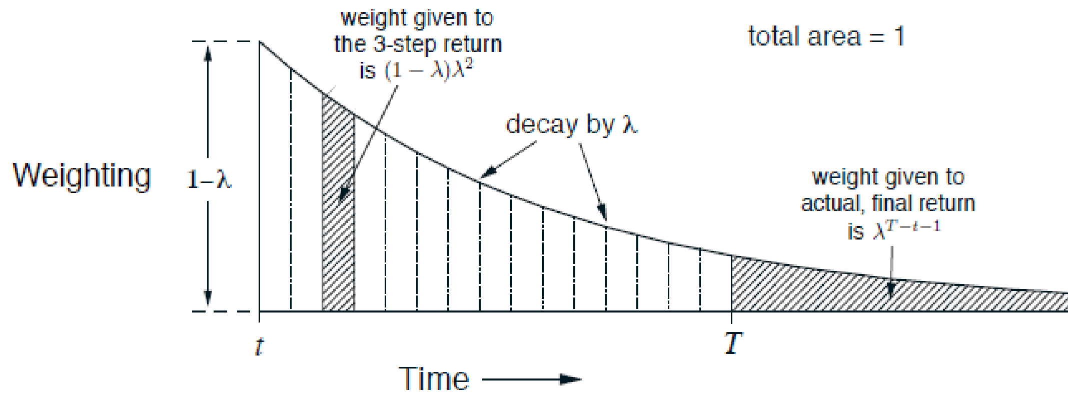


Figure 2.6 Weighting given in the λ-return to each of the n-step returns [55].

There are, however, two other kinds of methods for solving RL problems: *(i) Policy Gradient methods* that learn a parameterized policy and *(ii) Actor-Critic methods* that learn approximations to both policy and state-value and/or action-value functions.

2.7.1 Policy Gradient methods

As anticipated, the policy gradient methods learn a parameterized policy by little incremental changes in the policy, which leads to better convergence properties. Another advantage of these methods with respect to the action-value ones is that policy gradient methods can learn stochastic policies, but they typically converge to a local minimum rather than to a global optimum; moreover, in general, policy gradient methods have a greater variance than action-value ones.

As one can imagine, the goal is to find the best parameters θ for the parameterized policy $\pi_\theta(a|s)$. Now, it is necessary to define a way to establish how good a parameterized policy is (i.e., the objective function) [52]:

- In episodic environments it is possible to use the *start value*:

$$J_1(\theta) \doteq V^{\pi_\theta}(s_0) \doteq \mathbb{E}_{\pi_\theta}[v(s_0)]$$

where s_0 is the starting state (or it is a distribution over the possible starting states). The quality is given by the expected return the agent will get from the starting state.

- In continuing environments it is possible to use the *average value*:

$$J_{\text{avV}}(\theta) \doteq \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

where $d^{\pi_\theta}(s)$ is the probability to end up in the state s. In this case, there is not a start state, we consider the probability to be in a specific state s and we multiply this probability with the value of the state s onwards.

- An alternative is to consider the *average reward per time step*:

$$J_{\text{avR}}(\theta) \doteq \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} \pi_\theta(a|s)\mathcal{R}_s^a$$

where $\mathcal{R}_s^a$ is the immediate return received from the environment.

Definition 2.16 (Score Function)**.** Let us assume that π_θ is differentiable, and we know its gradient $\nabla_\theta \pi_\theta(a|s)$, then, the *Likelihood Ratios* exploit the following identity:

$$\begin{aligned}\nabla_\theta \pi_\theta(a|s) &= \pi_\theta(a|s)\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\ &= \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s)\ .\end{aligned}$$

The *Score Function* is $\nabla_\theta \log \pi_\theta(a|s)$.

Theorem 2.2 (Policy Gradient Theorem)**.** For any differentiable policy $\pi_\theta(a|s)$, and for any of the policy objectives functions ($J_1(\theta)$, $J_{\text{avV}}(\theta)$ or $J_{\text{avR}}(\theta)$). The policy gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) q_{\pi_\theta}(s,a)\right]\ .$$

2.7.2 Actor-Critic Methods

One of the shortcomings of gradient policy methods is that they have a high variance, so the two methods (action-value and policy gradient) are combined. A *critic* is used to estimate the action-value function $\hat{q}_(s,a,\mathbf{w}) \approx q_\pi(s,a)$. Therefore, there are two sets of parameters: *(i)* $\mathbf{w}$ for the critic, i.e., to update the action-value function; and *(ii)* θ for the actor, i.e., to learn the policy.

In other words, actor-critic methods follow an approximate policy gradient:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\hat{q}(s,a,\mathbf{w})\right]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(a|s)\hat{q}(s,a,\mathbf{w})\ .$$

Where α is the learning rate. In particular, at each step, we move a little bit, using stochastic gradient ascent, in the direction of the score multiplied by a sample from our approximated action-value function, meaning that the critic informs where we should move.

Another problem is that a bias is introduced when approximating the policy, but it is possible to reduce or eliminate this bias if we choose the value function approximation carefully, meaning that we can still follow the exact policy gradient and find the optimal solution.

Theorem 2.3 (Compatible Function Approximation Theorem)**.** If the following two conditions are satisfied:

1. The value function *approximator* is compatible with the policy.

$$\nabla_{\mathbf{w}}\hat{q}(s,a,\mathbf{w}) = \nabla_\theta \log \pi_\theta(a|s)\ .$$

2. The value function parameters $\mathbf{w}$ minimize the the mean-squared error:

$$\varepsilon = \mathbb{E}_{\pi_\theta}\left[\left(q_{\pi_\theta}(s,a) - \hat{q}(s,a,\mathbf{w})\right)^2\right]\ .$$

Then the policy gradient is exact:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\hat{q}(s,a,\mathbf{w})\right] .$$

Finally, it is possible to further improve the performance of actor-critic methods by reducing the variance with a baseline function $b(s)$, and at the same time without changing the expectation.

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)b(s)\right] &= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi_\theta(a|s)b(s) \\
&= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s)b(s) \sum_{a\in\nabla_\theta\mathcal{A}(s)} \pi_\theta(a|s) \\
&= 0 .
\end{aligned}$$

In particular, it turns out that a good baseline function is the state value function $b(s) = v_{\pi_\theta}(s)$.

Definition 2.17 (Advantage Function)**.** The *advantage function* is defined as follows:

$$A^{\pi_\theta}(s,a) \doteq q_{\pi_\theta}(s,a) - v_{\pi_\theta}(s)$$

So it is possible to rewrite the policy gradient using the advantages:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)A^{\pi_\theta}(s,a)\right] .$$

Once again, advantages can be estimated through two sets of parameters, one for the state-value function and the other for the action-value function. Alternatively, it is possible to use only one parameter set for the critic, since the TD error ($\delta_t \doteq \mathrm{TD}(0) - \hat{v}(S_t,\mathbf{w}) \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$ is an unbiased estimator of the advantage function. Therefore, it is possible to use the TD error to compute the policy gradient.

CHAPTER 3

State of the Art

This chapter analyzes and describes the main RL algorithms used in video games. As we have seen in Chapter 2, there are different types of algorithms: *action-value*, *policy gradients*, and *actor-critic*. It is possible to make another distinction between algorithms: *model-free* and *model-based*. The former refers to techniques that know nothing about the MDP of the environment; the latter, on the other hand, tries to learn the dynamics of the environment, thus trying to learn how to estimate the MDP of the model. Table 3.1 summarizes the approaches described in the following sections.

3.1 Model-free Algorithms

Let us start with the model-free algorithms: the first one is the DQN [39] that learns policies directly from high-dimensional inputs. It receives raw pixels and outputs a value function to estimate future rewards. In particular, they use an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual actions for the input state. Moreover, it uses the experience replay method to break the sample correlation, making the network stable because the trajectories become uncorrelated, in addition, it randomizes the order of the elements in the dataset (i.e., no temporal correlation anymore). Another characteristic is that DQN uses fixed Q-targets, so, there are two versions of the

DNN, one *online* (that is used to select actions) and one is the *target* one, that is a periodic copy of the online network which is not directly optimized.

DQN was later improved, in particular, there are different algorithms that started from the idea of DQN: *(i)* Double DQN [61] that introduces double Q-learning to reduce observed over-estimations bias. Instead, *(ii)* prioritized experience replay [47] helps prioritize experience to replay important transitions more frequently, so it improves data efficiency, and it sees transitions from which there is more to learn. *(iii)* Dueling DQN [63] includes two separate estimators: one for the state-value function and the other for the advantage function; this helps to generalize across actions. *(iv)* Noisy DQN [18] uses stochastic network layers for exploration, in particular, it adds Factorised Gaussian noise, which uses an independent noise per each output and another independent noise per each input. As one can derive, all the DQN algorithms are action-value methods.

Another model-free action-value algorithm is Distributional Q-learning [4] that learns a categorical distribution instead of the mean of the discounted returns, in particular, the authors found out that it is better to learn an approximate distribution rather than its approximate expectation. The distributional Bellman operator preserves multi-modality in value distributions, which they believe leads to more stable learning.

In [38] a policy gradient method is proposed. It is A3C, and it maintains a parameterized policy and an estimate of the value function: it uses a mix of n-step returns to update both of them, this enables to shift of the bias-variance trade-off and helps to propagate newly observed rewards faster to earlier visited states. The critics in A3C learn the value function while multiple actors are trained in parallel and get synchronized with global parameters every so often. Moreover, the gradients are accumulated as part of training for stability. There is also a synchronous and deterministic version of A3C, called A2C, that waits for each actor to finish its segment of experience before updating, averaging over all of the actors.

Rainbow [29] is an algorithm based on DQN that takes the strengths of six algorithms and combines them to form a state-of-the-art agent. The algorithms are: Dobule DQN [61], Prioritize DQN [47], Dueling DQN [63], A3C [38], Distributional Q-learning [4], and Noisy DQN [18]. They replace the 1-step distributional loss with a multi-step variant, then

the multi-step distributional loss is combined with double Q-learning by using the greedy action selected by the online network and evaluating it with the target network. Instead of prioritizing replays with the absolute TD error, they found that it was more effective to prioritize the transitions by the *Kullback-Leibler* (KL) loss since it is what the algorithm is minimizing; in particular, it seems to be more robust to noisy stochastic environments because the loss can continue to decrease even when the returns are not deterministic.

Another model-free algorithm that has been widely used in recent years is PPO [50]. To optimize policies, PPO alternates between sampling data from the policy and performing several epochs of optimization on the sampled data. It is inspired by TRPO [49], but it makes some improvements, for instance, it tries to attain the data efficiency and reliable performance of TRPO, while using only first-order optimization, and it proposes novel objective with clipped probability ratios, which forms a pessimistic estimate of the performance of the policy. In particular, it removes the constraint of the objective function of TRPO and turns it into a penalty. Moreover, it penalizes changes to the policy that move the ratio between the new policy and the old one away from one. PPO still is one of the benchmarks in RL, indeed, in [32] the authors present a solution based on PPO, it is an autonomous system that can race physical vehicles at the level of the human world champions. The system combines *Deep Reinforcement Learning* (DRL) in simulation with data collected in the physical world.

Despite being a model-free algorithm, PPO can also be used to complete tasks in which it is necessary to discover achievements that have a hierarchical structure. This requires agents to possess a broad range of abilities, including generalization and long-term reasoning. In [40], the authors present a method based on PPO that is able to outperform the SOTA model-based algorithms. In addition to training the model with the PPO algorithm, they train the model to predict the next achievement; moreover, they use a Contrastive loss [60] to maximize the similarity in the latent space of the state-action pairs and the corresponding next achievement. Finally, they leverage the fact that all episodes share the same achievement structure and maximize the similarity in the latent space between achievements from two different episodes that are matched via optimal transport [6].

In general on-policy algorithms, such as TRPO, PPO, or A3C, require new samples to be collected for each gradient step. This quickly becomes extravagantly expensive, as the number of gradient steps and samples per step needed to learn an effective policy increases with task complexity, instead, off-policy methods aim to reuse past experience. SAC [21] is an off-policy algorithm for continuous state and action spaces. It uses an entropy regularization in its objective function, in particular, it trains the policy to maximize a trade-off between entropy and expected return. This can be seen as the trade-off between exploration and exploitation: the greater the entropy, the more the exploration, (in general) the faster the learning process. Moreover, a high entropy can also prevent the learning policy from converging to a poor local optimum.

The last two model-free algorithms to be presented are DroQ [30] and its predecessor REDQ [10]. REDQ is a sample-efficient model-free method for solving maximum-entropy RL problems: it uses a high *Update-to-Data* (UTD) ratio, i.e., the number of updates taken by the agent compared to the number of actual interactions with the environment. A high UTD ratio increases the overestimation bias in the Q-function training. So, to reduce the overestimation bias, it uses an ensemble of n Q-functions for the target to be minimized.

DroQ introduces the dropout Q-function which is a Q-function equipped with dropout and layer normalization, moreover, it uses a small ensemble of dropout Q-functions (more dropout Q-functions provide better performance than using a single dropout Q-function). In particular, the number of dropout Q-functions is much smaller than the number of Q-functions in REDQ: this reduction makes DroQ more computationally efficient.

3.2 Model-based Algorithms

Model-based algorithms have begun to spread in recent years, partly due to the increased computing power that has grown exponentially in the last period.

TreeQN [17] dynamically constructs a tree by recursively applying a transition model in a learned abstract state space and then aggregating predicted rewards and state values using a tree backup to estimate Q-values. ATreeC [17] is an actor-critic variant that augments

TreeQN with a softmax layer to form a stochastic policy network. Both approaches are trained end-to-end, such that the learned model is optimized for its actual use in the planner.

Another model-based algorithm is presented by Vezhnevets et al. [62]: STRAW. It uses a NN to build implicit plans, in particular, it can learn high-level, temporally abstracted macro-actions of varying lengths that are solely learnt from data without any prior information. It can also partition this internal representation into contiguous sub-sequences by learning for how long the plan can be committed to.

Another solution is proposed by Ha et al. in [20], in which a generative recurrent neural network is quickly trained in an unsupervised manner to model popular reinforcement learning environments through compressed spatiotemporal representations. The extracted features are used to feed simple and compact features trained by evolution. The authors also train the agent entirely inside of an environment generated by its own internal world model and transfer this policy back into the actual environment.

This work inspired the authors of PlaNet [25] a purely model-based agent that learns the environment dynamics from images and chooses actions through fast online planning in latent space. In this work, they introduced a latent dynamic model with both deterministic and stochastic transition components. PlaNet encapsulates the basic ideas of the *Recurrent State-Space Model* (RSSM) [26], which will be described in Chapter 4.

VProp [42] proposes a set of parameter-efficient differentiable planning modules built on Value Iteration that can successfully be trained using reinforcement learning to solve unseen tasks, has the capability to generalize to larger map sizes, and can learn to navigate in dynamic environments.

The last two model-based algorithms are AlphaZero [53] and MuZero [48]. AlphaZero is inspired by AlphaGo Zero [54] one of the first algorithms to achieve superhuman performance in the game of Go, by *tabula rasa* reinforcement learning from games of self-play. AlphaGo Zero represents Go knowledge using Deep Convolutional NN. In particular, AlphaZero tries to generalize AlphaGo Zero in order to achieve, *tabula rasa*, superhuman performance in many challenging domains.

Algorithm	On-policy vs Off-policy	Category
DQN [39]	Off-policy	Model-free, Value-based
Double DQN [61]	Off-policy	Model-free, Value-based
Dueling DQN [63]	Off-policy	Model-free, Value-based
Noisy DQN [18]	Off-policy	Model-free, Value-based
Distributional Q-learning [4]	Off-policy	Model-free, Value-based
A2C [38]	On-policy	Model-free, Policy Gradient
A3C [38]	On-policy	Model-free, Policy Gradient
Rainbow [29]	Off-policy	Model-free, Value-based
TRPO [49]	On-policy	Model-free, Policy Gradient
PPO [?]	On-policy	Model-free, Policy Gradient
SAC [?]	Off-policy	Model-free, Policy Gradient
REDQ [10]	Off-policy	Model-free, Policy Gradient
DroQ [30]	Off-policy	Model-free, Policy Gradient
TreeQN [17]	On-policy	Model-based
ATreeC [17]	On-policy	Model-based
STRAW [62]	On-policy	Model-based
World Model [20]	On-policy	Model-based
PlaNet [25]	Off-policy	Model-based
VProp [42]	Off-policy	Model-based
AlphaZero [53]	Off-policy	Model-based
MuZero [58]	Off-policy	Model-based

Table 3.1 A survey of the SOTA algorithms in RL.

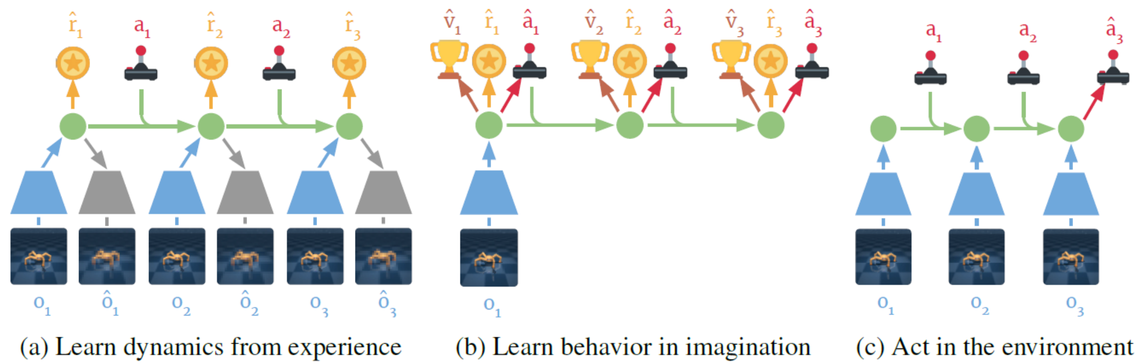
MuZero combines Monte-Carlo Tree Search with a learned model and predicts the reward, the action-selection policy, and the value function to make planning. It extends model-based RL to a range of logically complex and visually complex domains and achieves superhuman performance. The MuZero model is composed of three connected components for representation, dynamics, and prediction. Instead, the Monte-Carlo Tree Search is performed at each time step t to select an action for environment interaction. Finally, the model is trained by sampling a trajectory, then the representation function h encodes the observations, and, subsequently, the model is unrolled recurrently for K steps. The parameters of the representation, dynamics, and prediction functions are jointly trained, end-to-end by backpropagation-through-time to predict three quantities: the policy, the value function, and the reward.

CHAPTER 4

Dreamer

With the exponential growth of deep learning and computational power, RL algorithms have evolved more and more, and model-based algorithms have become increasingly popular in recent years. A model-based algorithm uses a predictive model to imagine consequences among all the possible actions, and then properly selects the best option. Dreamer is a model-based approach, capable of learning long-horizon behaviors from images purely by latent imagination, meaning that it learns an embedded representation of the real environment and uses this embedded representation to learn the optimal policy. Moreover, it is an off-policy algorithm, so it learns from previous experiences gathered with a different policy than the one the agent is trying to learn.

Since its publication, Dreamer has gained significant attention in the RL community because of the revolutionary idea behind it and its impressive performance. However, due to its complexity, there are few implementations and not so easy to understand, and none of them are in PyTorch. For this reason, we have meticulously studied Dreamer and its original implementation in Keras/Tensorflow, to provide a reliable, efficient, and easy-to-understand version in PyTorch.



(a) Learn dynamics from experience (b) Learn behavior in imagination (c) Act in the environment

Figure 4.1 The structure of Dreamer-V1: on the left is the "dynamic learning", in which the agent learns the dynamics of the environment. In the center is "behavior learning", in which the agent learns how to optimally behave and learns to correctly estimate state values through imagination. On the right is the "environment interaction", in which the agent collects new data for training.

4.1 Components

Let's start getting familiar with Dreamer: as mentioned before, it is an off-policy, model-base algorithm that learns a latent representation of the environment, i.e., it can abstract observations to predict rewards and values, and select actions. A great advantage of latent representation is that the latent states have a smaller memory footprint than predictions in image space, allowing Dreamer to imagine thousands of trajectories in parallel. Each latent state is made of two parts: a deterministic part, which embeds all the history of the episode; and a stochastic part, which embeds more information about the actual state. In other words, Dreamer learns to encode observations and actions from previous experiences into latent states and predicts rewards given by the environment. Moreover, it learns state values and the policy in the latent space, in this way, during the environment interaction phase, it can encode all the history of the episode in the current latent state, and, from it, it can predict the next action. An explanatory diagram of Dreamer's components is available in Figure 4.1.

As one can imagine, Dreamer is a complex agent that is made of various components: a world model, an actor, and a critic. The first one is responsible for learning the latent representation of the environment; the actor selects the actions from the latent state; whereas

the latter component predicts the state values. The world model is the most complex and it is composed of five parts:

1. An *encoder*, a fully convolutional NN, which encodes the observations in pixel form provided by the environment.
2. An RSSM [26] which generates the latent states, and it is made by three models:
   (a) The *recurrent model*: a linear layer followed by an ELU [13] activation function and a *Gated Recurrent Unit* (GRU) [11], that encodes the history of the episode and computes the deterministic part of the latent state.
   (b) The *representation model*: an *Multilayer Perceptron* (MLP), that computes the stochastic part of the latent state from the deterministic part of the latent state and the actual observations.
   (c) The *transition model*: an MLP, that predicts the stochastic part of the latent state, it is used to imagine trajectories in the latent dynamic.
3. An *observation model*, a linear layer followed by a convolutional NN composed of transposed convolutions, that reconstructs the original observation from the latent state.
4. A *reward model*, an MLP, that predicts reward for a given latent state.
5. A *continue model*, an MLP, that estimates the discount factor to apply to the rewards.

In comparison with the world model, the actor and the critic are very simple, they are two MLPs models, that are completely learned in the latent representation, so the actor selects actions, and the critic predicts state values by referring only to latent states.

4.2 Dreamer V1

Now that we have a general overview of how Dreamer works, we can dwell on all the details of this algorithm. First, it is necessary to shed light on the learning algorithm, it is divided into two parts: in the former, the agent learns the latent representation (dynamic learning),

whereas in the latter it learns the actor and the critic while the world model is frozen (behavior learning).

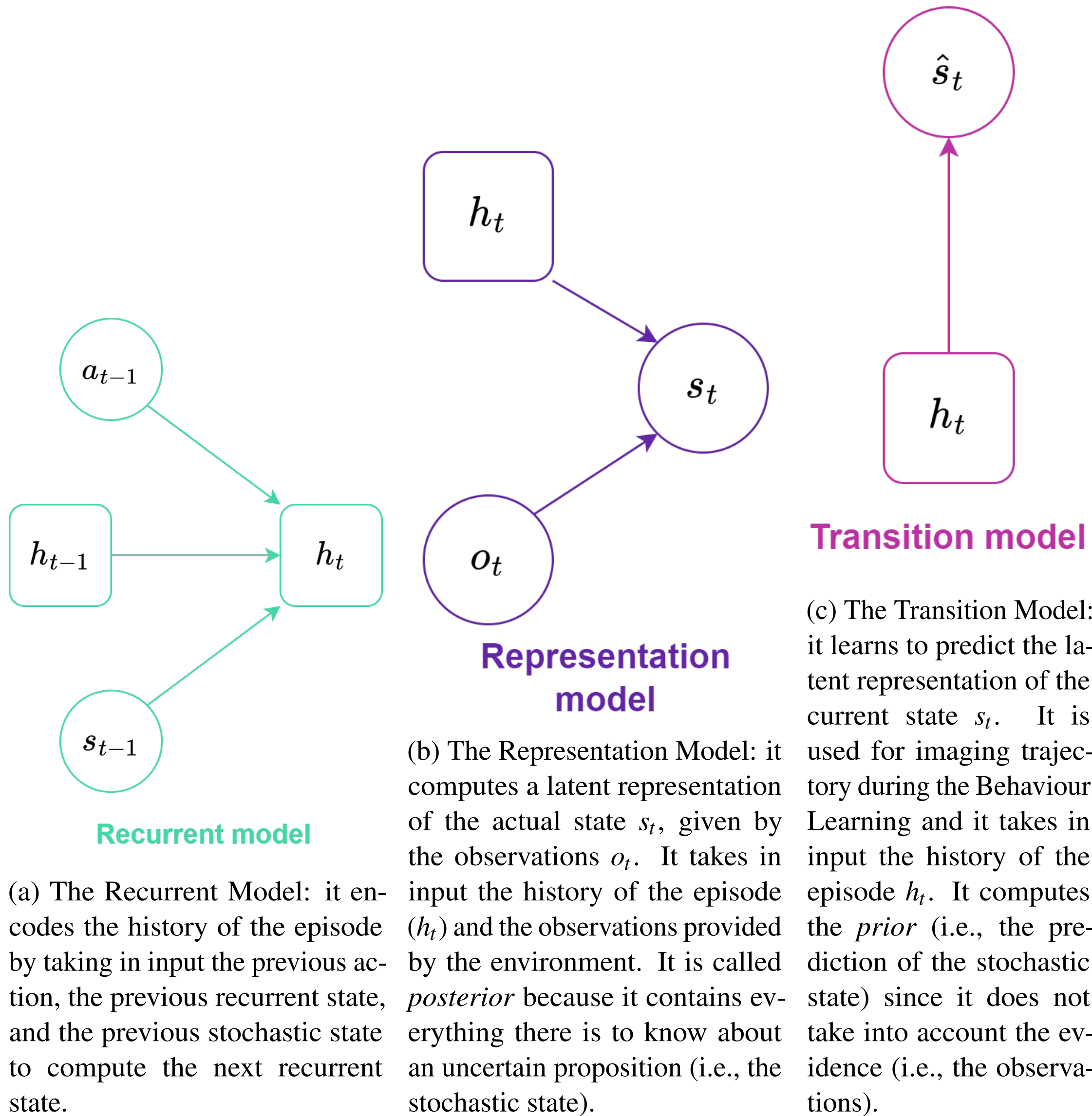
4.2.1 Dynamic Learning

In this phase the agent learns a latent representation of the environment from a batch of sequences, the whole world model is involved in this phase. The backbone of the world model is the RSSM, its goal is to embed the environment states into latent states. As previously anticipated, the latent state is composed of two parts: the deterministic part (called *recurrent state* from now on), which encodes all the history of the episode; and the so-called *stochastic state*, which contains more information about the current state.

The recurrent model is responsible for embedding the history of the sequence and it computes the recurrent state (h_t in Figure 4.2a) from the previous action, recurrent state, and stochastic state. It is implemented by a dense layer followed by a GRU: the dense layer takes in input the previous action and the previous stochastic state concatenated together, whereas the GRU takes in input the output of the dense layer and the recurrent state.

Both the representation and the transition models (in Figure 4.2b and Figure 4.2c, respectively) compute the distribution of the stochastic state (and then the stochastic state is sampled from them), the difference between them is that the former computes the actual stochastic state (posterior), whereas the latter predicts it (prior). So, the representation model uses the observations (embedded by the encoder) to compute the stochastic state, whereas the transition model does not use it. This means that the recurrent model is more precise than the transition model, but we need the second for the imagination of the trajectories, indeed, it is easier and less expensive to imagine possible future states in the latent space instead of in the image space. Do not worry if this concept is not very clear now, it will be better explained in the next section, now let's focus on the operation of the world model.

The states, computed by the RSSM, must be reliable and consistent with the environment, indeed, the agent should be able to predict rewards, values, the discounts of the values, and select actions from the latent states. For this reason, from the latent states, the reward and the continue models learn to predict rewards, and discounts (the probability that the episode



(a) The Recurrent Model: it encodes the history of the episode by taking in input the previous action, the previous recurrent state, and the previous stochastic state to compute the next recurrent state.

(b) The Representation Model: it computes a latent representation of the actual state s_t, given by the observations o_t. It takes in input the history of the episode (h_t) and the observations provided by the environment. It is called *posterior* because it contains everything there is to know about an uncertain proposition (i.e., the stochastic state).

(c) The Transition Model: it learns to predict the latent representation of the current state s_t. It is used for imaging trajectory during the Behaviour Learning and it takes in input the history of the episode h_t. It computes the *prior* (i.e., the prediction of the stochastic state) since it does not take into account the evidence (i.e., the observations).

has ended in that specific time step) respectively; moreover, the observation model tries to reconstruct the environment observations starting from the latent states.

As mentioned before, the agent learns a latent representation of the environment from sequences sampled from the buffer: it turned out that, for learning the dynamics of the environment, it is recommended to have 50 sequences of length 50; so the first two hyper-parameters are: the *Sequence Length* (SL) and the *Batch Size* (BS). We decide to shape the data by taking in the first dimension the index of the sequence, whereas the second dimension refers to the sequence in the batch $(\mathrm{SL}, \mathrm{BS}, *)$, where $*$ is the dimension of the data, for

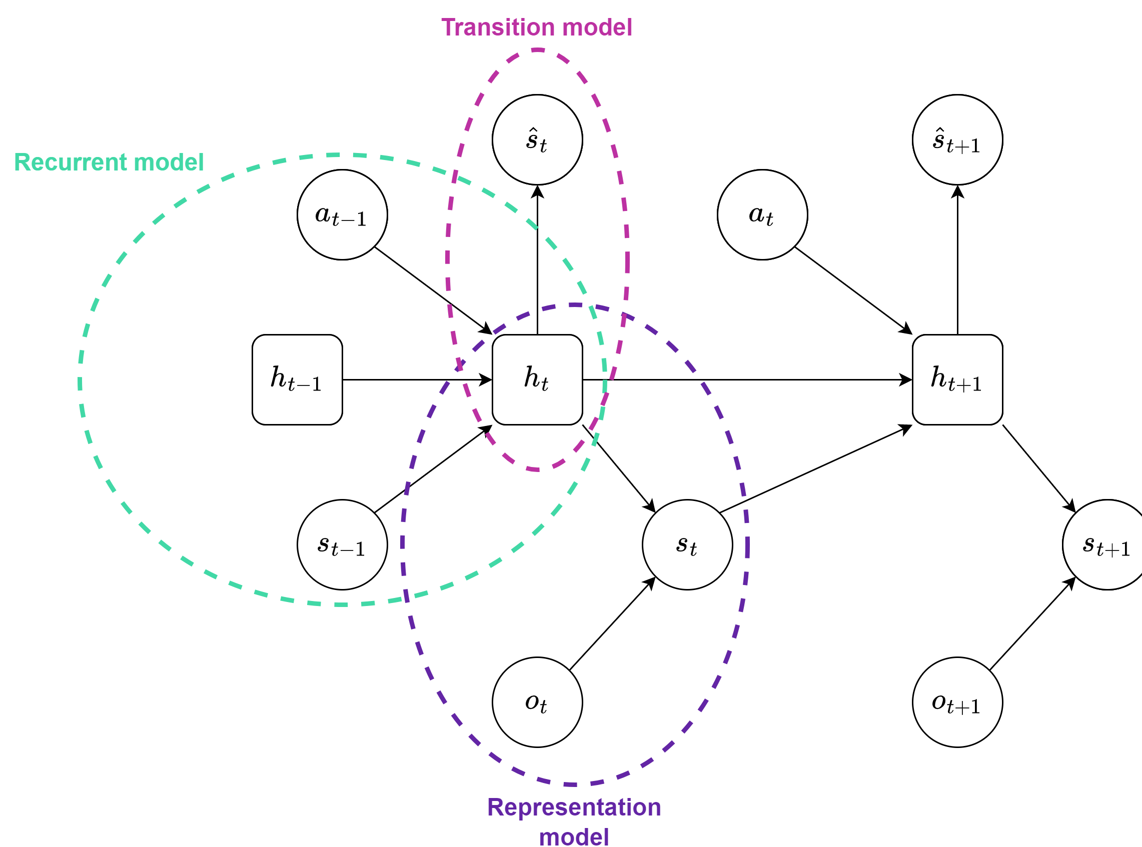
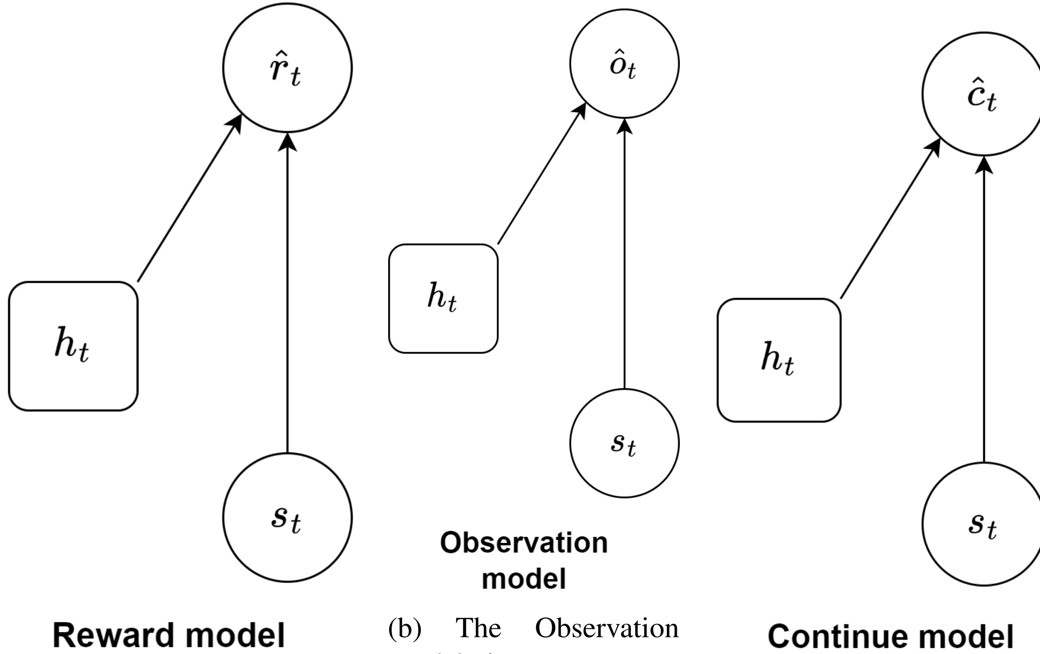


Figure 4.3 The RSSM is composed of the Recurrent Model, the Transition Model, and the Representation Model. These three models are iteratively used to learn the dynamics of the environment. First, the Recurrent Model computes the history of the episode, taking in input the previous history (i.e., the recurrent state h_{t-1}), the previous action a_{t-1} and the previous posterior s_{t-1}. Then, the transition model predicts the stochastic state (prior denoted with $\hat{s}_t$) from the new history computed by the recurrent model. Finally, the representation model computes the posterior state s_t from the history h_t and the observations o_t provided by the environment.



(a) The reward model takes in input the latent states (the posterior s_t concatenated with the recurrent state h_t and predicts the amount of reward received in that state.

(b) The Observation Model tries to reconstruct the observations from the latent states. In this way, the agent learns to map the observations provided by the environment in latent states.

(c) The Continue Model predicts the discount factor, i.e., the γ. The higher the predicted discount factor, the higher the probability the episode will continue for that time step.

instance, the observations will have dimension $(3,64,64)$ for RGB observations, so the observations during training will have shape equal to $(\mathrm{SL},\mathrm{BS},3,64,64)$, i.e., $(50,50,3,64,64)$. An essential consideration must be made for the observations, in fact, it is worthwhile to bring image observations in the range $[-0.5,0.5]$, to enable the observation model to more easily reconstruct them. Another important aspect that should not be overlooked is to properly initialize the states for the recurrent model, sure enough, the recurrent state and the stochastic state must be initialized with tensors of zeros.

Once all the tensors are accurately initialized and the observations are embedded by the encoder, it is possible to start with the core part of dynamic learning: the computation of all the latent states. After a careful analysis, we found that it is mandatory to perform a for loop on the sequence, due to the structure of the RSSM, indeed, the recurrent unit takes in input the previous action and the previous latent state, that is composed by the recurrent state and the posterior. The latter is computed from the recurrent state, so it is impossible to process

the entire sequence in one go. Both the representation and transition model produce in output the mean and the standard deviation of the distribution of stochastic state, that is a diagonal normal distribution.

Now that all the latent states in the batch are computed, we can proceed to learn to reconstruct the original observations, predict the rewards, and compute the probability that the episode continues from the latent states just computed. The outputs of the reward and observation models are the means of the normal distributions (with $\sigma = 1$) of the rewards and the observations respectively, whereas the outputs of the continue model are the *logits* of a Bernoulli distribution. Now it is possible to compare the distributions in output from the various models of the world model with the real information provided by the environment. The world model loss (reconstruction loss) is defined as follows:

$$\text{rec loss} \doteq \text{KL}(p\|q) - \ln q_o(o) - \ln q_r(r) + \ln q_c((1-d)\cdot\gamma)$$

where:

- p is the probability distribution of the posterior (computed by the representation model).
- q is the probability distribution of the prior (computed by the transition model).
- KL is the KL divergence between p and q.
- q_o is the probability distribution of the observations.
- q_r is the probability distribution of the reward.
- q_c is the probability distribution of the discount factor.
- o and r are the target observations and rewards, respectively, provided by the environment.
- $1-d$ is the target discount factor, where d indicates whether the episode is ended ($d = 1$) or not ($d = 0$).

The error between the predictions of both the reward and observation models is computed with the log prob of the computed probabilities with respect to the real rewards and observations respectively. The error between the posterior and the prior are, instead, computed through a KL divergence between their distributions, clipped when its value is below 3 *nats*. It is necessary to make a separate argument for the continue model loss, indeed, it predicts the probability that the episode continues or ends, so its targets are smoothed for better performances, i.e., the target probability is said to be zero where there is a done and gamma when there is not, where gamma is the gamma discount factor in Eq. 6 of [24].

4.2.2 Behaviour Learning

With the dynamic learning phase over, it is time to move on to learning the actor and critic. The goal is to leverage the learned world model to imagine ahead the consequences of the actions. In practice, the imagination starts from a "real" latent state (i.e., the posterior concatenated with the recurrent state) and terminates after a certain number of imagination steps (*horizon*) in which the agent iteratively selects an action (through the actor in Figure 4.5) from the current latent state and compute the next latent state by exploiting the world model.

As one can see in Figure 4.6, the recurrent and transition models of the RSSM are used to imagine trajectories. The transition model is used instead of the representation model because the observations provided by the environment are not available, so it is necessary to predict the states without them in a reliable way; moreover, the imagination in the latent space is much faster and cheaper than the imagination in the image space, so it is necessary to use the transition model.

All the latent states computed during the previous phase are used as starting points for completely imagined trajectories, for this reason, the latent states are reshaped in $(1, \mathrm{SL} \cdot \mathrm{BS}, \mathrm{latent_state_size})$, meaning that we are considering independently all the latent states computed before (one of sequence length and $\mathrm{SL} \cdot \mathrm{BS}$ as new batch size). Also for behaviour learning is needed a for loop because the trajectories are imagined one step at a time, i.e., the actor selects an action considering the last imagined state and then the new imagined latent state is computed.

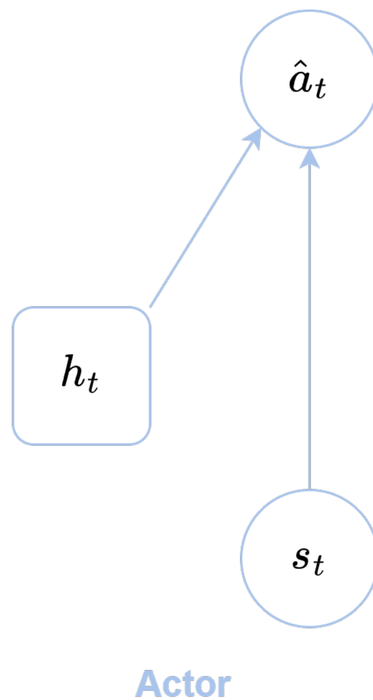


Figure 4.5 The Actor is responsible for selecting actions given latent states, indeed, it approximates the policy. Latent states are the only information it needs to choose actions. The stochastic state can be either the prior or the posterior. In the first case, the actor selects an action based on the imagined stochastic state; whereas, in the second case, it selects the action based on more precise information about the current state.

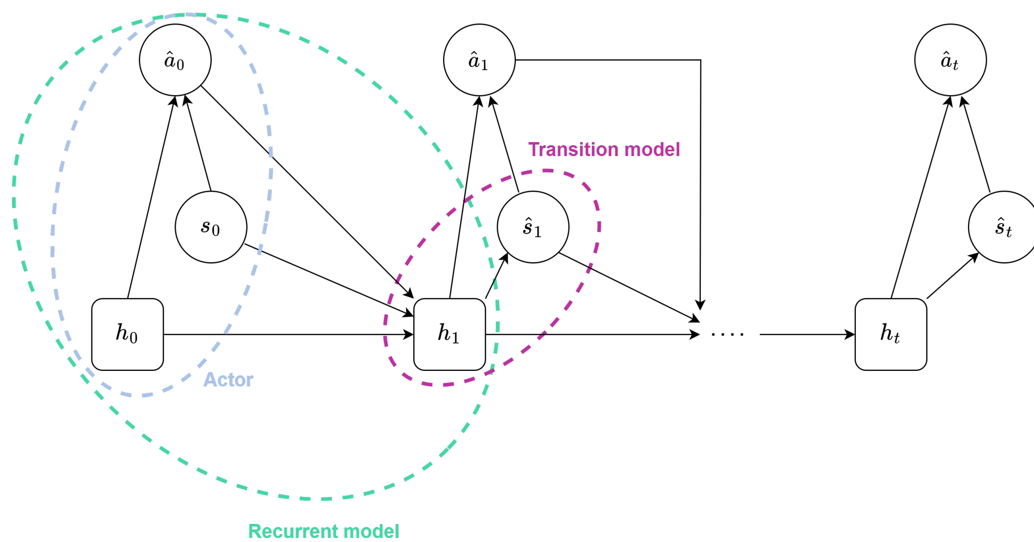


Figure 4.6 Imagination Phase: the agent starts from "real" latent states, i.e., posteriors concatenated with recurrent states (the history of the episode). It selects an action $\hat{a}_0$, then the RSSM computes the new recurrent state h_1 and the transition model computes the prior state $\hat{s}_1$. Then it iteratively performs these imagination steps up to a certain horizon H. The imagined trajectory will be used to learn the actor and the critic.

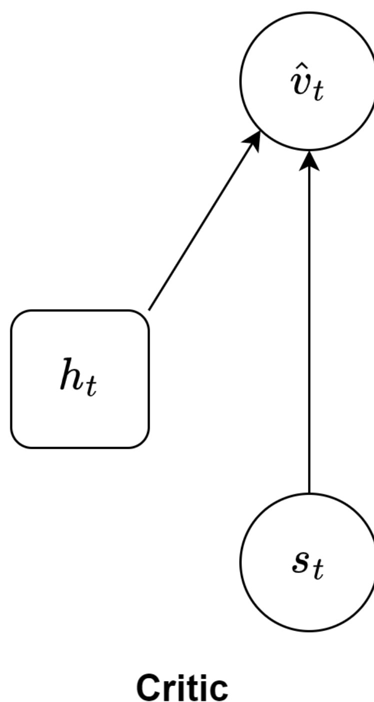


Figure 4.7 The Critic approximates the state-value function: given a latent state, it estimates the values of the values of that state.

From the imagined trajectories, the values (estimated by the critic in Figure 4.7), rewards, and, optionally, continue probabilities are predicted by the reward model, the critic, and the continue model respectively. These predicted quantities are used to compute the λ-returns TD(λ), i.e., the target values to use for the actor and critic losses.

A consideration must be done on the composition of the imagined trajectories, it turned out that the starting latent state (i.e., the posterior: the one computed by the representation model from the recurrent state and the embedded observations) must be excluded from the trajectory because it is not homogeneous with the rest of the trajectory. This was one of the first difficulties encountered in the implementation of Dreamer, and without this expedient, the agent would not converge.

It is possible to notice that when the λ-returns are computed, the last element in the list is "lost", this happens because to compute the TD(λ) it is necessary to have the next value, but the last imagined state has not a next value, so it is impossible to compute its value. As mentioned in [24], the lambda targets are weighted by the cumulative product of

the predicted discount factors, estimated by the continue model, so terms are weighted down based on how likely the imagined trajectory would have ended. This fundamental detail (without it the agent diverged) is not very visible in the paper, that's why we initially missed it, and only realized it by meticulously analyzing the official code. These weighted lambda targets are used in the actor (or policy) loss as follows:

$$\text{policy loss} \doteq -\underline{\gamma} \cdot \underline{\text{TD}(\lambda)}\ .$$

Where:

- $\underline{\gamma} = \left[1, \gamma, \gamma^2, \ldots, \gamma^{H-2}\right]$ is the vector of cumulative discounts to apply at each time step.
- H is the imagination horizon.
- $\underline{\text{TD}(\lambda)} = [lv_0, lv_1, \ldots, lv_{H-2}]$ is the vector of λ-returns computed over the imagined trajectory.

Now that the actor has been updated, it remains only to update the critic. We start by predicting the values without the last element of the imagined trajectory. So, the distribution of the predicted values is obtained and compared with the lambda targets through log prop. The mean of the discounted log probabilities is the critic loss.

$$\text{value loss} \doteq -\underline{\gamma} \cdot \ln q_v(\underline{\text{TD}(\lambda)})\ .$$

4.2.3 Actor

For what concerns the actor, it is necessary to open a little parenthesis, it supports continuous, discrete, and multi-discrete actions: in the first case, it produces in output the mean and the standard deviation of the actions; whereas for discrete (or multi-discrete) control it produces the logits of a categorical distribution (or the logits for each action). Let's start analyzing the continuous case, by reading the paper it is possible to understand that the output of the model is a Tanh[1] mean scaled by a factor of 5, whereas a SoftPlus [66] function is applied to

[1] https://pytorch.org/docs/stable/generated/torch.nn.Tanh.htmlhttps://pytorch.org/docs/stable/generated/torch.nn.Tanh.html

the standard deviation, to avoid negative values. The paper does not specify in detail the two formulas, so only after reading the code one can derive the two formulas:

$$\mu_a = 5 \cdot \tanh(\mu_\phi / 5)$$

$$\sigma_a = \text{softplus}(\sigma_\phi + \chi) + 0.1 \ .$$

Where:

- $\chi \doteq \ln e^{5-1}$ us the *raw init std* value.
- μ_π is the mean computed by the actor.
- σ_π is the standard deviation computed by the actor.

It is possible to notice that, for the standard deviation, the output of the model is incremented by a *raw init std* and then the SoftPlus is applied; in the end, a minimum amount of std is added to the result. Finally, the normal distribution is transformed using a Tanh and the correct event shape is set using the Independent distribution. Let's turn on the discrete (or multi-discrete) case, no transformations are applied to the model output, which is used as logits of a one-hot categorical distribution for each action (one for discrete control and more than one for multi-discrete control).

The last thing to talk about is the possibility of using the actor in training or test mode: the former aims to produce a sample with gradients as output, whereas the latter seeks to select the best possible actions for a given state. In the continuous case, the sample with gradients is obtained through reparameterization sampling [34, 46], and the best possible action (according to the learned policy) is given by the Tanh mean of the actions; in the discrete (or multi-discrete) control case, on the other hand, straight-through gradients [7] are used for sampling during the latent imagination, while the mode of the one-hot categorical distributions is exploited for selecting the best possible actions.

4.2.4 Environment Interaction

The last critical aspect to analyse is the environment interaction, i.e., the agent that selects actions and gets rewards and observations from the environment. There are two types of environment interaction: one used during training in which the agent selects actions with some noise to increase the exploration of the environment and in which the episodes are saved in the replay buffer; instead, in the second one, the agent plays to the best of his ability, always choosing the best action based on the policy he has learned. Since the agent learns to select actions entirely in the latent space, it is necessary to encode the observations and compute the latent state. For this reason, we need the following models: the encoder to encode the observations received from the environment, the recurrent model to compute the recurrent state (i.e., the deterministic part of the latent state), the representation model to compute the stochastic part of the latent state, and the actor to select actions from the latent states.

These models are wrapped in the player class, in which two methods are defined. The first one exploits the models to select the greedy actions, whereas the second adds some noise to increment the exploration of the environment and it is used only during training. Another critical aspect is that the player must keep track of the last latent state and the last played actions, indeed, it needs them to compute the next latent state, so this information must be properly initialized (the initial latent state and the initial action are set to zero).

Now we have all the information, we can talk about the simple idea behind the environment interaction, reflecting the classical standards of the RL: from the initial observation, the agent selects the actions and receives the information from the environment (next observations, rewards, done, truncated, and additional info), if a done (or truncated) is true, then the agent resets its state and the environment is reset and a new episode starts (we want to remember that in the training mode, the episode is added to the buffer). Moreover, before starting the training, it can be useful to collect some episodes played with random actions; this does not negatively affect the training because the agent is an off-policy algorithm, so it can learn the optimal policy by taking advantage of the experiences gathered with a different

policy (for instance, pre-emptively filling the buffer with 5000 steps played with random actions might be a good idea).

From this simple idea, you can apply some small expedient to improve performance: the first one is the possibility to add the action repeat, meaning that an action is repeated n times once it has been chosen; in [24] discovered that the best value for this hyper-parameter is 2 among all the environments they tested. The second expedient is to limit the maximum number of steps in a single episode, to prevent the agent from focusing on a single episode, thereby incentivizing exploration and diversification of experiences. Other hyper-parameters are whether to clip the rewards using the Tanh function; or whether to use grayscale instead of RGB images as observations or the maximum number of no-ops in Atari environments[2]. Instead, the last detail to focus on is also the most critical: how to save episodes in the buffer. It turned out that the best way to save experiences is the following: when the environment is reset, the initial observation is saved in the buffer with the *null* action (all zeros), the reward is zero and the done/truncated is false; then, at each step, the executed action is saved in the buffer with the next observation, the obtained reward and the obtained done/truncated.

4.2.5 Buffer

While it is necessary to save experiences in the right way, it is equally critical to retrieve this information in the right way, so we implemented a sequential replay buffer, that is the classical *First In First Out* (FIFO) replay buffer with a custom *sample* function. The idea behind it is straightforward, the function takes in input the BS, the SL, and the number of samples, i.e., how many times you want to sample BS sequences of length SL: it is equivalent to call n times the sample function, where $n = \text{n samples}$. Then the $\text{BS} \cdot \text{n samples}$ starting indices are computed randomly (the indices from where the sequences begin). After that, from the starting indices, all the indices of all the sequences are computed and the data is retrieved from the buffer. Since all the batches are independent of each other, we can fuse together the batch size with the number of samples and then split them after retrieving the data.

[2]https://gymnasium.farama.org/environments/atari/

For instance, let us suppose: BS = 16, SL = 64, and n samples = 2. We need to sample $16 \cdot 2$ trajectories of length 64. We start by sampling the starting indices of the trajectories: $[0, 2, 5, 19, 4, 12, \dots]$ then we compute the whole trajectories by adding at each starting index the vector $[0, 1, 2, 3, \dots, 63]$. in this way, we will obtain the indices of the trajectories:

$$
\begin{aligned}
\text{sequences} = [&[0,1,2,3,\dots,63],\\
&[2,3,4,5,\dots,65],\\
&[5,6,7,8,\dots,68],\\
&[19,20,21,22,\dots,82],\\
&\dots]
\end{aligned}
$$

of shape $(\text{n samples} \cdot \text{BS}, \text{SL})$. Then we can reshape it to $(\text{n sample}, \text{BS}, \text{SL})$ and permute the BS dimension with the SL one to end up with a shape of $(\text{n sample}, \text{SL}, \text{BS})$.

If the buffer is not full, then it is possible to sample the starting indices in the range $[0, \text{pos})$, where pos is the index of the next empty cell in the buffer. Whereas, if the buffer is full (meaning that the oldest experiences are being replaced by the newest ones), the starting indices cannot fall in [pos˘SL, pos), being careful that pos˘SL is not negative, otherwise it is needed to circularly go back and remove some positions from the bottom of the buffer. In Figure 4.8, the three possible cases are shown: on the top the non-full buffer, on the left the case with full buffer and pos˘SL ≥ 0, and on the right the case with full buffer and $\text{pos} - \text{SL} < 0$. The green area contains the valid start indices; whereas in the red area, there is no data. Finally, the grey area contains all the indices that cannot be sampled as start indices but can be included in a sampled sequence.

4.2.6 Critical Aspects

In this section, we want to highlight the most significant details that led to the convergence of this algorithm: first, the dimension of the models, we tried to reduce their dimension to speed up the training and reduce the amount of memory needed, but the results were not good enough to say that the algorithm converged. Moreover, it turned out that the GRU is the best

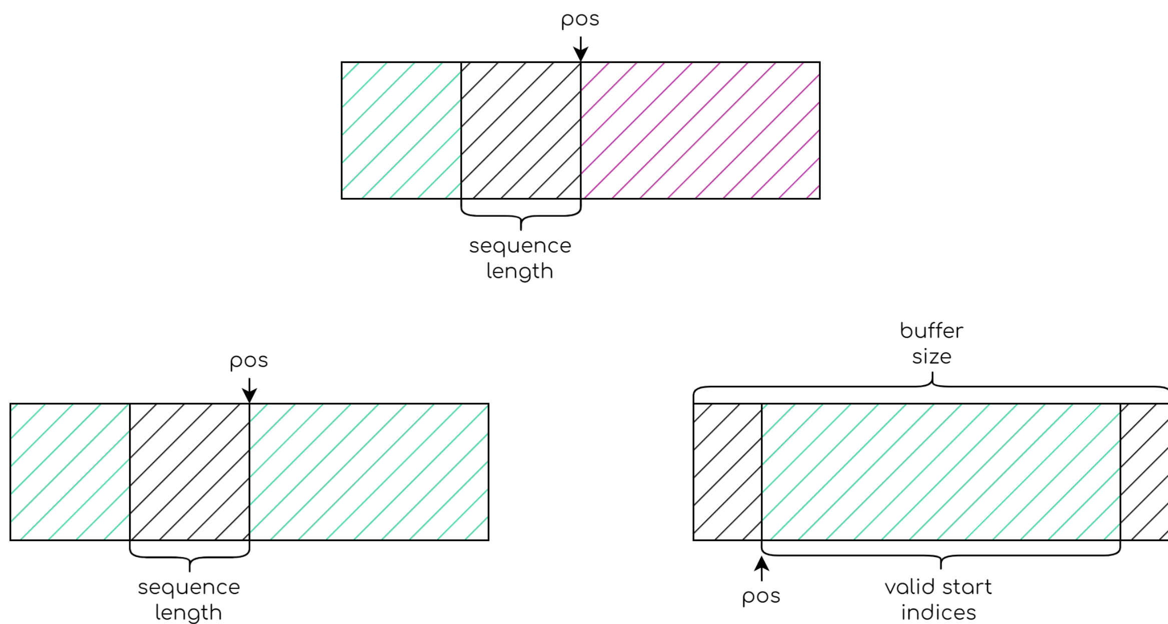


Figure 4.8 The possible scenarios during sampling from the Sequential Replay Buffer: *(i)* On the top, the case in which the buffer is not full, so the sequence can start in the green area and continue in the grey one, without falling into the red area. *(ii)* On the bottom, the two cases where the buffer is full, on the left of the index `pos` there are new collected experiences, instead, on the right there are old experiences. It is possible to start sequences in the green area, so as to be sure not to mix new and old experiences into a single trajectory.

recurrent unit for the recurrent model and that is not possible to process the whole sequence at once (as mentioned before). Speaking of models, it was found that the activation functions have a considerable impact on performance, so it is recommended to use the ones specified in [24], for instance, the encoder has a ReLU [2] after each convolutional layer (even the last layer), whereas the observation model does not have the ReLU activation function after the last transpose convolutional layer, as well as all the MLPs do not have the ELU [13] activation function after the last dense layer.

Another aspect not to be underestimated is the proper use of distributions and the correct handling of model outputs: the two most sensitive to changes in hyper-parameters are the action distribution and the stochastic state distribution, in particular, the SoftPlus function is applied to the standard deviation of both the stochastic state and the actions and then a minimum standard deviation is added to the result of the SoftPlus (the actor has also an init std $= 5$ that is added before applying the SoftPlus), moreover, the stochastic state distribution must be a multivariate normal diagonal distribution. The mean and standard deviation of the stochastic state are computed as follows:

$$\mu_s = \mu_\theta$$

$$\sigma_s = \text{softplus}(\sigma_\theta) + 0.1 \ .$$

Where:

- μ_θ is the mean of the distribution in output of either the representation or transition model.
- σ_θ is the standard deviation of the distribution in output of either the representation or transition model.
- min std $= 0.1$ is the minimum amount of standard deviation for the stochastic state (either posterior or prior).

Instead for the actor, as described in the previous section, it is better to scale the mean by 5 and apply a Tanh transform, this aspect was mentioned in the paper, but there were not

enough details to correctly implement it. The formulas for the computation of the mean and standard deviation for the actor (only for continuous actions) are the following:

$$\mu_a = 5 \cdot \tanh(\mu_\phi/5)$$

$$\sigma_a = \text{softplus}(\sigma_\phi + \chi) + 0.1\ .$$

Where:

- $\chi \doteq \ln e^{5-1}$ us the *raw init std* value.
- μ_π is the mean computed by the actor.
- σ_π is the standard deviation computed by the actor.

All the other distributions are Normal distributions except the distribution to estimate how probably the episode ends at that time step (computed by the continue model), which is a Bernoulli. The only thing to pay attention to is the batch shape and event shape of the distributions, for instance, the reconstructed observation distribution must have batch shape equal to (SL, BS) and event shape equal to (C, H, W), where C is the number of channels, H is the height, and W is the width of the image.

Moving from the modes to the training recipe, the two key aspects were: *(i)* the removal of the initial state from the imagined trajectory (the one computed by the representation model, i.e., the first latent state of Figure 4.6); *(ii)* the introduction of the discounts, computed from the probabilities that the episode ends at that time step (estimated by the continue model).

Finally, as previously said, the environment interaction and the buffer gave the decisive breakthrough: the observations are brought into the range $[-0.5, 0.5]$, the action is associated with the next observations, and the initial observations are associated with the all-zero-action, and the buffer correctly samples the sequences from past experience.

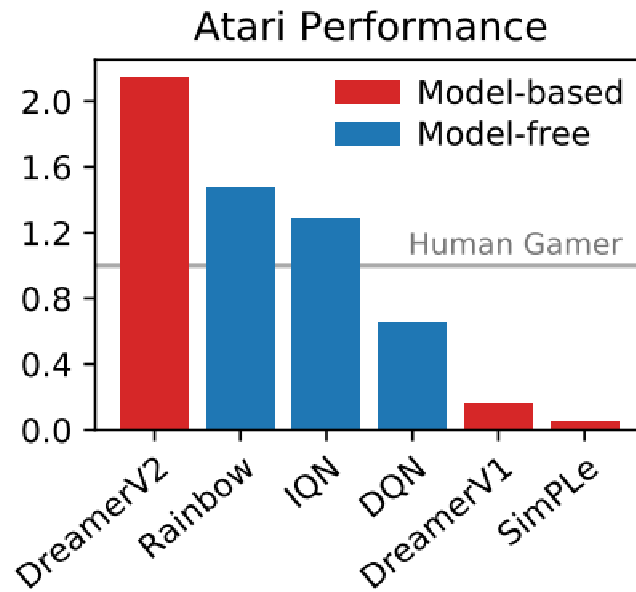


Figure 4.9 This picture [27] shows the performance on the Atari environments, showing how Dreamer-V2 achieves superior performance compared to the best model-free algorithms and other model-based algorithms present at the time. Moreover, it shows how Dreamer-V2 is able to outperform humans in the Atari environments.

4.3 Dreamer V2

As explained in the previous section, Dreamer is a novel algorithm that leverages the imagination of trajectories to learn the optimal policy. The potential is high, so the creators have devoted a lot of work to improve it: initially Dreamer- V1 was thought for continuous control, but it was not able to outperform the best model-free algorithms on discrete environments. Dreamer-V2 [27] was the first algorithm to outperform model-free algorithms in various discrete and continuous tasks (Figure 4.9).

Dreamer-V2 improves its predecessor by modifying some details, but leaving the main structure unchanged. As one can notice in Figure 4.10, one of the main changes is definitely the change in stochastic state representation (posterior and prior), in Dreamer-V2 they are represented with a set of categorical distributions instead of as a multivariate normal diagonal distribution. Moreover, the KL loss between them is slightly modified to improve

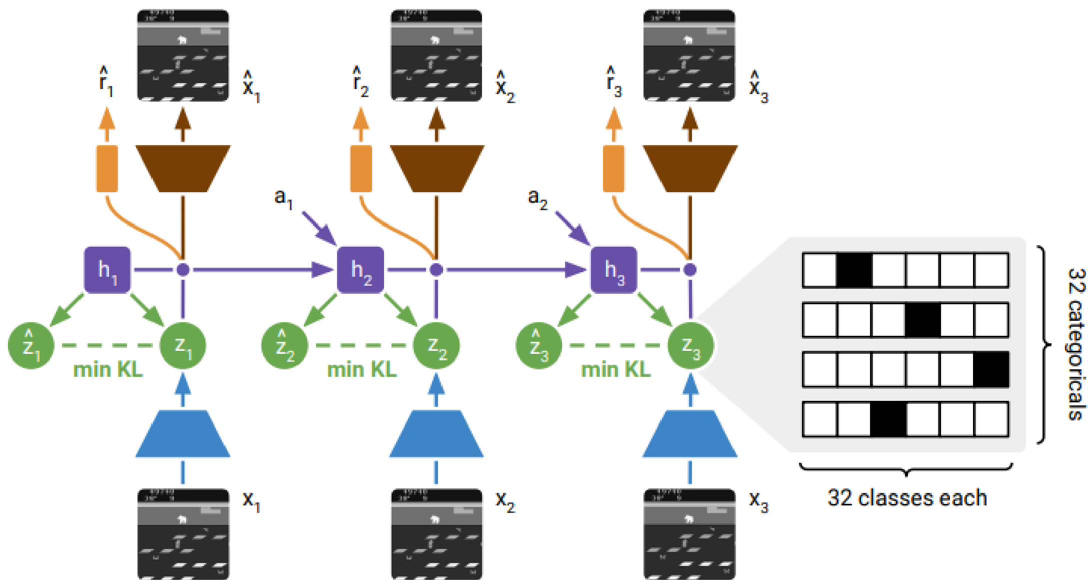


Figure 4.10 The world model of Dreamer-V2 [27]. It is possible to notice that it maintains the idea of Dreamer-V1: the observations are encoded and used to compute the posterior z_t. The prior $\hat{z}_t$ is computed from the history h_t (i.e., the recurrent model) and it is learned to be similar to the posterior. From the latent state (z_t concatenated with h_t), the observations are reconstructed $\hat{x}_t$ and the reward $\hat{r}_t$ is predicted. Finally, the posterior and the prior are now represented as 32 categoricals of 32 classes each.

performance. Finally, the actor and the critic learning are improved to add robustness to the algorithm and enable it to achieve better performance.

4.3.1 Stochastic States

As mentioned before, the main difference is the representation of the posterior and the prior. Dreamer-V2 learns a discrete latent state of the environment as a mixture of 32 categoricals with 32 classes each. This discretization of the latent state is helpful for different reasons, as well-explained in [27]:

- A categorical prior can perfectly fit the aggregate posterior, because a mixture of categoricals is again a categorical. In contrast, a Gaussian prior cannot match a mixture of Gaussian posteriors, which could make it difficult to predict multi-modal changes between one image and the next.

- The level of sparsity enforced by a vector of categorical latent variables could be beneficial for generalization. Flattening the sample, from the 32 categorical with 32 classes each, results in a sparse binary vector of length 1024 with 32 active bits (since the categorical are represented with *onehot* encoding).
- Despite common intuition, categorical variables may be easier to optimize than Gaussian variables, possibly because the straight-through gradient estimator ignores a term that would otherwise scale the gradient. This could reduce exploding and vanishing gradients.
- Categorical variables could be a better inductive bias than unimodal continuous latent variables for modeling the non-smooth aspects of Atari games, such as when entering a new room, or when collected items or defeated enemies disappear from the image

4.3.2 KL Loss

During the *dynamic learning* phase, i.e. the one in which the world model is learned, the prior distribution over the latent states, estimated by the transition model, and the posterior one, estimated by the representation model, are learned so to minimize their KL divergence:

$$
\begin{aligned}
\mathrm{KL}(P\|Q) &= \int_{\mathcal{X}} p(x)\log\left(\frac{p(x)}{q(x)}\right)dx \\
&= \int_{\mathcal{X}} p(x)\log(p(x))dx - \int_{\mathcal{X}} p(x)\log(q(x))dx \\
&= \mathrm{H}(P,Q) - \mathrm{H}(P)\ .
\end{aligned}
$$

Where P, Q, $\mathrm{H}(P,Q)$, and $\mathrm{H}(P)$ are the posterior and prior distributions, the cross-entropy between the posterior and the prior, and the entropy of the posterior distribution respectively.

The KL loss serves two purposes: it trains the prior toward the representations, and it regularizes the representations toward the prior. However, since learning the prior is difficult, we want to avoid regularizing the representations toward a poorly trained prior. To overcome this issue, the divergence between the posterior and the prior is replaced with the following:

Algorithm 1: Straight-Through Gradients with Automatic Differentiation

```
sample = one_hot(draw(logits))            # sample has no gradient
probs  = softmax(logits)                  # want gradient of this
sample = sample + probs - stop_grad(probs)  # has gradient of probs
```

Figure 4.11 The algorithm of the Straight-Through Gradients with Automatic Differentiation trick: the sample is extracted from the Categorical distribution, then the probabilities are computed. Finally, these probabilities are summed and subtracted from the sample. When we subtract the probabilities, we need to stop the gradients, to prevent them from nullifying those of the probabilities that were summed.

$$\alpha \mathrm{KL}(\mathrm{sg}(P) \| Q) + (1-\alpha)\mathrm{KL}(P \| \mathrm{sg}(Q)) \ .$$

With $\alpha = 0.8$ and sg stands for "stop-gradient", meaning that, during the backpropagation that variable will be not used to compute the gradients. By scaling up the prior cross entropy relative to the posterior entropy, the world model is encouraged to minimize the KL by improving its prior dynamics toward the more informed posteriors, as opposed to reducing the KL by increasing the posterior entropy.

4.3.3 Straight-Through Gradients

Since in Dreamer-V2, the stochastic states have a different representation with respect to the one in Dreamer-V1, it is necessary to find a technique to backpropagate gradients through a drawn sample from a Categorical distribution. The most suitable trick is the straight-through gradients with automatic differentiation trick [7]. This enables the world model to receive gradients from both the KL loss and the sampled latent states, used to predict rewards and discounts, reconstruct observations, predict values, and select actions. In Figure 4.11, the algorithm for the Straight-Through gradients trick is shown.

4.3.4 Critic

As in Dreamer-V1, the critic is trained toward a value target, i.e., the λ-returns computed from the predicted rewards and values in the imagined trajectories. The difference is that

the value learning is stabilized using a target network (called *target critic*) whose parameters are updated with an *Exponential Moving Average* (EMA) at a certain frequency (every 100 gradient steps). The *critic loss* is defined as follows:

$$\mathcal{L}(\xi) \doteq \mathbb{E}_{p_\phi, p_\psi}\left[\sum_{t=1}^{H-1} \frac{1}{2}\left(v_\xi(\hat{z}_t) - \mathrm{sg}(V_t^\lambda)\right)^2\right] .$$

Where:

- ξ are the parameters of the critic.
- p_ϕ denotes the predictors of the world model (RSSM and reward predictor).
- p_ψ denotes the actor predictor.
- V_t^λ is the target value at the imagined time step t.
- $\hat{z}_t$ is the imagined stochastic state.

4.3.5 Actor

As in Dreamer-V1, the actor is trained to maximize the prediction of long-term future returns made by the reward model and the critic. To incorporate intermediate rewards more directly, the actor is trained to maximize the same λ-returns that were computed for training the critic. To improve actor learning, in Dreamer-V2, the unbiased but high-variance Reinforce gradients are combined with biased but low-variance straight-through gradients. Moreover, the entropy of the actor is regularized to encourage exploration, where feasible.

The actor loss is defined as follows:

$$\mathcal{L}(\psi) = \mathrm{E}_{p_\phi, p_\psi}\left[\sum_{t=1}^{H-1}\left(-\rho \ln p_\psi(\hat{a}_t|\hat{z}_t)\ \mathrm{sg}\left(V_t^\lambda - v_\xi(\hat{z}_t)\right)\right) - (1-\rho)V_t^\lambda - \eta \mathrm{H}[\hat{a}_t|\hat{z}_t]\right]$$

Where:

- ψ are the parameters of the actor.

- $-\ln p_\psi(\hat{a}_t|\hat{z}_t)\ \mathrm{sg}\left(V_t^\lambda - v_\xi(\hat{z}_t)\right)$ is the Reinforce term, which aims to maximize the probability of the sampled actions of the actor, weighted by the values of those actions.

- $-V_t^\lambda$ is the *dynamics* term, which aims to output actions that maximize the prediction of long-term future rewards made by the critic. Thanks to straight-through gradients we can backpropagate through the sampled actions and state sequences.

- $\eta \mathrm{H}[\hat{a}_t|\hat{z}_t$ is the entropy of the actions, regularized by η.

- $\rho \in [0,1]$ is the parameter to balance the Reinforce gradients and the straight-through gradients.

- p_ϕ denotes the predictors of the world model (RSSM and reward predictor).

- p_ψ denotes the actor predictor.

- H is the imagination horizon.

4.3.6 Buffer

The last difference concerns the buffer: in Dreamer-V2, the buffer becomes an "Episode Buffer". This means that episodes are stored separately, and that, when you sample from it, it is not possible for a trajectory to consist of two different episodes. This means that episodes that are shorter than the SL cannot be stored in the buffer.

When sampling from the Episode Buffer, the starting index of the sequence must be in the range $[0, T - \mathrm{SL})$, where T is the length of the episode. Sometimes, though, it can be helpful to favor sampling the last steps of the episode, so that you have the *done* in the trajectory. For this reason, the Episode Buffer allows you to prioritize the end of episodes: instead of sampling the initial index in the range $[0, T - \mathrm{SL})$, it is sampled in $[0, T - \mathrm{SL})$ and then the minimum between the sampled starting index and $T - \mathrm{SL} - 1$ is taken:

$$\min\left(X \sim U(0, T-1), T - \mathrm{SL} - 1\right)\ .$$

Finally, since the buffer is not infinite in size, when an episode is added to the buffer and there is not enough space, then the minimum number of episodes is removed from it, starting from the oldest.

4.4 Dreamer V3

As one can imagine, the structure of Dreamer-V3 [28] is similar to the one of Dreamer-V1 and Dreamer-V2, as shown in Figure 4.12. This algorithm enables to solve the tasks across several domains with fixed hyper-parameters. Like its predecessors, Dreamer-V3 is an off-policy agent (i.e., it can learn the optimal policy from a different policy), and a model-based algorithm, meaning that it learns the dynamics of the environment by creating a latent representation of the world and exploits this latent representation to imagine consequences of it actions.

In the next sections, the differences between Dreamer-V2 and Dreamer-V3 will be described. There are no significant differences in the main idea of the algorithm, but there are a lot of little details that are changed and that significantly improved the performance.

4.4.1 Differences with respect to Dreamer V2

We are now going to list the differences between Dreamer-V3 and its predecessor. As anticipated before, the components of the two agents are the same (as the ones employed in Dreamer-V1), thus we will not go into detail about the components of the agent. The only thing to note here is that the continue model is always present and it predicts the continues (whether or not the episode continues), instead of predicting the probability of the likelihood of continuing the episode as done in Dreamer-V2.

Symlog

As mentioned before, Dreamer-V3 aims to solve tasks in different domains, a difficult challenge because of the varying scales of inputs, rewards, and values. To solve this issue, Hafner et al. in [28] propose to use *symlog predictions*, i.e., given a neural network $f(x,\theta)$,

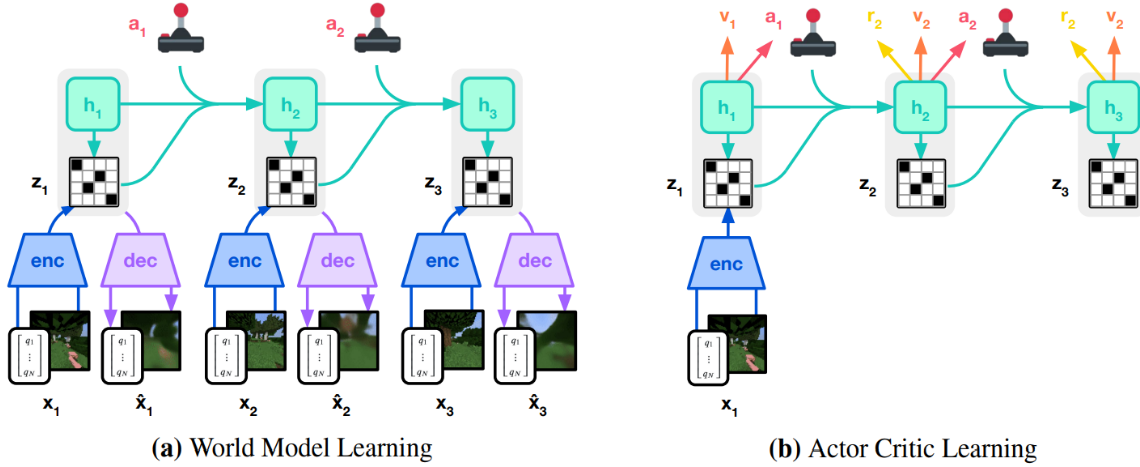


Figure 4.12 The dynamic and the behavior learning in Dreamer-V3: as in Dreamer-V2, the prior and the posterior are represented by various categorical distributions. The world model is used to learn the dynamics of the environment and it computes the latent states (z_t concatenated to h_t). The latent states are exploited to imagine future scenarios that are used to learn the optimal behavior and the approximated optimal state-value function in the imagined trajectories.

with inputs x and parameters θ, learns to predicted the *symlog transformed* targets: $\mathcal{L}(\theta) \doteq \frac{1}{2}\left(f(x,\theta) - \text{symlog}(y)\right)^2$. Where the *symlog* function (Figure 4.13) is defined as follows:

$$\text{symlog}(x) \doteq \text{sign}(x) \ln\left(|x| + 1\right) \ .$$

Given the neural network prediction, it is possible to obtain the non-transformed target by applying the inverse transformation (i.e., the *symexp*):

$$\text{symexp}(x) \doteq \text{sign}(x) \left(\exp\left(|x|\right) + 1\right)$$

The last detail to report is that the *symlog* prediction is used in the decoder, the reward model, and the critic. Moreover, the inputs of the MLP encoder (the one that encodes observations in vector form) are squashed with the *symlog* function.

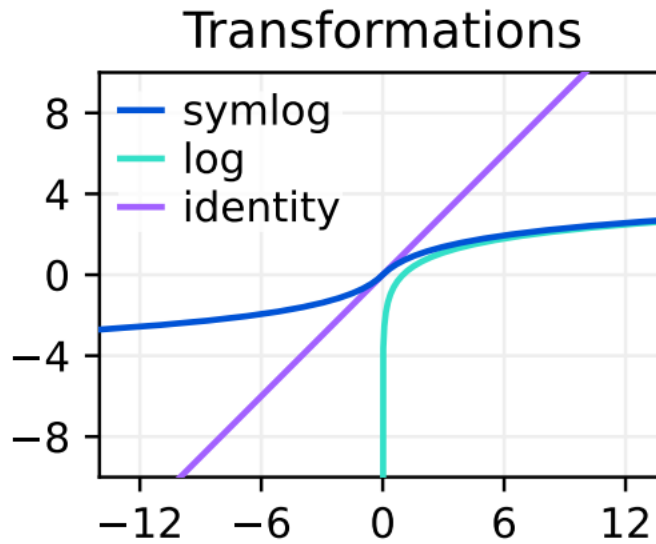


Figure 4.13 The *symlog* function: it compresses the magnitudes of both large positive and negative values, allowing to quickly move the network predictions to large values when needed.

KL Regularizer

As in Dreamer-V2, during the *dynamic learning* phase, the posterior and the prior (estimated by the representation and the transition models, respectively) are learned to minimize their KL divergence. Since 3D domains may contain unnecessary details whereas in 2D domains the background is often static and each pixel could be important for solving the task, the KL loss was slightly modified. The KL loss is divided into two losses: the dynamic and the representation.

The dynamic loss is the following:

$$\mathcal{L}_{\text{dyn}}(\phi) \doteq \max\left(1, \text{KL}(\text{sg}(p) \| q)\right) \ .$$

The representation loss is the following:

$$\mathcal{L}_{\text{rep}}(\phi) \doteq \max\left(1, \text{KL}(p \| \text{sg}(q))\right) \ .$$

Where:

- p and q are the posterior and prior distributions, respectively.

- sg means “stop gradients”, i.e., the gradients of the variable are not passed on. Thus, the variable is not considered during the backpropagation phase of the model.

- 1 is the *free bits* used to clip the dynamic and representation losses below of the *free bits*, necessary to avoid degenerate solutions where the dynamics are trivial to predict, but do not contain enough information about the inputs.

Finally, to have a good balance between complex and very detailed environments (e.g., 3D environments) and simpler and less detailed environments (e.g., static 2D environments), the two losses are scaled differently and summed together:

$$\mathcal{L}_{\text{KL}}(\phi) \doteq 0.5 \cdot \mathcal{L}_{\text{dyn}}(\phi) + 0.1 \cdot \mathcal{L}_{\text{rep}}(\phi) \ .$$

Uniform Mix

To prevent spikes in the KL loss, the categorical distributions (the one for discrete actions and the one for the posteriors and/or priors) are parameterized as mixtures of 1% uniform and 99% neural network output. This prevents the distributions from becoming near deterministic. To implement the *uniform mix*, we applied the *uniform mix* function to the *logits* returned by the NNs, as shown in Figure 4.14.

Return regularizer for the policy

The main difficulty in the Dreamer-V2 *actor learning* phase is the choosing of the entropy regularizer, which heavily depends on the scale and the frequency of the rewards. To have a single entropy coefficient, it is necessary to normalize the returns using moving statistics. In particular, they found out that it is more convenient to scale down large rewards and not scale up small rewards, to avoid adding noise.

Moreover, the rewards are normalized by an exponentially decaying average of the range from their 5th to their 95th percentile. The final actor loss becomes:

```python
import torch
from torch import Tensor
from torch.distributions.utils import probs_to_logits

def uniform_mix(self, logits: Tensor, unimix: float = 0.01) -> Tensor:
    if unimix > 0.0:
        # compute probs from the logits
        probs = logits.softmax(dim=-1)
        # compute uniform probs
        uniform = torch.ones_like(probs) / probs.shape[-1]
        # mix the NN probs with the uniform probs
        probs = (1 - unimix) * probs + unimix * uniform
        # compute the new logits
        logits = probs_to_logits(probs)
    return logits
```

Figure 4.14 The `uniform_mix` function applied to all categorical distribution, i.e., to the distribution of discrete/multi-discrete actions and to the distribution of the stochastic states (i.e., posterior and prior). The logits are first converted into probabilities, then the probabilities returned by the NN are mixed with the uniform probabilities. Finally, the new computed probabilities are converted into *logits* and returned.

$$\mathcal{L}(\theta) \doteq \sum_{t=1}^{T} \mathbb{E}_{\pi_\theta,\rho_\phi}\left[\mathrm{sg}(R_t^\lambda)/\max(1,S)\right] - \eta \mathrm{H}\left[\pi_\theta(a_t|s_t)\right] .$$

Where

- R_t^λ are the returns.
- a_t is the action.
- s_t is the latent state (z_t concatenated with h_t).
- $S \doteq \mathrm{Per}(R_t^\lambda, 95) - \mathrm{Per}(R_t^\lambda, 5)$.
- η is the entropy coefficient.
- $\mathrm{H}\left[\pi_\theta(a_t|s_t)\right]$ is the entropy of the distribution returned by the actor.

Critic

As in Dreamer-V2, there are two critics: the critic and a *target critic* (updated with an exponential moving average every time the parameters of the models are updated). Differently from Dreamer-V2, in Dreamer-V3, the λ-returns are computed with the values estimated by the critic and not the values estimated by the *target critic*. Moreover, the critic is trained to correctly estimate both the discounted returns and the *target critic* predictions.

Another difference is that the critic learns the *twoho*-encoded *symlog*-transformed returns, to be able to predict the expected value of a widespread return distribution. So, the *symlog*-transformed returns are discretized into a sequence of $K = 255$ equally spaced buckets, since the critic outputs a softmax distribution over the buckets.

The *twohot* encoding is defined as follows:

$$\mathrm{twohot}(x)_i \doteq \begin{cases} |b_{k+1} - x| \;/\; |b_{k+1} - b_k| & \text{if } i = k \\ |b_k - x| \;/\; |b_{k+1} - b_k| & \text{if } i = k+1 \\ 0 & \text{otherwise} \end{cases}$$

Where:

- x is the input to encode.
- i is the index of the *twohot* encoding.
- b_k is the value of the k-th bucket.
- $k = \sum_{j=1}^{B} \delta(b_j < x)$.

In this way a number x is represented by a vector of K numbers, all set to zero except for the two positions corresponding to the two buckets among which is situated x. For instance, if you have 5 buckets which equally divide the range $[0,10]$ (i.e., the 5 buckets are: $[0,2.5,5,7.5,10]$) and you have to represent the number $x = 5.5$, then its two hot encoding is the following:

$$\text{twohot}(5.5) = [0,0,0.8,0.2,0]$$

Because 5.5 is closer to bucket 5 than bucket 7.5.

Models

The last differences concern the hyper-parameter used, for instance, the used activation function is the SiLU [14]. Moreover, all the models use the *LayerNorm* (LN) [3] on the last dimension, except for the convolutional layers that apply the LN only on the channels dimension. The last detail is the presence of the bias in the models, in particular, all the layers followed by a LN are instantiated without the bias.

Moreover, in [28], the models are initialized in a custom way: all the models are initialized in a way similar to the *xavier normal* initialization [23], except for the heads of the actor, the last layer of the transition, representation, continue and encoder models that are initialized with a *uniform* initialization and the last layer of the critic and the reward model that are initialized with all zeros (to speed up the convergence).

Buffer

Finally, the last difference with Dreamer-V2 is the buffer. First, the buffer returns to be equal to the one of Dreamer-V1 (Figure 4.8). Moreover, it differs from both Dreamer-V1 and Dreamer-V2 on how experiences are stored in the buffer: in Dreamer-V2 each action was associated with the next observations, instead in Dreamer-V3, the actions are associated with the observations that have led the agent to choose those actions, so in Dreamer-V2 an action was associated to its consequences, instead in Dreamer-V3 the observation is associated to the next action to be performed.

4.5 Plan2Explore

In Dreamer [24, 27, 28], we saw that the world model is reliably learned by the world model. P2E [51] is an ad-hoc method to learn the world model by exploring the environment and increase the generalization of the world model so that different tasks can be learned with less effort.

The main idea is to replace the reward of the task with an *intrinsic reward* that estimates the level of novelty of a state of the environment. The newer the state, the more intrinsic reward is given to the agent. If the model visits always the same state, the intrinsic reward will be low, so the agent is pushed to visit states it has never encountered before.

4.5.1 Ensembles

Now the question that arises is, how does one compute the novelty of a state? Sekar et al. [51] introduced the ensembles (Figure 4.15), i.e., several MLPs initialized with different weights, that try to predict the embedding of the next observations (provided by the environment and embedded by the encoder). The more similar the predictions of the ensembles are, the lower the novelty of the state. Indeed, novelty comes from the disagreement of the ensembles, and the models will converge towards more similar predictions for states that are visited many times.

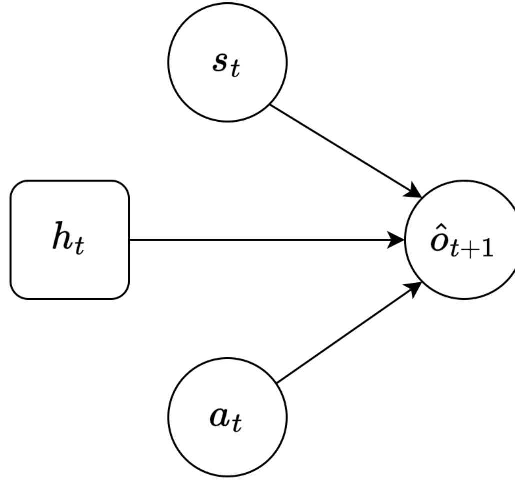


Figure 4.15 The Ensemble: it takes in input the latent state, and the previous action to estimate the next observation it will receive from the environment. The latent state is composed of the posterior s_t concatenated with the recurrent state h_t. Several ensembles are exploited to compute the novelty of a state: the greater the disagreement between them, the newer the state. Indeed, the more time a state is visited, the more trained the ensembles are to predict the next observation from that state, so, the more precise and similar will be their predictions.

A note should be made about the prediction of the embedding of the next observation: the ensemble takes as input the latent state, composed by the actual stochastic state (i.e., the posterior computed by the transition model) and the recurrent state, and the performed action (the one selected from that latent state).

4.5.2 Intrinsic Reward

Now we need to measure the level of disagreement between the ensembles: in the solution proposed in [51], the disagreement is given by the *intrinsic reward* (ir), i.e., the variance of the outputs of the ensembles.

Definition 4.1 (Intrinsic Reward)**.** The *intrinsic reward* ir is the variance of the predictions of the ensembles:

$$\mathrm{ir} = \frac{1}{K-1} \sum_k \mu_k(s_t, h_t, a_{t-1}) - \mu'$$

Where K is the number of ensembles, $\mu_k(s_t, h_t, a_{t-1})$ is the output of the k-th ensemble at time step t, and $\mu' = \frac{1}{K}\sum_k \mu_k(s_t, h_t, a_{t-1})$ is the mean of the outputs of the ensembles. These intrinsic rewards are computed during the imagination phase of the training and are used instead of the reward obtained from the environment. In this way, the actor is trained to explore the environment, trying to maximize the intrinsic reward, instead of, learning to maximize the expected return of the task.

4.5.3 Zero-shot vs Few-shot

Since Dreamer is an off-policy algorithm, it is possible to train the agent to solve the task during the exploration. In this way, the agent learns the optimal policy using the data collected when exploring. With the world model trained to explore the environment, one can test:

- In a *zero-shot* setting whether the exploration experience is useful to learn the task at hand: given the task rewards (the ones that the environment returns at every step and that represent the task to be solved) obtained during the exploration, the agent is able to learn a behavior that also solves the task.
- in *few-shot* setting whether fine-tuning the agent with few interactions with the environment helps to improve the performances further. In this setting the agent will collect new experiences with the intent to maximize its performance in solving the task: it is no more interested in exploring the environment.

The key difference then is in what experiences are used to learn the task: in the first case the agent is trained on the so-called *exploration data*, i.e., the agent selects the actions to maximize the novelty of the states. Whereas, in the second case, the agent selects actions to solve the task, and these experiences are used to improve the learned policy.

CHAPTER 5

Tools and Technologies

In this chapter, the SheepRL[1] framework will be presented: it is an open-source framework that provides several RL algorithms in *coupled* or *decoupled* form. It is built on top of PyTorch [44] and PyTorch Lightning [15] and Lightning Fabric[2] to enable distributed training in a simple and scalable way. Moreover, SheepRL leverages Hydra [64] to dynamically create a hierarchical configuration for your experiments by composition and override it through *config* files and the command line. Finally, SheepRL allows monitoring of the experiment thanks to TensorBoard (provided by TensorFlow [1]).

SheepRL aims to be in between CleanRL [31] and Stable-Baselines3 [45] or RLLib [35] (built on top of the Ray [41] framework): the main idea is to have a simple framework that allows you to have all the logic of the algorithm in a few files, but at the same time, structure the more marginal components so that they can be generalized for all algorithms. So, on one hand, he supports the idea of CleanRL of trying to show the logic of the algorithm clearly; but at the same time, it does not go to extremes, since for some algorithms it would be unthinkable to have the entire algorithm in the same file.

SheepRL provides both on-policy and off-policy algorithms, and model-free and model-based agents: from benchmark algorithms such as the on-policy, model-free PPO (both classic and recurrent versions) and the off-policy, model-free SAC, to more structured and

[1]https://github.com/Eclectic-Sheep/sheeprl
[2]https://lightning.ai/docs/fabric/stable/

complex off-policy, model-based algorithms, such as SAC-AE [65], Dreamer (V1, V2, and V3), and P2E. These are the starting algorithms, another strength of SheepRL is that it can add new algorithms easily and effectively, either by taking advantage of some classes already present (e.g., models or buffers) or by implementing all the necessary customized components.

Finally, SheepRL provides a series of environments to be able to start training and conducting experiments immediately with the agents implemented in the framework: from simple environments such as the MuJoCo environments [58] (either provided by Gymnasium [59] or DeepMind Control [56, 57]), to the Atari environments provided by gymnasium [59], to more complex environments such as Crafter [22], or DIAMBRA environments [43], or Minecraft ones [19, 16]. All the environments are converted to meet the directives provided by Gymnasium, so where these directives were not met, special wrappers were implemented to unify the interaction between the environments. For example, many environments have an interface like the one proposed by Gym [8] (the predecessor of Gymnasium), so, a custom wrapper was created to convert the interface to Gymnasium.

5.1 SheepRL

Let us present SheepRL, an open-source RL framework. SheepRL wants to be at the same time simple and scalable thanks to Lightning Fabric. Moreover, in many RL repositories, the RL algorithm is tightly coupled with the environment, making it harder to extend them beyond the Gymnasium interface. We want to provide a framework that allows to easily decouple the RL algorithm from the environment so that it can be used with any environment.

5.1.1 Coupled vs Decoupled

The algorithms are in two forms: *(i) Coupled* (Figure 5.1): the agent interacts with the environment and executes the training loop iteratively. *(ii) Decoupled* (Figure 5.2): there are at least two processes, one for the player (i.e., the process that interacts with the environment)

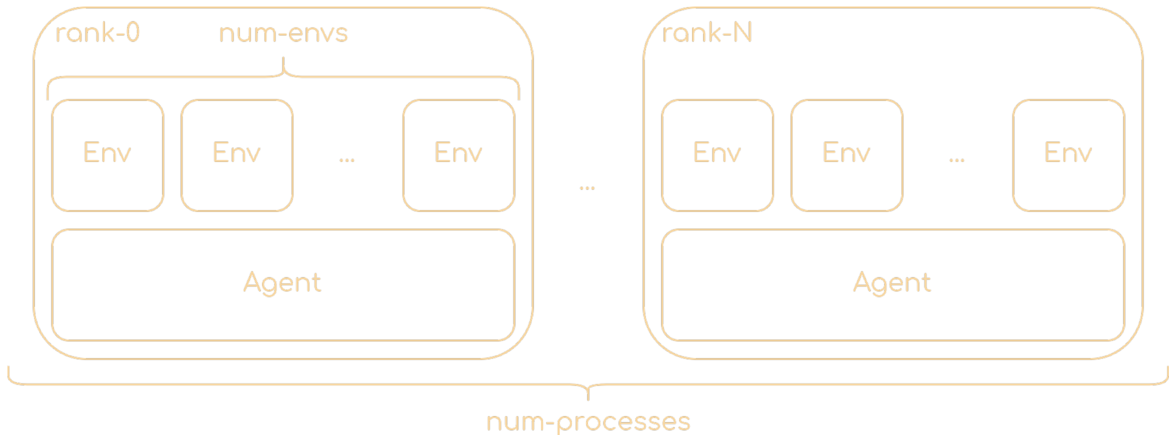


Figure 5.1 The coupled version of the algorithms: several processes can be used to distribute training. All the processes contain both the environment and the agent, which means that different instances of the environment are created. Each process interacts with its local instances of the environment, and then each process executes the training loop: the parameters of the agent models are shared and synchronized thanks to Lightning Fabric.

and at least one for training (i.e., it executes the training loop with the data collected by the zero-rank process).

The coupled version of the algorithms (Figure 5.1) allows training to be distributed among multiple processes, let us define k as the number of processes spawned for distributed training. Each process has m local instances of the environment with which the agent interacts and every process has a local buffer in which to store the collected experiences. This means that each process performs both the environment interaction and the training loop. During training, data may or may not be shared between processes; it is the user's choice. If data is shared, then each process can see the data collected from other processes, otherwise, each process sees only the data it collected. Anyway, the weights of the models are automatically synchronized by Lightning Fabric, so as to keep the training stable.

In the decoupled version of our algorithms, we employ a distributed training approach that involves multiple processes, with a minimum of two processes in play. The zero-rank process, often referred to as the *player*, is responsible for managing multiple instances of the environment and performing the environment interaction phase. On the other hand, all the remaining processes called *trainers*, are primarily tasked with executing the training loop. Here's how the decoupled algorithm works:

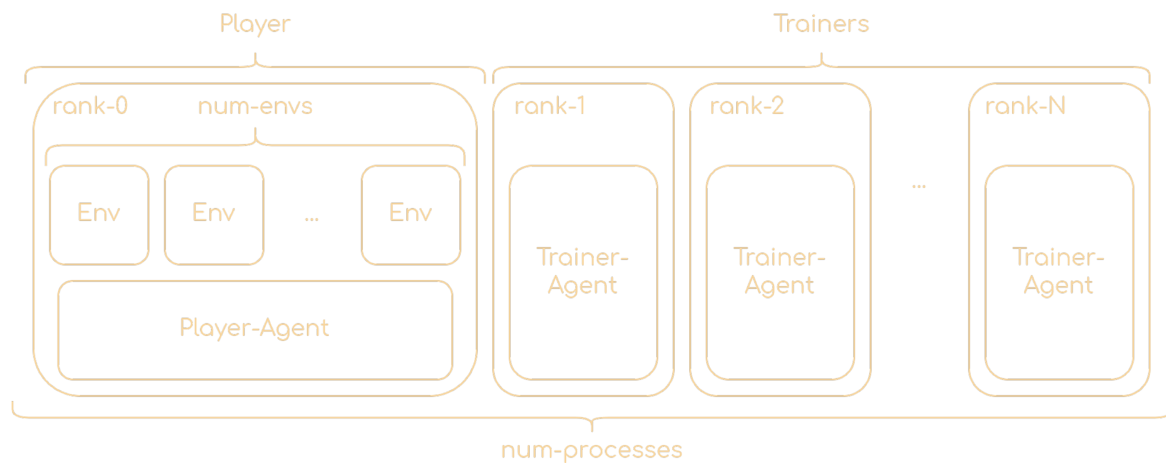


Figure 5.2 The decoupled version of the algorithms: several processes can be spawned to distribute training (at least two processes). The zero-rank process (called *player*) contains m instances of the environment and performs the environment interaction. Whereas, all the other processes (called *trainers*) execute the training loop. The player interacts with the environment and collects experiences that are sent to the trainers. The trainers receive the data from the player and execute the training loop: again, Lightning Fabric synchronizes the weights of the models during training. At the end of the training loop, the rank-1 process (a trainer) sends the updated weights to the player.

- Player process: the player process is dedicated to interacting with the environment. It manages and operates multiple instances of the environment, actively collecting experiences during these interactions. These experiences are subsequently transmitted to the trainers for further processing.

- Trainer processes: The trainer processes are responsible for executing the core training loop. They receive the collected data and experiences from the player process. Within this training loop, the Lightning Fabric system ensures the synchronization of model weights during the training process, promoting consistent and coherent learning across all trainers.

- Weight synchronization among trainers: as the training loop progresses, Lightning Fabric plays a crucial role in harmonizing the model weights among all trainers. This synchronization is essential to maintain consistency and prevent divergence in the learning process.

- Weight update: At the conclusion of the training loop, one of the trainer processes, rank-1, is designated to transmit the updated model weights back to the player process. This step ensures that the player, responsible for environment interaction, always has access to the latest model parameters.

5.1.2 Environments

Another strength of SheepRL is the environments it provides, indeed, it provides a number of very different environments with the same interface, allowing one not to worry about which environments are compatible with which algorithm; thus avoiding wasting time during implementation and debugging. Table 5.1 summarises the environments available in SheepRL, with their respective characteristics.

All the environments are converted to a Gymnasium-like interface[3], in particular, each environment should have:

- A `reset` method to initialize the environment, that returns the initial observation and a Python dictionary containing auxiliary information complementing the observations.
- A `step` method which takes in input the actions, to perform a step in the environment. This method returns the next observations, the reward, whether or not the episode is terminated (*done*), whether or not the episode is truncated (*truncated*[4]), and a Python dictionary containing auxiliary information complementing the observations.
- Two parameters: the `observation_space` and the `action_space` which specify the observation space and the action space of the environment, respectively.
- A `render` method which computes the render frames as specified by `render_mode` property.
- A `close` method to "clean up" the environment.

[3] https://gymnasium.farama.org/api/env/

[4] Whether the truncation condition outside the scope of the MDP is satisfied. Typically, this is a *time limit*, but could also be used to indicate an agent physically going out of bounds. Can be used to end the episode prematurely before a terminal state is reached.

Moreover, SheepRL imposes another constraint on the observations, they must be of type `gymnasium.spaces.Dict`[5], To allow the algorithms to be able to work with multiple observations of different types and to always work with observations of the same type for each environment. The dictionary must be flattened, so there cannot be nested dictionary as observations. In addition, it is important to specify that SheepRL provides for two types of observations: vector observations and images, both of type `gymnasium.spaces.Box`[6]. The vector observations are 1-dimensional arrays; whereas, images 3 or 4-dimensional arrays (depends on *frame stacking* parameter that will be explained later).

MuJoCo

MuJoCo [58] environments are one of the most widely used and recognized benchmarks in the literature. It is a physics engine for facilitating research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed. SheepRL provides both the version of Gymnasium[7] and the one of DM Control Suite [56]. They differ in some small details, such as the function of the reward or how an episode ends. Anyway, all the MuJoCo environments are fully described.

However, the main tasks and features remain unchanged, in particular, the observations of MuJoCo environments can be both images and vectors, and they have continuous control. For example, in Figure 5.3, it is shown the *walker walk* [36] environment, in which the agent has to learn how to walk.

Gymnasium and Atari

Another recognized benchmark is the Atari environments[8]: they are provided by *Arcade Learning Environment* (ALE) [5, 37] through the Stella emulator, and are easily installable with Gymnasium. The Atari environments provide only images as observations, moreover,

[5]https://gymnasium.farama.org/api/spaces/composite/#gymnasium.spaces.Dict
[6]https://gymnasium.farama.org/api/spaces/fundamental/#box
[7]https://gymnasium.farama.org/environments/mujoco/
[8]https://gymnasium.farama.org/environments/atari/

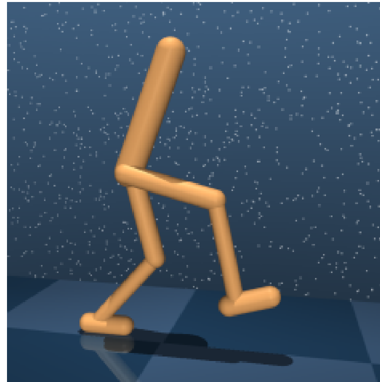


Figure 5.3 The walker walk environment provided by the DM Control Suite. The agent has to learn how to walk in the forward (right) direction by applying torques on the six hinges connecting the seven body parts.

they have discrete actions. For example, you can train your agent on famous vintage video games such as *MsPacman* (Figure 5.4), *Breakout*, *Pong*, and many others.

Furthermore, Gymnasium provides other environments that can be used as starting point for your experiments, for instance, the Box2D environments[9]. These environments all involve toy games based around physics control, using Box2D[10]-based physics and PyGame-based rendering. The observations for these environments are: *(i)* both images and vectors for LunarLander and BipedalWalker; instead, *(ii)* only images for CarRacing (Figure 5.5).

DIAMBRA

DIAMBRA Arena [43] is a platform for reinforcement learning research and experimentation, featuring a collection of high-quality environments exposing a Python API fully compliant with the Gym standard. They are episodic tasks with discrete (or multi-discrete) actions and observations composed of both images and vectors; all supporting both single-player and two-player mode, allowing to work on standard reinforcement learning, competitive multi-agent, human-agent competition, self-play, human-in-the-loop training, and imitation learning. Since DIAMBRA environments are compliant with the Gym standard, a wrapper was developed to be compliant with the Gymnasium standard.

[9]https://gymnasium.farama.org/environments/box2d/
[10]https://box2d.org/

Figure 5.4 The MsPacman Atari environment: the agent has to collect all of the pellets on the screen while avoiding the ghosts.

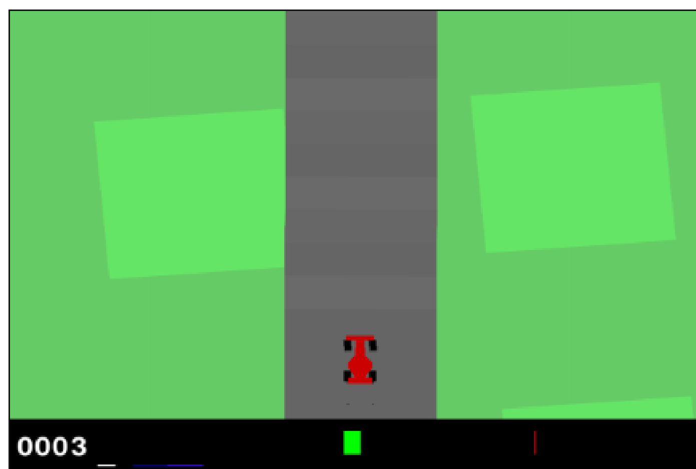


Figure 5.5 The CarRacing environment is the easiest control task to learn from pixels: a top-down racing environment. The agent has to learn to complete a lap as quickly as possible.

Figure 5.6 The “Dead or Alive ++” environment: the agent has to challenge different opponents. To advance to the next stage, he must defeat the same opponent twice (it has to win two rounds); when he passes the 8th stage, the agent has won the game. If he loses 2 games in the same stage, though, then the agent has lost and the episode ends.

Interfaced games have been selected among the most popular fighting retro games. While sharing the same fundamental mechanics, they provide different challenges, with specific features such as different types and numbers of characters, how to perform combos, health bars recharging, and so on and so forth. So, the agent’s goal is to defeat the opponent. For instance, Figure 5.6 shows the “Dead or Alive ++” game.

Crafter

Crafter [22] features randomly generated 2D worlds where the player needs to forage for food and water, find a place to sleep, defend against monsters, collect materials, and build tools. Crafter stands as a new benchmark since it requires that the agents be able to generalize on unseen situations, explore the environment, and carry out long-term reasoning. Crafter provides only images as observations and has a discrete action space, in addition, it is a partially observable MDP, since, the agent sees only a small part of the world.

MineRL

MineRL [19] is a Python library for interacting with the video game Minecraft. There are three possible tasks:

Figure 5.7 The Crafter environment: the agent receives only the image as observation, it has to retrieve the information from it, including the life level, hunger level, and inventory.

1. Navigate (Figure 5.8): the agent has to follow a compass to reach a diamond in the environment.

2. Obtain Iron Pickaxe: the agent has to craft an iron pickaxe.

3. Obtain Diamond: the agent has to craft a diamond.

MineRL has a multi-discrete action space, in SheepRL we converted it to discrete, adding the possibility of sticky actions for *attack* and *jump* actions, which means that it forces the agent to repeat the selected actions for a certain number of steps (this is easily implemented thanks to the multi-discrete nature of the actions in MineRL). Also, as suggested by [28], the `break_speed_multiplier` parameter was increased to smash objects faster, we limited the *pitch* of the agent between -60 and 60 degrees, and the observation space was modified as follows:

- The inventory is represented with a vector with one entry for each item of the game which gives the quantity of the corresponding item in the inventory.

- A max inventory vector with one entry for each item which contains the maximum number of items obtained by the agent so far in the episode.

Figure 5.8 The Navigate task of MineRL: the agent has to follow the compass to find the diamond in the environment, it is the light blue square located at the bottom left of the figure.

- The RGB first-person camera image.
- A vector of three elements representing the life, the food, and the oxygen levels of the agent.
- A one-hot vector indicating the equipped item, only for the obtain tasks (Diamond and Iron Pickaxe).
- A scalar indicating the compass angle to the goal location, only for the navigate tasks.

As Crafter, MineRL is a partially observable MDP.

MineDojo

MineDojo environments [16] are very similar to MineRL ones: multi-discrete actions with both images and vectors as observations. The main difference is that MineDojo raises an exception when the agent tries to perform a move that cannot be performed in the environment. So, it provides action masks to specify which actions can be performed by the agent at that time step. For this reason, a custom Actor for the MineDojo environments was developed, to take into account action masks. Moreover, the observations are the following:

- The inventory is represented with a vector with one entry for each item of the game which gives the quantity of the corresponding item in the inventory.

- A max inventory vector with one entry for each item that contains the maximum number of items obtained by the agent so far in the episode.
- A delta inventory vector with one entry for each item which contains the difference of the items in the inventory after the performed action.
- The RGB first-person camera image.
- A vector of three elements representing the life, the food, and the oxygen levels of the agent.
- A one-hot vector indicating the equipped item.
- A mask for the action type indicating which actions can be executed.
- A mask for the equip/place arguments indicating which elements can be equipped or placed.
- A mask for the destroy arguments indicating which items can be destroyed.
- A mask for craft smelt indicating which items can be crafted.

The actions are handled as in MineRL, so multi-discrete actions are converted into discrete ones, the `break_speed_multiplier` parameter is increased, the sticky action is added to the *attack* and *jump* actions, and the *pitch* is limited in the range $[-60, 60]$ degrees.

Wrappers

Finally, SheepRL provides some useful wrappers for the environments:

- The Action Repeat Wrapper: it allows the same action to be repeated a specified number of times.
- Reward As Observation Wrapper: it adds to the observation the reward, as a vector.
- Frame Stack Wrapper: it is applicable only to image observations, it stacks the last n frames by adding a dimension to the observations. Moreover, it allows us to set a

Table 5.1 The environments available in SheepRL: the "Obs" column can be: Both (both pixels and vector observations), Depends (on the specific environment), Pixels (only images as observation), Vector (only vectors as observations). The "Action Space" column can be Continuous, Discrete, Multi-Discrete, or All (for all the action spaces). Finally, the "Category" column can be Depends (on the specific environment), Fully Described, or Partially Observable.

Environment	Obs	Action Space	Category
MuJoCo	Both	Continuous	Fully Described
Gymnasium	Depends	All	Depends
Atari	Pixels	Discrete	Fully Described
DIAMBRA	Both	Discrete/Multi-Discrete	Fully Described
Crafter	Pixels	Discrete	Partially Observable
Minecraft	Both	Discrete	Partially Observable

dilation in the stacked frames, for instance, let us suppose to have a dilation $\mathrm{dil} = 3$ and the number of stacked frames set to $n = 4$. Let us define $[\mathrm{f}_{t-11}, \mathrm{f}_{t-10}, \dots, \mathrm{f}_{t-1}, \mathrm{f}_t]$ the last 12 frames received from the environment. Then, the stacked image observation will have the following frames: $[\mathrm{f}_{t-9}, \mathrm{f}_{t-6}, \mathrm{f}_{t-3}, \mathrm{f}_t]$. Thus, the dilation is the number of frames between one stacked frame and the following one.

- Mask Velocity Wrapper: it can be applied only to a few Gymnasium environments, such as *CartPole* or *LunarLander*, and it masks some of the values of the observations representing the velocity.

5.1.3 Algorithms

As previously stated, SheepRL provides several algorithms, some of them are on-policy, others off-policy, and there are model-free and model-based agents. Most of them can work with both vector and pixel observations and can work with both continuous, discrete, and multi-discrete actions. Table 5.2 shows which algorithms are implemented and which action space they support. Finally, it is necessary to specify that the MineDojo environments can be used only when you are training one of the Dreamer or P2E algorithms.

Table 5.2 The algorithms in SheepRL. The *Type* column can be "Coupled", "Decoupled" or "Both"; the *Recurrent* one specifies whether or not the algorithm contains a recurrent model (such as LSTM or GRU); the *Observations* column can be "Image", "Vector" or "Both"; and the *Action Space* can be "Continuous", "Discrete", "Multi-Discrete" or "Both".

Algorithm	**Type**	**Recurrent**	**Observations**	**Action Space**
PPO	Both		Both	All
PPO Recurrent	Coupled	X	Both	All
SAC	Both		Vector	Continuous
SAC-AE	Coupled		Both	Continuous
DroQ	Coupled		Vector	Continuous
Dreamer-V1	Coupled	X	Both	All
Dreamer-V2	Coupled	X	Both	All
Dreamer-V3	Coupled	X	Both	All
P2E (Dreamer V1)	Coupled	X	Both	All
P2E (Dreamer V2)	Coupled	X	Both	All

5.1.4 Buffers

Buffers are data structures that allow the agent to store the collected experiences and to use them from training. This means that there must be a method to retrieve the experiences from the buffers. Since the algorithms are very different from each other, it is necessary to have different types of buffers:

- Replay Buffer: it is the classical FIFO buffer, when it is full, the new experiences replace the oldest ones. When we need to retrieve the data from it, it randomly samples from the experiences it has stored, without considering the temporal correlation between them.

- Sequential Replay Buffer: it stores data as the Replay Buffer, but when we need to retrieve experiences from it, it returns sequences of experiences; a sequence is composed of contiguous steps in time. This buffer is useful when we are using an algorithm that has a recurrent model, such as Dreamer, and we need the data used for training to have some contiguous steps in time. The Sequential Replay Buffer does not take into account when an episode ends, so a sequence can contain two different episodes.

- Episode Buffer: like the Sequential Replay Buffer, samples batches of sequences, with the difference that it takes into account the episode to which the sampled sequences belong. In fact, it does not allow a sequence to have data from two different episodes. For this reason, the Episode Buffer stores a list of the collected episodes; when it is full, the oldest episodes are removed and the new ones are added.

- Async Replay Buffer: it is used when there are multiple environments that are not synchronized with each other, meaning that the number of steps performed in one environment can be different from the ones executed in another one. It stores a list of buffers, either Replay or Sequential Replay buffers according to need, and it samples randomly from these buffers.

5.2 PyTorch Lightning, Fabric

The utilization of PyTorch Lightning and Lightning Fabric played a pivotal role in the scaling and distribution of our model training processes in SheepRL. Lightning Fabric, in particular, offered several distinct advantages that made it an ideal choice for our distributed training needs. First and foremost, it enables to speed up the implementation process.

In addition, Fabric streamlined crucial aspects such as distributed training, hardware management, checkpoints, and logging, but it left the design and orchestration of these processes entirely in our hands. In sum, the combination of PyTorch Lightning and Lightning Fabric not only accelerated our model training but also granted us the autonomy and versatility required for the diverse and intricate experiments conducted in this work.

5.3 Hydra

Regarding Hydra, we can define a hierarchical configuration of our experiments, allowing us to define the configurations of each component separately and then assemble them to define the configuration of the experiment. In particular, we organized the configurations in the following way: *(i)* One configuration for the algorithm (Figure 5.9); *(ii)* one for the environment; *(iii)* one for the buffer; *(iv)* one for Fabric; *(v)* one for logging; *(vi)* one for the checkpoint; *(vii)* one for the optimizer; and *(viii)* one for the experiments, that assembles the other above-mentioned configurations and, if necessary, it overrides some parameters.

```yaml
# Model related parameters
layer_norm: True
dense_units: 1024
mlp_layers: 5
dense_act: torch.nn.SiLU
cnn_act: torch.nn.SiLU
unimix: 0.01
hafner_initialization: True

# World model
world_model:
  discrete_size: 32
  stochastic_size: 32
  kl_dynamic: 0.5
  kl_representation: 0.1
  kl_free_nats: 1.0
  kl_regularizer: 1.0
  continue_scale_factor: 1.0
  clip_gradients: 1000.0

  # Encoder
  encoder:
    cnn_channels_multiplier: 96
    cnn_act: ${algo.cnn_act}
    dense_act: ${algo.dense_act}
    mlp_layers: ${algo.mlp_layers}
    layer_norm: ${algo.layer_norm}
    dense_units: ${algo.dense_units}

  # Recurrent model
  recurrent_model:
    recurrent_state_size: 4096
    layer_norm: True
    dense_units: ${algo.dense_units}
```

Figure 5.9 A part of the default configuration of the Dreamer-V3 algorithm. It is possible to notice that it is possible to define any type of parameter in a hierarchical manner, being able to assign the same value to a parameter as to another parameter.

CHAPTER 6

Experiments and Results

In this chapter, the experiments and the obtained results are presented. The experiments are aimed at obtaining some satisfactory results on the characteristics that an RL agent should meet in the video game industry. In particular, through experience and a survey of video game developers, it turns out that the main requirements are the following:

- The game has to entertain the user, agents can neither be too good at the game nor too weak: if it is too good at the game, then the human player cannot win any match; on the other hand, if the human player defeats RL agents too easily, then he/she might get bored and stop playing that video game.

- Since the models could take a long time to train, they should be able to *generalize* to unseen situations. So, the agent should be able to act optimally, or make good decisions even if encounters a situation he has never seen before. For instance, when Orobix S.r.l.[1] developed the first RL agent applied to a video game (A.N.N.A.[2] trained in the MotoGP video game), they had some problems in the generalization of the environment. A.N.N.A. is a PPO agent trained for the MotoGP19 video game.

  They had difficulty passing on what the agent had learned on a particular track (e.g., the Mugello track) and passing that knowledge to another complete another track (e.g.,

[1] https://orobix.com/

[2] https://datasciencemilan.medium.com/a-n-n-a-artificial-neural-network-agent-for-motogp-19-f554fbe9ca

Jerez). In particular, the agent trained on the Mugello track was not able to complete a lap on the Jerez track.

- Another use case could be automatic testing, video game developers have to develop tests for the game, consuming a lot of time and resources to write tests. RL can be exploited to train the agent on the video game and observe its behavior, understanding if there are some bugs or anomalous behaviors. For instance, during the development of A.N.N.A., they discovered that the agent gained a lot of speed abnormally when touching a curb on a bend, this enabled the video game developer to fix the physics of the game.
- To return to the theme of playability and the fact that a video game must be challenging for the human player, but at the same time not too complex, an RL algorithm can be used to understand whether or not the audience can appreciate it. For instance, by observing the actions the agent selects for completing the game.

Dreamer and P2E algorithms seem perfect to meet the requirements imposed by the video game industry, therefore, experiments were conducted to study how effectively RL can be used to date in the video game industry. Starting from the correctness of our implementations, through the ability of the world model to generalize the environment to situations never encountered before, to the realization of automatic tests for video games.

6.1 Correctness of Implementations

Given the complexity of the algorithms used, experiments were conducted to verify that, actually, our proposed implementations in PyTorch were correct and that the algorithms worked at least as well as the original versions in TensorFlow.

6.1.1 DreamerV1

Dreamer-V1 was trained on a continuous control task and a discrete control task. The first one was the *Walker Walk* environment from *Deep Mind Control*: it is a continuous control task,

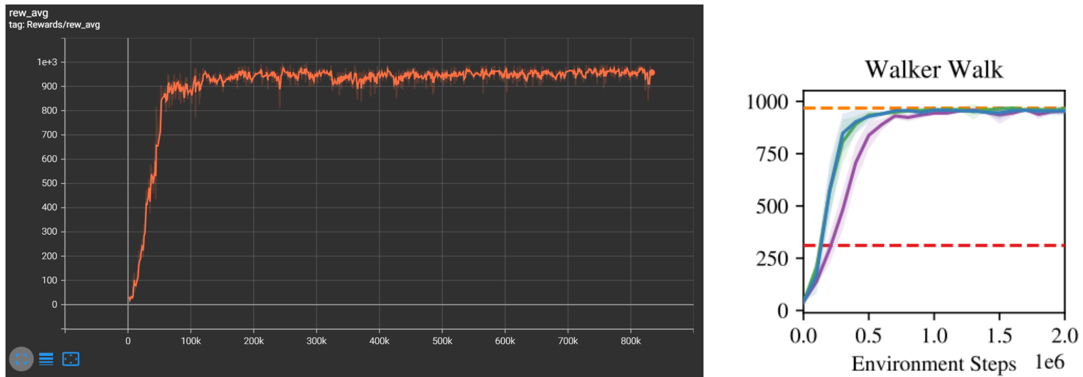


Figure 6.1 Walker Walk: on the left the result obtained by the implementation of Dreamer-V1 in SheepRL (PyTorch); instead, on the right, the result published in [24] (TensorFlow).

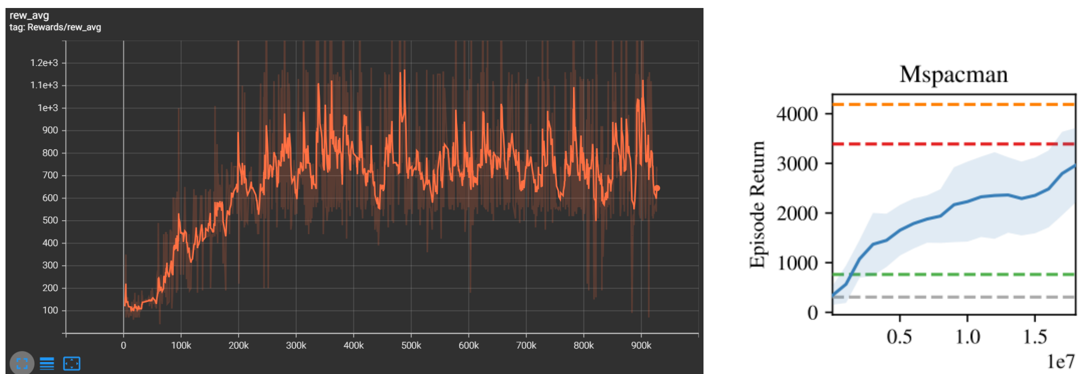


Figure 6.2 MsPacman: on the left the result obtained by the implementation of Dreamer-V1 in SheepRL (PyTorch); instead, on the right, the result published in [24] (TensorFlow). As one can notice, we performed many fewer steps with respect to the original work because of the limited availability of resources.

where the agent has to learn how to walk in the forward (right) direction by applying torques on the six hinges connecting the seven body parts, so there are six continuous actions that the agent has to select at each time step. The experiment was run with the same hyper-parameters declared in [24]. As one can notice in Figure 6.1, the results of our implementation and the official one are similar, meaning that the algorithm is correct.

A second confirmation comes from the second experiment we performed for Dreamer-V1, as shown in Figure 6.2. The environment is the Atari *MsPacman*, so, the agent has to eat the yellow dots on the map, and at the same time, it has to avoid ghosts. This experiment was trained for 1 million steps, instead of training for 20 million steps, as in the original work.

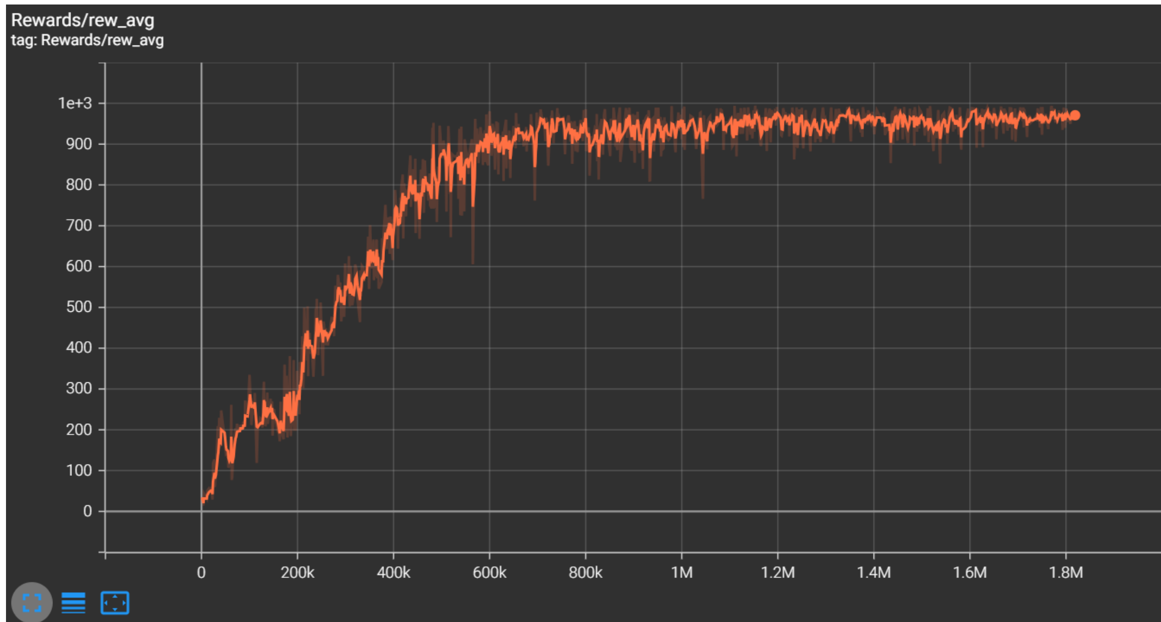


Figure 6.3 The rewards obtained by Dreamer-V2 in the Walker Walk environment.

6.1.2 Dreamer-V2

Dreamer-V2 was trained in two environments, one with continuous actions and the other with discrete ones. The continuous control environment is *Walker Walk* of DMC, whereas the discrete control one is *Pong* of Atari. Figure 6.3 shows the results in the *Walker Walk* environment and Figure 6.4 shows the ones in the *Pong* environment. In both experiments, the agent receives the maximum reward.

6.1.3 Dreamer-V3

To check the correctness of Dreamer-V3 we were able to rely on some references in the paper, in fact, in [28], have carried out many experiments on Atari environments, pulling the model for 100K steps with a modestly sized model, in Figure 6.5 and Figure 6.6 our results and the original ones are shown.

Moreover, Dreamer-V3 is trained in more challenging environments: the first one is *Crafter*. Also in this case our agent reaches the same amount of cumulative reward as the original work (Figure 6.7).

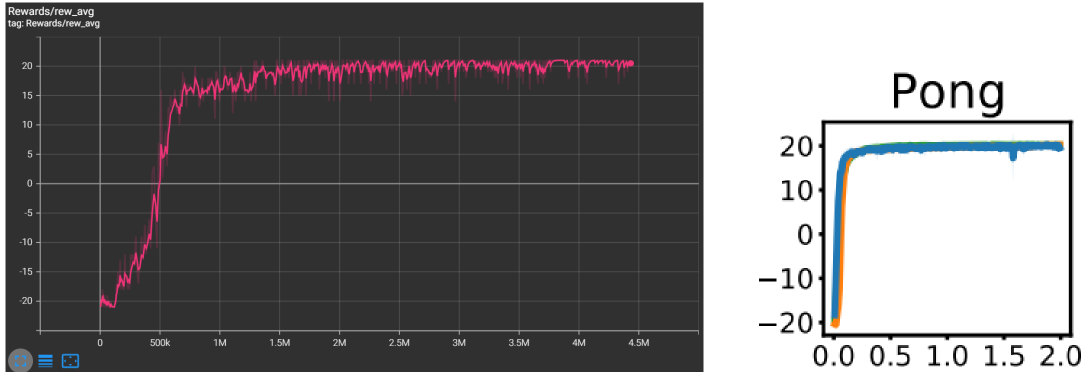


Figure 6.4 Pong: on the left the result obtained by the implementation of Dreamer-V2 in SheepRL (PyTorch); instead, on the right, the result published in [27] (TensorFlow). As one can notice, we performed many fewer steps with respect to the original work because of the limited availability of resources.

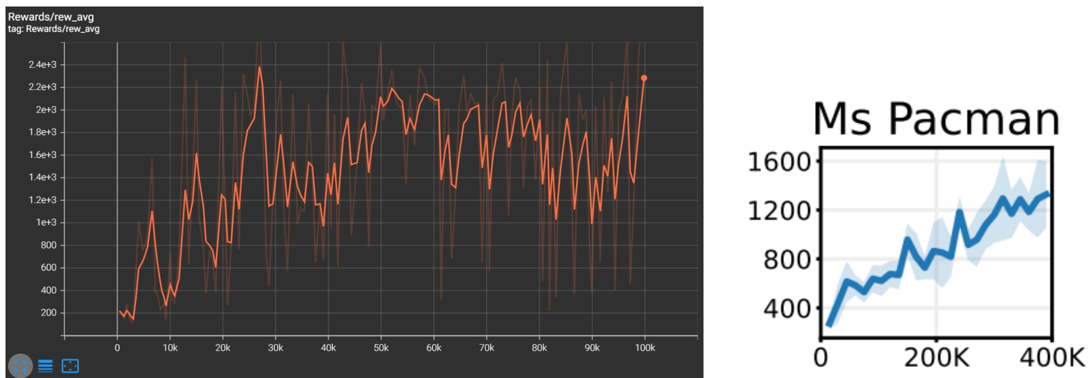


Figure 6.5 MsPacman: on the left the result obtained by the implementation of Dreamer-V3 in SheepRL (PyTorch); instead, on the right, the result published in [28] (TensorFlow). The graphs show a different number of steps, this is simply due to the measurement used on the x-axis: both experiments are performed with the same hyper-parameters and take the same number of steps. On the right, the steps correspond to the number of times the actor chooses an action (policy steps); while on the left, the steps correspond to the steps played in the environment. The values differ because of the action repeat parameter, set to 4; in fact, $400\text{K}/4 = 100\text{K}$ steps.

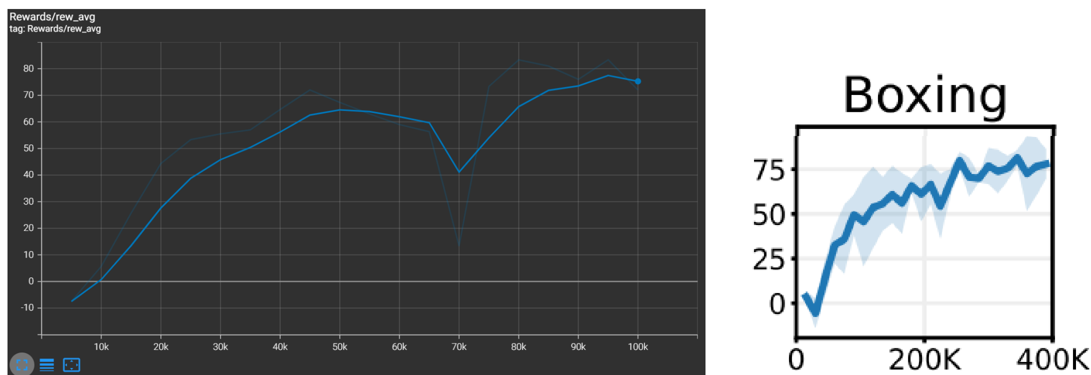


Figure 6.6 Boxing: on the left the result obtained by the implementation of Dreamer-V3 in SheepRL (PyTorch); instead, on the right, the result published in [28] (TensorFlow). The graphs show a different number of steps, this is simply due to the measurement used on the x-axis: both experiments are performed with the same hyper-parameters and take the same number of steps. On the right, the steps correspond to the number of times the actor chooses an action (*policy steps*); while on the left, the steps correspond to the steps played in the environment (*environment steps*). The values differ because of the action repeat parameter, set to 4; in fact, $400K/4 = 100K$ steps.

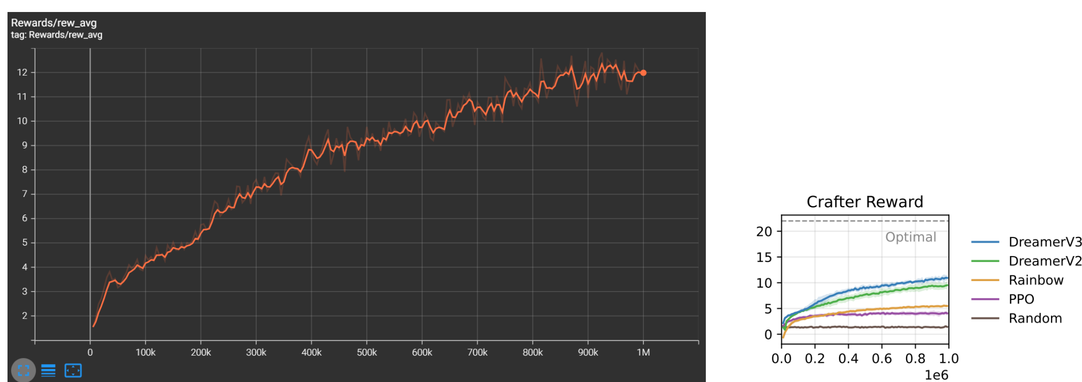


Figure 6.7 The results obtained by the SheepRL Dreamer-V3 (on the left) and the results obtained by its original implementation [28] (on the right) in the Crafter environment. The agent shows a strong ability to generalize (newly generated maps for each episode), to deal with partial observability (each input image reveals only a small part of the world), and to long-term reasoning and survival.

The other experiment is performed in the Navigate environment of MineRL, unfortunately, though, there are no other experiments done with Dreamer-v3 with which we can compare our results. Recall that the Navigate task corresponds to find a diamond located in the world by following a compass. The agent is given a positive reward when he approaches the diamond and a negative reward when he allots instead. Moreover, a reward of 100 is given when he collects the diamond. The agent starts always at 64 meters from the diamond, so, in general, when it gets a reward above 100, it means he has successfully completed the task, on the other hand, if he gets a reward between 50 and 100, it means he got near the diamond but did not find it. In addition to the difficulty of having a 3D environment, the diamond is not always in plain sight, often it is located underwater or underground, therefore, the agent must learn human-like behaviors to pick up the diamond in these situations. As shown in Figure 6.8, the agent is not always able to collect the diamond, even if, in general, it has learned to follow the information of the compass to get closer to the diamond. In particular, it sometimes happens that even in the simplest cases where the diamond is in plain sight, the agent has not learned to go toward the diamond to pick it up, preferring vector information (compass) with respect to visual information (the RGB frame).

6.2 Generalization

For the generalization, we were inspired by the case of MotoGP, but not having the MotoGP environment available, we fell back to a simpler environment, but one that has similar characteristics: CarRacing. Dreamer-V1 was trained on a single CarRacing track, always the same. It was then tested on other tracks, never previously seen, to see if it had actually learned the dynamics of the environment, or if it had simply learned only to complete the lap on the track on which it was trained. The same procedure was performed on the model-free SAC algorithm, trained and tested on the same tracks Dreamer was trained and tested on respectively. The results are satisfactory, even in a relatively simple environment like CarRacing, Dreamer is able to generalize much better compared to model-free algorithms such as SAC. In fact, both achieved a very high score on the track they were trained on.

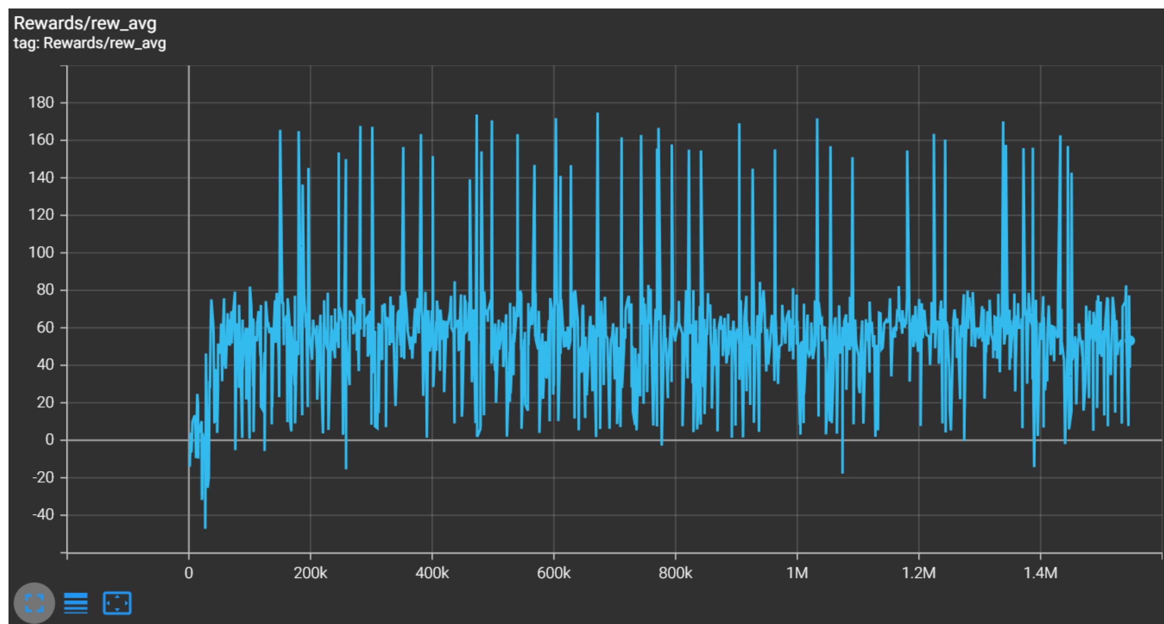


Figure 6.8 The reward obtained in the Navigate task. As it is possible to notice, Dreamer-V3 is able to follow the information of the compass, but it has some difficulties in collecting the diamond. Indeed, the reward shows that some episode ends with a cumulative reward greater than 100 (the reward has been collected) and other with a cumulative reward lower than 100. In most of the episodes, the agent receives a reward of (almost) 60, meaning that it has followed the information in the compass.

Table 6.1 The generalization capabilities of Dreamer-V1 and SAC on the CarRacing environment. The track number 42 is the one on which the two algorithms were trained. The other three tracks have never been seen by the two agents. From the rewards, it is evident that Dreamer is still able to complete the lap quickly, while SAC struggles more. Considering that this environment is simple, we expect this difference to increase as the difficulty level rises. The underlined scores refer to the maximum score achieved by an algorithm on that particular track.

Seed	**Dreamer-V1**	**SAC**
42	<u>936</u>	934
10	<u>893</u>	292
200	<u>854</u>	628
5000	<u>858</u>	637

However, Dreamer is capable of completing, albeit not perfectly, other tracks as well; whereas SAC encounters much more difficulty to complete them. Table 6.1 shows the differences between the two algorithms.

6.3 Automatic Testing

Finally, let us discuss the testing phase. Referring back to the experience with MotoGP, an experiment was conducted with P2E to verify whether a RL algorithm can be effectively used to explore the environment and uncover bugs or situations and/or locations where the player may become stuck. Exploration was carried out in the Open-Ended environment of MineDojo, which does not have a specific task. Consequently, the agent does not receive a reward, and it can perform any action. Hence, the natural choice was P2E, which defines the *intrinsic reward* as a measure of novelty. The exploration of the agent was then compared to that of a random agent, i.e., making purely random choices. The results, as shown in Figure 6.9, demonstrate that a RL algorithm can assist video game developers in testing their products effectively.

Furthermore, we conducted another experiment in which the actions of the agent were limited to movement, camera rotation, jumping, and attacks (to break objects). We discovered that there are certain locations on the map from which the agent cannot escape using only

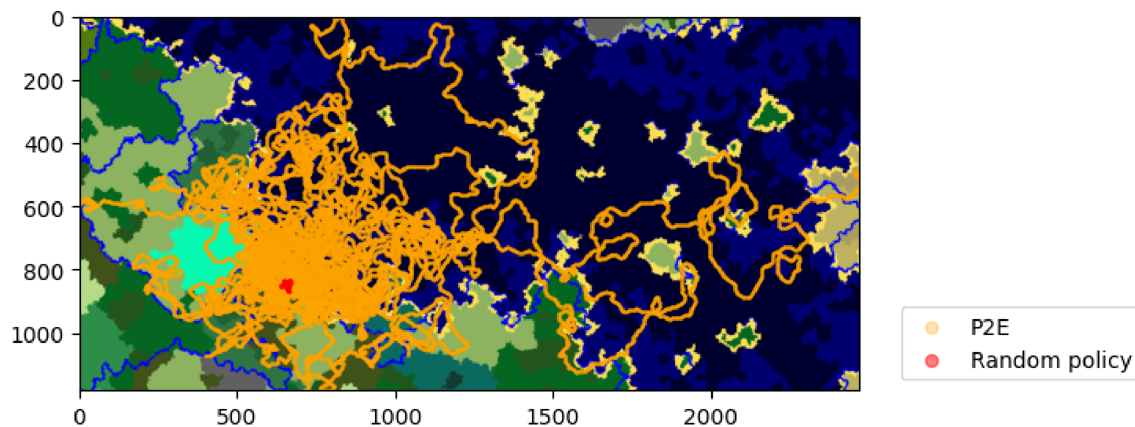


Figure 6.9 The portion of the Minecraft map explored by the P2E agent. The orange indicates the locations visited by P2E; whereas, the red part indicates the visited locations of the random agent.

the available actions. This once again illustrates how RL can be effectively utilized in video game testing.

An unintended testing scenario emerged during the training of Dreamer-V3 in the "Dead or Alive ++" environment provided by DIAMBRA. We initially began the training to assess the performance of the agent in this environment, but then we realized that RL could also be used to observe how enjoyable a game is for users. As depicted in Figure 6.10, Dreamer-V3 learned to complete the game very effectively: the maximum reward is approximately 3300, but the agent regularly defeats all opponents, thereby completing the game.

Since the agent can successfully complete the game, we observed the actions it uses to defeat opponents and realized that it consistently employs the same two or three actions. This could serve as a warning sign for developers, as the game may become repetitive and less engaging for human players.

Although DIAMBRA is a fully described environment, it harbors difficulties: for one thing, different matches in the same episode have different backgrounds; moreover, there are many different characters. These two aspects require the agent to be able to generalize (be able to solve the task even in situations never seen before or encountered a few times) and increase the difficulty of training.

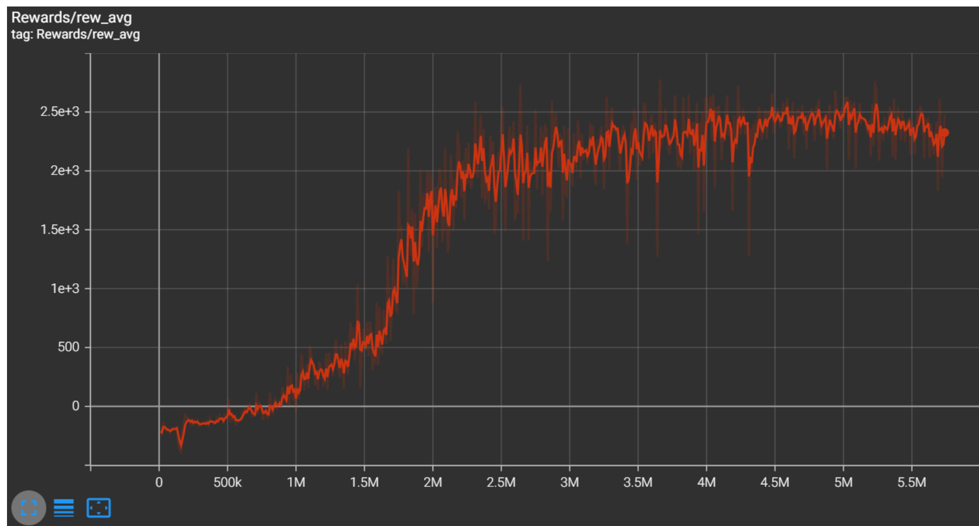


Figure 6.10 The reward obtained by Dreamer-V3 in the “Dead or Alive ++” environment.

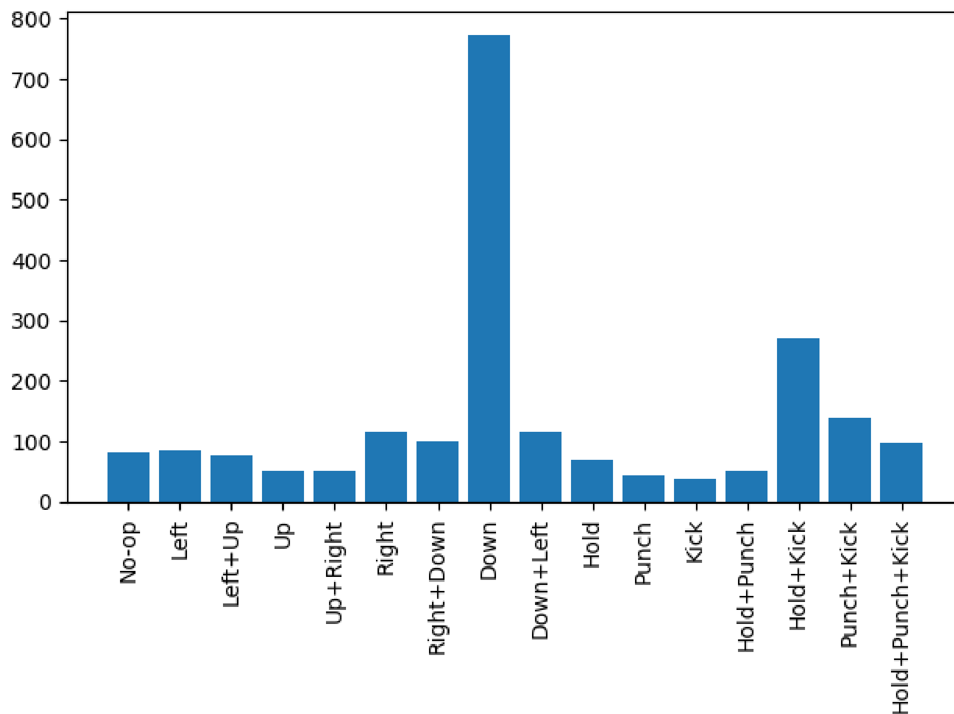


Figure 6.11 The actions distribution of Dreamer-V3 for completing the “Dead or Alive ++” game. As one can see, it is possible to win the game by repeatedly executing the same actions.

CHAPTER 7

Conclusions

In summation, this research has shed light on the intricate relationship between RL and the video game industry. While RL exhibits immense promise, it also faces notable challenges when tasked with emulating human-like behaviors, as exemplified by the complexities inherent in conquering games such as the Navigate environment in MineRL. This realization underscores that RL, in its present form, outperforms humans in many tasks, but still has difficulty replicating some high-level behaviors.

Nonetheless, the true strength of RL within the video game industry manifests in its capacity for robust testing and evaluation. RL algorithms prove themselves to be useful in discovering latent bugs, evaluating game difficulty levels, and discerning whether a game achieves the delicate equilibrium between challenge and accessibility. These tools can empower video game developers to iteratively refine their creations, ultimately fostering the evolution of more engaging and captivating gaming experiences.

Furthermore, the implementation of advanced RL algorithms presents an avenue to generalize video game environments, potentially reducing the arduous training time required for NPCs. However, the integration of RL into NPC creation does not come without its own set of challenges. Its resource-intensive nature and the proclivity for NPCs to outmatch human player capabilities necessitate developers to commit additional effort towards manually curbing the ability of NPCs.

7.1 Next Steps

Regarding the automatic creation of NPCs through the RL, two paths can be pursued in the future. The first is to use constraint programming applied to the RL [9] to be able to limit the skills of an agent or to force it to behave in a certain way, depending on the needs of the game. Alternatively, it is possible to introduce in SheepRL the possibility of Imitation Learning [12], i.e., letting the agent not learn from scratch by collecting episodes from the environment, but learn from a dataset filled with games played by human players or other agents, thus losing the online nature of the RL.

Another interesting development to be made to SheepRL is the addition of the Curious Replay Buffer proposed by Kauvar et al. [33]: it is a kind of prioritized experience replay tailored to model-based agents through the use of a curiosity-based priority signal. Agents using Curious Replay exhibit improved performance in an exploration paradigm inspired by animal behavior and on the Crafter benchmark. In particular, they noticed that P2E can fail in some situations when curiosity is needed: they trained a P2E agent in an empty environment, after some steps, they put an object in the environment and they observed that P2E does not interact much with the new object, as an animal would. With the Curious Replay Buffer, instead, the number of interactions with the new object is many more.

Since Dreamer-v3 had difficulty solving the Navigate task in Minecraft because it was having trouble associating the diamond with the very high reward he received when he collected it, the Curious Replay Buffer could help Dreamer-V3 associate the diamond with the reward and, thus, solve the problem.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems.

[2] Agarap, A. F. (2018). Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375.

[3] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.

[4] Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887.

[5] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2012). The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708.

[6] Benamou, J.-D., Carlier, G., Cuturi, M., Nenna, L., and Peyré, G. (2014). Iterative bregman projections for regularized transportation problems.

[7] Bengio, Y., Léonard, N., and Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation.

[8] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

[9] Cappart, Q., Moisan, T., Rousseau, L., Prémont-Schwarz, I., and Ciré, A. A. (2020). Combining reinforcement learning and constraint programming for combinatorial optimization. *CoRR*, abs/2006.01610.

[10] Chen, X., Wang, C., Zhou, Z., and Ross, K. W. (2021). Randomized ensembled double q-learning: Learning fast without a model. *CoRR*, abs/2101.05982.

[11] Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259.

[12] Ciosek, K. (2022). Imitation learning by reinforcement learning.

[13] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2016). Fast and accurate deep network learning by exponential linear units (elus).

[14] Elfwing, S., Uchibe, E., and Doya, K. (2017). Sigmoid-weighted linear units for neural network function approximation in reinforcement learning.

[15] Falcon, W., Borovec, J., Wälchli, A., Eggert, N., Schock, J., Jordan, J., Skafte, N., Ir1dXD, Bereznyuk, V., Harris, E., Murrell, T., Yu, P., Præsius, S., Addair, T., Zhong, J., Lipin, D., Uchida, S., Bapat, S., Schröter, H., Dayma, B., Karnachev, A., Kulkarni, A., Komatsu, S., Martin.B, SCHIRATTI, J.-B., Mary, H., Byrne, D., Eyzaguirre, C., cinjon, and Bakhtin, A. (2019). Pytorch lightning.

[16] Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. (2022). Minedojo: Building open-ended embodied agents with internet-scale knowledge.

[17] Farquhar, G., Rocktäschel, T., Igl, M., and Whiteson, S. (2017). Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. *CoRR*, abs/1710.11417.

[18] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2017). Noisy networks for exploration. *CoRR*, abs/1706.10295.

[19] Guss, W. H., Houghton, B., Topin, N., Wang, P., Codel, C., Veloso, M., and Salakhutdinov, R. (2019). Minerl: A large-scale dataset of minecraft demonstrations. *CoRR*, abs/1907.13440.

[20] Ha, D. and Schmidhuber, J. (2018). Recurrent world models facilitate policy evolution. *CoRR*, abs/1809.01999.

[21] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290.

[22] Hafner, D. (2021). Benchmarking the spectrum of agent capabilities. *CoRR*, abs/2109.06780.

[23] Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. (2022). Mastering atari with discrete world models.

[24] Hafner, D., Lillicrap, T. P., Ba, J., and Norouzi, M. (2019). Dream to control: Learning behaviors by latent imagination. *CoRR*, abs/1912.01603.

[25] Hafner, D., Lillicrap, T. P., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. (2018a). Learning latent dynamics for planning from pixels. *CoRR*, abs/1811.04551.

[26] Hafner, D., Lillicrap, T. P., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. (2018b). Learning latent dynamics for planning from pixels. *CoRR*, abs/1811.04551.

[27] Hafner, D., Lillicrap, T. P., Norouzi, M., and Ba, J. (2020). Mastering atari with discrete world models. *CoRR*, abs/2010.02193.

[28] Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. (2023). Mastering diverse domains through world models.

[29] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298.

[30] Hiraoka, T., Imagawa, T., Hashimoto, T., Onishi, T., and Tsuruoka, Y. (2021). Dropout q-functions for doubly efficient reinforcement learning. *CoRR*, abs/2110.02034.

[31] Huang, S., Dossa, R. F. J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., and Araújo, J. G. (2022). Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18.

[32] Kaufmann, E., Bauersfeld, L., Loquercio, A., Müller, M., Koltun, V., and Scaramuzza, D. (2023). Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987.

[33] Kauvar, I., Doyle, C., Zhou, L., and Haber, N. (2023). Curious replay for model-based adaptation.

[34] Kingma, D. P. and Welling, M. (2022). Auto-encoding variational bayes.

[35] Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Gonzalez, J., Goldberg, K., and Stoica, I. (2017). Ray rllib: A composable and scalable reinforcement learning library. *CoRR*, abs/1712.09381.

[36] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.

[37] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. (2017). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR*, abs/1709.06009.

[38] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

[39] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[40] Moon, S., Yeom, J., Park, B., and Song, H. O. (2023). Discovering hierarchical achievements in reinforcement learning via contrastive learning.

[41] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Paul, W., Jordan, M. I., and Stoica, I. (2017). Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889.

[42] Nardelli, N., Synnaeve, G., Lin, Z., Kohli, P., Torr, P. H. S., and Usunier, N. (2018). Value propagation networks. *CoRR*, abs/1805.11199.

[43] Palmas, A. (2022). Diambra arena: a new reinforcement learning platform for research and experimentation.

[44] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703.

[45] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

[46] Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models.

[47] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay.

[48] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. P., and Silver, D. (2019). Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265.

[49] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization. *CoRR*, abs/1502.05477.

[50] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

[51] Sekar, R., Rybkin, O., Daniilidis, K., Abbeel, P., Hafner, D., and Pathak, D. (2020). Planning to explore via self-supervised world models. *CoRR*, abs/2005.05960.

[52] Silver, D. (2015). Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/.

[53] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017a). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[54] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017b). Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.

[55] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

[56] Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., de Las Casas, D., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T. P., and Riedmiller, M. A. (2018). Deepmind control suite. *CoRR*, abs/1801.00690.

[57] Tassa, Y., Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T. P., and Heess, N. (2020). dm_control: Software and tasks for continuous control. *CoRR*, abs/2006.12983.

[58] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033.

[59] Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., de Cola, G., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., and Younis, O. G. (2023). Gymnasium.

[60] van den Oord, A., Li, Y., and Vinyals, O. (2018). Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748.

[61] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461.

[62] Vezhnevets, A., Mnih, V., Agapiou, J. P., Osindero, S., Graves, A., Vinyals, O., and Kavukcuoglu, K. (2016). Strategic attentive writer for learning macro-actions. *CoRR*, abs/1606.04695.

[63] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1995–2003. JMLR.org.

[64] Yadan, O. (2019). Hydra - a framework for elegantly configuring complex applications. Github.

[65] Yarats, D., Zhang, A., Kostrikov, I., Amos, B., Pineau, J., and Fergus, R. (2019). Improving sample efficiency in model-free reinforcement learning from images. *CoRR*, abs/1910.01741.

[66] Zheng, H., Yang, Z., Liu, W., Liang, J., and Li, Y. (2015). Improving deep neural networks using softplus units. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–4.