



Institut für Systems Engineering
Fachgebiet Echtzeitsysteme



Leibniz
Universität
Hannover

Comparison of Different State-of-the-Art Reinforcement Learning Methods in a 2D Simulation Environment

Bachelorarbeit

Osama Salem

Matrikelnummer: 10027030

Erstprüfer: Prof. Dr.-Ing. Bernardo Wagner

Betreuer: M.Sc. Julian Kunze

April 12, 2023

Osama Salem

Matrikelnummer: 10027030

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Hannover, den 12. April 2023

Aufgabenstellung zur Bachelorarbeit:

Vergleich verschiedener State-of-the-Art Verfahren des Reinforcement Learning in einer 2D Simulationsumgebung

Bearbeitet von: Osama Salem
Studiengang: Elektrotechnik und Informationstechnik

Matrikel-Nr.: 10027030
Betreut von: Julian Kunze, M.Sc.

Im Rahmen des BMBF-Projekts *UAV-Rescue* soll untersucht werden, inwiefern die Suche nach Vermissten in Gebäuden durch den Einsatz von unbemannten Luftfahrzeugen (engl. *Unmanned Aerial Vehicle*, kurz *UAV*) erleichtert werden kann. In einem Teilauftrag sollen dabei übergeordnete Such- und Erkundungsstrategien mithilfe neuartiger Verfahren des *Reinforcement Learning* (RL) erlernt werden. Um diese Lernverfahren effizient entwickeln zu können, wurde eine leistungsfähige 2D Simulationsumgebung geschaffen, die eine schnelle Iteration über verschiedene Ansätze ermöglicht.

Im Rahmen dieser Bachelorarbeit sollen RL Algorithmen, die den gegenwärtigen Stand der Technik widerspiegeln, in der Simulationsumgebung getestet und miteinander verglichen werden. Hierfür soll zunächst recherchiert werden, welche Ansätze des RL als State-of-the-Art infrage kommen. Es sollen dabei sowohl *model-free* als auch *model-based* Ansätze betrachtet werden. Anschließend sollen bestehende RL Bibliotheken und Open-Source Implementierungen dieser Algorithmen auf ihre Eignung für den Einsatz in der Simulation geprüft werden. Die vielversprechendsten Ansätze sollen dann auf die Simulationsumgebung adaptiert und anschließend trainiert werden. In einer umfangreichen und nach gängigen Standards durchgeführten Evaluation sollen die einzelnen Ansätze abschließend sowohl untereinander als auch mit menschlichen Ergebnissen verglichen werden. Optional können darüber hinaus kleine Variationen der bestehenden Ansätze implementiert werden, um möglicherweise eine stärkere Baseline für zukünftige Experimente zu erhalten.

Die Arbeit umfasst folgende Arbeitsschritte:

- Literaturrecherche zu State-of-the-Art RL Algorithmen
- Vergleich verschiedener RL Bibliotheken und weiteren Open-Source Implementierungen
- Adaption der Verfahren auf die Simulationsumgebung
- Evaluation der RL Verfahren untereinander sowie mit menschlichen Ergebnissen
- *Optional*: Implementierung kleiner Variationen der bestehenden Ansätze

Hannover, 26. September 2022



Prof. Dr.-Ing. Bernardo Wagner

Contents

Figures	VII
Tables	VIII
Symbols	X
1 Introduction	1
2 Foundations	2
2.1 Definition	2
2.2 Reinforcement Learning, Supervised Learning and Unsupervised Learning	2
2.3 Training Reinforcement Learning Algorithms	3
2.4 Reinforcement Learning Components	6
2.4.1 Policy	6
2.4.2 Reward Function	7
2.4.3 Transition Model	7
2.4.4 Q-Function and Value Function	8
3 Model-Free Reinforcement Learning	10
3.1 Policy-Based Methods	10
3.1.1 REINFORCE	10
3.1.2 Advantage Actor-Critic (A2C)	13
3.1.3 Proximal Policy Optimization (PPO)	16
3.2 Value-Based Methods	19
3.2.1 Deep Q Network (DQN)	19
3.2.2 Rainbow	22
3.2.3 Soft Actor-Critic (SAC)	27
4 Model-Based Reinforcement Learning	29
4.1 Dreamer	29

5	Exploration	33
5.1	Intrinsic Curiosity Module (ICM)	33
5.2	Random Network Distillation (RND)	35
6	Implementation Details	37
6.1	Search And Rescue Environment	37
6.1.1	Wrappers	37
6.1.2	Issues Fixed	39
6.1.3	Additional Features	41
6.2	Baselines	43
6.2.1	Algorithms	44
6.2.2	Buffers	45
6.2.3	Data Preprocessors	45
6.2.4	Recurrence	47
6.2.5	Transfer Learning	49
7	Experiments	50
7.1	Algorithms	50
7.1.1	PPO + RND	51
7.1.2	A2C + RND	51
7.1.3	A2C + Transfer Learning	52
7.1.4	DQN + RND	52
7.1.5	SAC + RND	53
7.1.6	Rainbow	55
7.1.7	DreamerV3	56
7.1.8	Human Operator	56
7.2	Modifications	57
7.3	Hyperparameters	60
8	Discussion	61
9	Conclusion and Outlook	67
	Bibliography	74

List of Figures

2.1	Reinforcement Learning Agent's Learning Loop	3
2.2	Supervised Learning vs Unsupervised Learning	4
2.3	Training Reinforcement Learning Algorithms Paradigm	4
2.4	Reinforcement Learning Basic Components	6
2.5	Reinforcement Learning Policy Approximated By A Deep Neural Network	7
2.6	Markovian Transition Model	8
3.1	PPO Surrogate Advantage Clipping	18
3.2	Difference Between Standard And Dueling DQN.	25
4.1	Dreamer's World Model And The Actor-Critic Learner	30
5.1	Intrinsic Curiosity Module Network	34
5.2	Random Network Distillation	35
6.1	Overview Of The SAR Environment	38
6.2	Point Cloud Of The SAR Laser Readings	38
6.3	Agent Spawning On Top Of A Wall	40
6.4	Agent's Laser Range Issue.	40
6.5	Decrease of the episodic length of a solved environment	41
6.6	Terminal State Illustration	42
6.7	Processing SAR Observations By Flattening Raster Data.	45
6.8	Overview Of 2DLaserNet.	46
6.9	Channel And Spatial Gates Of CBAM.	47
6.10	Recurrent Observation Pre-Processor.	48
7.1	Trajectories Of PPO And RND Agents.	51
7.2	Trajectories Of A2C And RND Agents.	52
7.3	Trajectories Of Fine-Tuned A2C Agents.	53
7.4	Trajectories Of DQN and RND Agents.	54
7.5	Trajectories Of SAC and RND Agents.	55

7.6	Trajectories Of Rainbow Agents.	56
7.7	Trajectories Of Dreamer Agents.	57
7.8	Trajectories By A Human Operator.	58
7.9	Modifications Experiment Average Reward Chart.	58
7.10	Trajectories of PPO agents with different policy implementations and observations. .	59
7.11	Rewarding Charts Of The Hyperparameter Experiment.	60
8.1	PPO And A2C Policy Loss.	61
8.2	Entropy Chart Of Policy Gradient Methods	62
8.3	Q Values Training Chart	63
8.4	Intrinsic Rewarding Chart Of RND	63
8.5	Intrinsic Rewarding Chart Of ICM	64
8.6	Rainbow Episodic Length Training Chart	64
8.7	Noisy Laser Readings	66

List of Tables

7.1	Comparison of RL different approaches performance metrics.	50
7.2	Comparison of different PPO policy implementations performance metrics.	60

Symbols

Acronyms

Acronym	Meaning
<i>UAV</i>	unmanned aerial vehicle
<i>RL</i>	reinforcement learning
<i>AI</i>	artificial intelligence
<i>ML</i>	machine learning
<i>A2C</i>	advantage actor-critic
<i>MDP</i>	markovian decision process
<i>TD</i>	temporal difference
<i>GAE</i>	generalized advantage estimation
<i>PPO</i>	proximal policy optimization
<i>TRPO</i>	trust region policy optimization
<i>KL</i>	kullback leibler
<i>DQN</i>	deep q network
<i>SAC</i>	soft actor-critic
<i>RSSM</i>	recurrent state-space model
<i>ICM</i>	intrinsic curiosity module
<i>RND</i>	random network distillation
<i>SAR</i>	search and rescue
<i>IMU</i>	internal measurement unit
<i>GPU</i>	graphics processing unit
<i>CPU</i>	central processing unit
<i>DPI</i>	dots per inch
<i>IQN</i>	implicit quantile network
<i>EMA</i>	exponential moving average
<i>MLP</i>	multi-layer perceptron
<i>CBAM</i>	convolutional block attention module
<i>CNN</i>	convolutional neural network
<i>POMDP</i>	partially observed markovian decision process
<i>GRU</i>	gated recurrent unit
<i>LSTM</i>	long short-term memory
<i>CQL</i>	conservative q-learning

1 Introduction

Catastrophic occurrences are often accompanied by substantial harm to infrastructure, in which individuals may be buried or ensnared. To locate and recover these missing individuals promptly, without putting their lives at risk, search and rescue teams require a comprehensive overview of the damaged and imminent-to-collapse sites. This task can be facilitated through the deployment of an unmanned aerial vehicle (UAV) to capture a full indoor map, as well as detect potential vital signs.

The UAV system can be empowered with advanced search and reconnaissance strategies acquired through innovative reinforcement learning (RL) methods. The exponential growth of communication and computation technologies in recent years has enabled access to vast amounts of data and parallel computing hardware, thereby facilitating the training and construction of large-scale artificial intelligence (AI) models. To optimize the development of these learning methods, a potent 2D simulation environment has been created, which allows for a quick iteration of different approaches. In the context of this bachelor's thesis, RL algorithms that reflect the current state-of-the-art will be evaluated and compared within the simulation environment.

Initially, in chapter 2, we will explore the basic concepts and foundations of RL. Then, in chapters 3, 4, and 5, we will conduct research to identify RL approaches considered state-of-the-art, including both model-free and model-based approaches, as well as exploration methods. Thereafter, in chapter 6, existing RL libraries and open-source implementations of these algorithms will be scrutinized for suitability in simulation. The most promising methods will then be adapted and even extended with recurrent nets, exploration methods, and observations processors, allowing the agent to overcome many downsides associated with the environment's nature. Lastly, the individual approaches will be subjected to an evaluation in chapter 7, and compared against each other and human results.

2 Foundations

RL has undergone a noteworthy progression due to advancements in computational technology in recent years. RL possesses a broad spectrum of practical applications, including the training of computers to play games such as chess [1], Go [2], and even Dota [3] at a proficient level. Furthermore, RL finds utility in numerous fields such as finance, healthcare, and natural language processing [4]. However, the utilization of RL in robotics [5] is particularly intriguing, where robots can be educated to execute tasks such as object grasping, ambulation and traversal through various environments. By instructing unmanned aerial vehicles to navigate treacherous settings, RL can reduce the weight of responsibilities on rescue departments who might otherwise endanger their own safety when trying to locate individuals in perilous situations, such as in burning buildings, collapsing mines, and burning forests.

2.1 Definition

RL pertains to the study of optimizing a numerical reward through the acquisition of knowledge regarding the most efficacious actions to take in a given situation [6]. Essentially, they are closed-loop problems, as the actions of the learning system influence its future input. Furthermore, in contrast to many forms of machine learning (ML) in which the learner is instructed as to which actions to take, the learner in RL must determine which actions yield the most reward through experimentation. In the most intriguing and demanding cases, actions may not only impact the immediate reward, but also the subsequent situation and all rewards that follow [7, chapter 1.1].

2.2 Reinforcement Learning, Supervised Learning and Unsupervised Learning

RL diverges from supervised learning, a common focus in the realm of ML, which involves being provided with a set of labeled examples by an informed external supervisor [9]. These examples consist of descriptions of various situations accompanied by the appropriate action for the system to

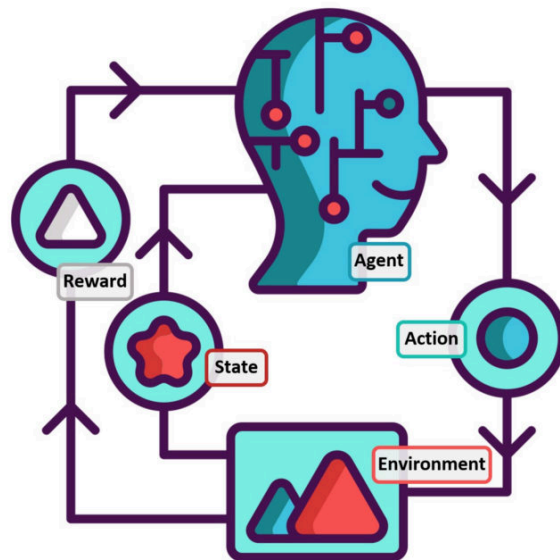


Figure 2.1: RL agents learn by collecting experience from an environment. [8]

take in response as shown in figure 2.2 (right), often for the purpose of classifying the situation. The goal of supervised learning is for the system to generalize its responses to correctly handle situations not included in the training set. While this type of learning is crucial, it is insufficient for learning through interaction. In interactive scenarios, it may be infeasible to obtain examples of the desired behavior that are both correct and representative of all the circumstances in which the agent must act. Therefore, in unexplored territory an agent must be able to learn from its own experiences.

Contrary to unsupervised learning, an ML approach focused on identifying patterns within collections of unlabeled data [10] as shown in figure 2.2 (left), RL by default does not completely rely on existing examples of proper behavior and instead endeavors to optimize a reward signal [11]. While the terminology of supervised and unsupervised learning seems to thoroughly categorize ML methodologies, this is not the case. In fact, the utilization of RL, which may be perceived as a type of unsupervised learning due to its lack of reliance on examples of correct behavior, to maximize reward rather than discover hidden structure distinguishes it as a separate ML paradigm. It is important to note that although the revelation of structure within an agent's experience can be beneficial in RL, it does not solve the issue of maximizing reward faced by RL agents.

2.3 Training Reinforcement Learning Algorithms

The process of training an RL algorithms typically comprises three principal components, as depicted in figure 2.3.

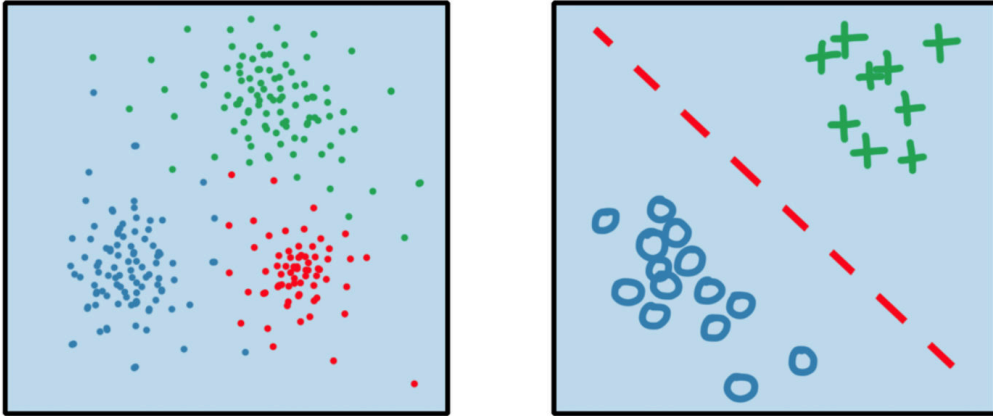


Figure 2.2: Unsupervised learning algorithms identify patterns within unlabeled data (left), while supervised learning algorithms categorize data based on predefined labels (right). [12]

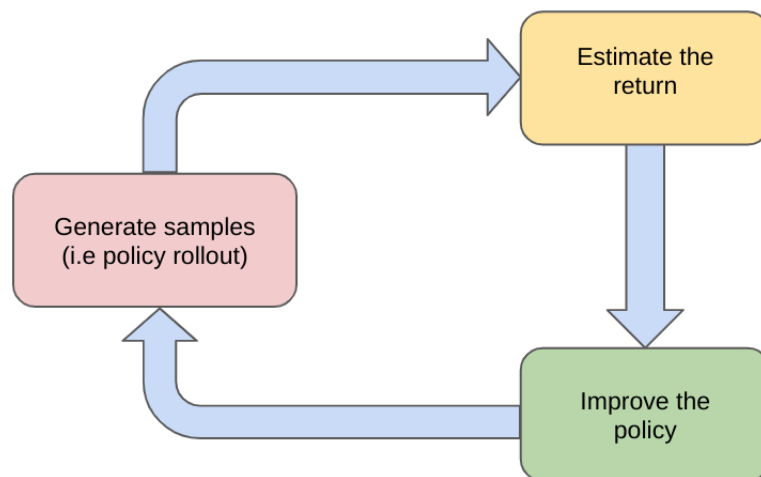


Figure 2.3: RL algorithms undergo a training process that involves the accumulation of experience, followed by the computation of the estimated return and ultimately the refinement of the policy. [13, p. 18]

The initial phase involves the production of samples, which may be obtained through the execution of the policy initially configured to execute random actions during the training process. Subsequently, the return associated with each sample is evaluated, which in basic cases involves calculating the mean reward provided to the agent by the environment for each trajectory undertaken by the agent [6], as illustrated in equation 2.1.

$$J(\theta) = E_{\pi} \left[\sum_t r_t \right] \approx \frac{1}{N} \sum_{i=1}^N \sum_t r_t^i \quad (2.1)$$

Here, $J(\theta)$ denotes the objective function, which is a measure of the performance of the algorithm. The function is dependent on the parameters θ of the policy π . The expectation operator E_{π} computes the expected value of the sum of rewards under the policy, where r_t represents the reward obtained at time step t .

The procedure for this step may vary depending on the specific algorithm being utilized. For example, in actor-critic algorithms, this step involves training a neural network (i.e., the critic) to approximate the goodness of actions induced by the policy [14]. Alternatively, other model-based methods may involve training a function f with parameters ϕ (e.g. a neural network) to approximate the transition dynamics of the model in order to inform decision-making [15], such that:

$$s_{t+1} = f_{\phi}(s_t, a_t) \quad (2.2)$$

where, s_t and a_t are state and action at timestep t , respectively. The final step involves improving the policy based on the feedback provided by the critic, which can be accomplished through a method such as gradient ascent. This process can be represented mathematically as follows:

$$\theta = \theta + \alpha \nabla_{\theta} J(\theta) \quad (2.3)$$

where α is learning rate. However, this step may also differ in model-based algorithms, such that the policy may be trained through backpropagation with respect to both the model f_{ϕ} and the learned reward function r_{ϕ} .

The duration of the three primary steps may vary significantly. Collecting data using a physical robot or vehicle may require several months, while utilizing simulators such as MuJoCo [16], OpenAI Gym [17], or Megastep [18] may be significantly more efficient, with a potential speed-up factor of up to thousands. Training duration of the actor, critic and potentially the environment model in the subsequent steps may also depend on the dimensionality of the data; for example, high-dimensional data such as images may necessitate longer training times compared to lower-dimensional data [19].

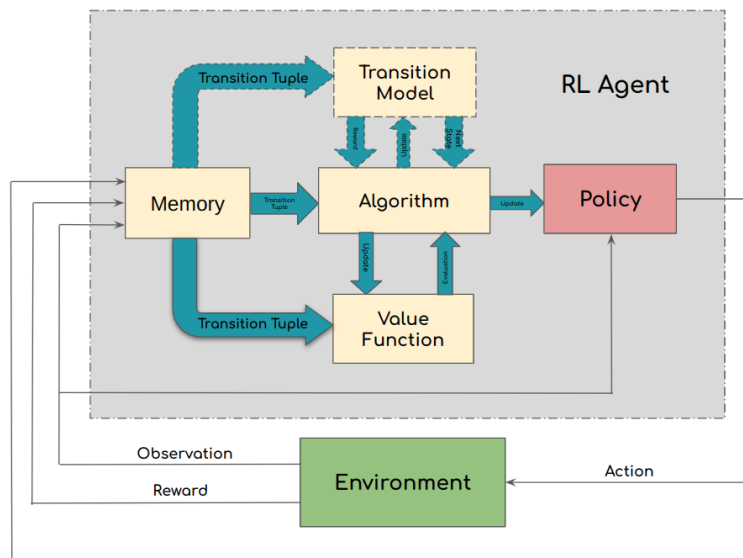


Figure 2.4: Figure shows basic components of a reinforcement learning algorithm and interaction between the different components.

2.4 Reinforcement Learning Components

Prior to delving into the explanation and assessment of various RL algorithms, it would be beneficial to establish some fundamental concepts that may be pertinent in the ensuing discussion. In an RL system, in addition to the agent and the environment, there are three primary components: a policy, a reward signal, and optionally, a model of the environment [7, chapter 1.3] as shown in figure 2.4.

2.4.1 Policy

The operational strategy of a learning agent is referred to as a policy π_θ as shown in figure 2.5. This policy can be viewed as a mapping between the perceived environmental states and the actions taken in response. The policy can take various forms, ranging from a simple function or table to more complex methods such as search or learning a universal approximation model (e.g. deep neural network). The policy is an essential component of an RL agent as it is solely responsible for determining the behavior of the agent. Although policies can exhibit randomness during initialization, they can be transferred between agents using transfer learning [11] techniques, which can enhance the agent's performance during learning.

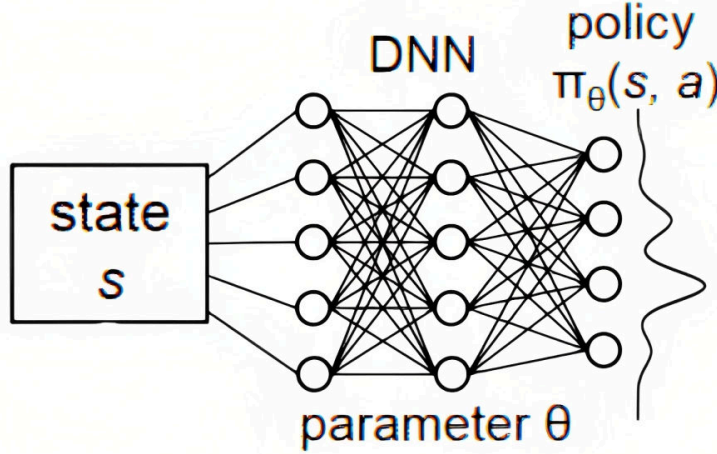


Figure 2.5: A policy takes state as an input and produces a distribution over possible actions. [20, p. 51]

2.4.2 Reward Function

The reward function $r(s, a)$ in an RL scenario designates the target for the agent. At each time interval, the environment sends the agent a single numerical value, the reward, and the agent's singular aim is to maximize the total reward received over an extended period. The reward signal thereby defines the positive and negative events for the agent. The reward given to the agent at any time is determined by the agent's current action and the current state of the environment and cannot be altered by the agent [7, chapter 3.2]. The only way the agent can impact the reward signal is through its actions, which may have a direct or indirect effect on the reward through changes in the environment's state. The reward signal serves as the primary factor for modifying the policy, with low reward potentially leading to changes in the policy to select alternative actions in similar situations in the future. Additionally, reward signals may exhibit randomness as a function of the state of the environment and the actions taken (e.g. densely rewarded stochastic environment).

2.4.3 Transition Model

In certain RL systems, a transition model $p(s_{t+1}|s_t, a_t)$ – depicted as a Markovian chain [7, chapter 3.5] in figure 2.6 – of the environment is also included. This component is capable of replicating the behavior of the environment or making predictions about how the environment will behave in the future. For example, given a state and action, the model may predict the resulting subsequent state and reward [15]. Models are utilized for the purpose of planning, which refers to the act of determining an appropriate course of action by considering potential future scenarios prior to their actual occurrence. Methods that utilize both models and planning to solve RL problems are

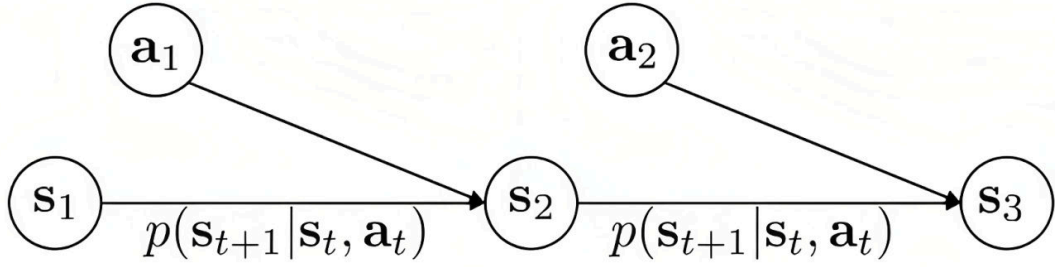


Figure 2.6: Transition to next state is conditioned by current state and current action. [13, p. 9]

referred to as model-based methods, in contrast to model-free methods that only utilize trial and error learning, which can be considered the antithesis of planning.

2.4.4 Q-Function and Value Function

Recalling the goal of RL algorithms as mentioned in section 2.3 is to maximize the expected value of the reward under some trajectory τ which can be expressed similarly to equation 2.1 as:

$$E_{\tau \sim p_{\pi}(\tau)} \left[\sum_t r(s_t, a_t) \right] \quad (2.4)$$

where p_{π} is the probability of taking actions induced by the policy π to traverse the trajectory τ . The probability of such trajectory can be re-expressed as a chain of the probability of states and actions by utilizing the Markovian property of the environment as follows:

$$p_{\pi}(\tau) = p_{\pi}(s_1, a_1, \dots, s_t, a_t) = p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t | s_t) p(s_{t+1} | s_t, a_t) \quad (2.5)$$

The concept of expectation may then be leveraged to generate a recursive representation of the trajectory distribution. Particularly, the expected value relative to this trajectory distribution can be formulated as a nested series of expectations, with the outermost expectation encompassing the overall expected value. As such, the aforementioned equation can be reexpressed as follows:

$$E_{s_1 \sim p(s_1)} \left[E_{a_1 \sim \pi(a_1 | s_1)} \left[r(s_1, a_1) + E_{s_2 \sim p(s_2 | s_1, a_1)} \left[E_{a_2 \sim \pi(a_2 | s_2)} [r(s_2, a_2) + \dots | s_2] | s_1, a_1 \right] \right] \right] \quad (2.6)$$

By substituting the inner part of the equation 2.6 with $Q(s_1, a_1)$ such that:

$$Q(s_1, a_1) = r(s_1, a_1) + E_{s_2 \sim p(s_2 | s_1, a_1)} \left[E_{a_2 \sim \pi(a_2 | s_2)} [r(s_2, a_2) + \dots | s_2] | s_1, a_1 \right] \quad (2.7)$$

the RL objective can be expressed as follows:

$$E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right] = E_{s_1 \sim p(s_1)} \left[E_{a_1 \sim \pi(a_1|s_1)} [Q(s_1, a_1)|s_1] \right] \quad (2.8)$$

Assuming $Q(s_1, a_1)$ is known for each possible action in action space at this time step, it would be feasible to maximize the objective by choosing the action, that maximizes the expected reward. At this point the Q function can be defined as the expected reward that the agent takes under the policy by taking some action at some state and can be mathematically shown as follows:

$$Q(s_t, a_t) = \sum_{t'=t}^T E_{\pi_{\theta}} [r(s_{t'}, a_{t'})|s_t, a_t] \quad (2.9)$$

On the other hand, a closely related function can be defined, which is the value function [7, chapter 3.2]. The value function is conditioned only on the state rather than on both the state and the action and can be expressed as follows:

$$V^{\pi}(s_t) = \sum_{t'=t}^T E_{\pi_{\theta}} [r(s_{t'}, a_{t'})|s_t] \quad (2.10)$$

Having only the Q function or both the Q function and the value function, there would be two different approaches to improve the policy. The first is that we may use the Q function to judge all actions that the policy may take and set $\pi'(a|s) = 1$ if $a = \operatorname{argmax}_a Q(s, a)$, which means that the improved policy is at least as good as its predecessor or probably better. The other way, both $Q(s, a)$ and $V(s)$ are given. Assuming $Q(s, a) > V(s)$, this means that taking action a is a better choice than average. Thus, we may modify the policy π to increase the probability of taking the action a .

In summary, both value function and the Q function are of primary importance when making and evaluating decisions. Action selection and policy optimization are based on value judgments, with the pursuit of actions that lead to states of maximal expected reward over the trajectory, rather than maximal instant reward, due to their potential to generate the greatest reward over the long term.

3 Model-Free Reinforcement Learning

RL algorithms are characterized by three key components: policy, reward, and value function. Additionally, some algorithms may include a model function of the environment as an optional element. Thus, these algorithms can be classified into two categories: model-based and model-free. Model-based systems involve an agent that endeavors to comprehend its surroundings and formulates a model of it based on its interactions with said environment. On the other hand, model-free algorithms utilize experience-based learning to determine the consequences of their actions through the use of algorithms such as policy gradient and Q-learning. Essentially, these algorithms repeatedly perform an action and adapt their strategy, known as a policy, in pursuit of optimal rewards based on the results achieved. In this chapter we will first discuss different model-free algorithms.

3.1 Policy-Based Methods

Model-free algorithms can be further divided into two subtypes: policy-based and value-based. Policy-based algorithms seek to learn a policy function directly, which defines a mapping between states and the most suitable corresponding action or a probability distribution over the set of possible actions at that state. This function may be deterministic, meaning that it always returns the same action for a given state or stochastic, meaning that it outputs a probability distribution over actions. In contrast, value-based methods focus on training a value function, which maps states to the anticipated value of being in that state, calculated as the expected discounted return the agent can obtain if it begins in that state and follows the designated policy. This value function allows the policy to identify the state with the highest value at each step, as determined by the value function.

3.1.1 REINFORCE

Given a trajectory τ , the probability of the agent taking actions at each timestep to traverse this trajectory can be unpacked using the Markovian property of the environment as shown in

section 2.4.4 and expressed as follows:

$$p_{\pi}(\tau) = p_{\pi}(s_1, a_1, \dots, s_t, a_t) = p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t|s_t) p(s_{t+1}|s_t, a_t) \quad (3.1)$$

Recalling the main objective of RL algorithms is to maximize the expected value of the reward function under the probability of a trajectory induced by the policy which can be expressed as follows:

$$J(\theta) = E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right] = \int p_{\theta}(\tau) r(\tau) d\tau \quad (3.2)$$

In order to improve this objective using gradient ascent, the gradients of $J(\theta)$ shall be calculated. Gradients' calculation can be expressed and simplified as follows:

$$\nabla_{\theta} J(\theta) = \int \nabla_{\theta} p_{\theta}(\tau) r(\tau) d\tau = \int p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau) r(\tau) d\tau = E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) r(\tau)] \quad (3.3)$$

Utilizing the logarithmic identity for multiplication and the fact that the derivative of the transition dynamics terms with respect to θ is equal to zero. Furthermore, the gradients of the objective function can be simplified by substituting equation 3.1 into equation 3.3 as follows:

$$\begin{aligned} \nabla_{\theta} J(\theta) &= E_{\tau \sim p_{\theta}(\tau)} \left[\nabla_{\theta} \left[\log p(s_1) + \sum_{t=1}^T \log \pi_{\theta}(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right] r(\tau) \right] \\ &= E_{\tau \sim p_{\theta}(\tau)} \left[\nabla_{\theta} \left[\sum_{t=1}^T \log \pi_{\theta}(a_t|s_t) \right] r(\tau) \right] \end{aligned} \quad (3.4)$$

Due to the intractability of calculating the integral of the calculated value, it is necessary to approximate it with a more tractable expression. This approximation takes the form of the average of the collected trajectories, which can be represented as follows:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t}) \right) \left(\sum_{t=1}^T r(s_{i,t}, a_{i,t}) \right) \quad (3.5)$$

The procedure of training an agent through the use of such methods is referred to as the REINFORCE algorithm [21]. Algorithm 1 summarizes the required steps to train the policy.

Upon examination, the behavior of this algorithm can be interpreted as an attempt to influence the policy's decision-making process within the set of collected trajectories. Specifically, it appears to augment the probability of actions that are associated with the most rewarding trajectories, while concurrently diminishing the likelihood of actions corresponding to trajectories that yield

Algorithm 1: REINFORCE

Data: $N \geq 0, \alpha > 0$

```

1  $i \leftarrow 0$ ;
2 while  $i < N$  do
3   sample  $\tau^i$  from  $\pi_\theta(a_t|s_t)$ ;
4    $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left( \sum_{t=1}^T r(s_{i,t}, a_{i,t}) \right)$ ;
5    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$ ;
6 end
```

poor rewards. In this manner, the algorithm incentivizes the policy to prioritize trajectories that have collected the most reward, while concurrently discouraging the selection of trajectories that have resulted in poor rewards.

The previously described configuration of this algorithm is likely to result in poor performance in real-world settings due to the high variance of its gradients during the training process. This is especially likely to occur in complex environments where the agent receives significantly different rewards for similar or even identical behavior or conversely, receives similar rewards for highly divergent behavior. Policy gradient methods such as REINFORCE, which require the updating of the policy distribution in the direction of the received reward, are prone to generating highly variable gradients and resultant unstable training, leading to suboptimal performance of the resulting policy.

However, there are several strategies [22] that can mitigate this issue. One such approach involves utilizing the inherent causality of natural phenomena, which is an inherent characteristic. It is important to recognize that the policy's actions at a particular timestep do not affect the rewards associated with past actions, but rather only exert an influence on future rewards, which can be referred to as "reward to go" $\hat{Q}_{i,t}$. Based on this understanding, the gradient calculation can be reformulated as follows:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}) \right) \quad (3.6)$$

The rewriting of the gradient formula to the aforementioned expression leads to a reduction in gradient values due to the diminution of reward terms and subsequently results in a decrease in variance according to the variance calculation formula provided below:

$$\text{Var}(X) = E[X^2] - E[X]^2 \quad (3.7)$$

3.1.2 Advantage Actor-Critic (A2C)

Advantage actor-critic (A2C) [23] is one of the first practical policy gradient algorithms that was widely used as good alternative to REINFORCE because of its ability to reduce the gradients' variance. Given that the agent traverses through varying trajectories, each producing positive rewards in an environment that exclusively rewards the agent and refrains from penalizing it, it is observed that the agents who receive higher rewards perform superiorly compared to those who receive lower rewards, despite all agents receiving a positive reward value at the end of every episode. In such conditions, the algorithm behaves in a way that contradicts our anticipated outcome, which entails the increase in the probability of taking actions that yield high rewards while decreasing the probability of actions that yield low rewards. Consequently, the algorithm ends up increasing the probability of taking both favorable and unfavorable actions, proportionally weighted by the rewards obtained at the end of each episode. To mitigate this behavior, baselines b are implemented by deducting a specific amount from the "Reward to go" term $\hat{Q}_{i,t}$, thereby nullifying unfavorable trajectories and maintaining positivity in the favorable ones, which can be shown as follows:

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \log p_{\theta}(\tau) [r(\tau) - b] \quad (3.8)$$

A prevalent approach to constructing baselines is to subtract the mean of the trajectories from $\hat{Q}_{i,t}$. This process aims to augment the likelihood of actions that surpass the average, while diminishing the likelihood of actions that fall below the average. This mathematical representation of the equation is as follows:

$$b = \frac{1}{N} \sum_{i=1}^N r(\tau) \quad (3.9)$$

The act of subtracting a constant value from the reward is not a random action but is instead a deliberate choice. Baselines have two significant advantages. The first advantage is that it does not have an impact on the expected gradients and causes no bias since probability density equations always integrate to 1 and the gradient of constant values is always equal to 0. This mathematical relationship can be expressed as follows:

$$E[\nabla_{\theta} \log p_{\theta}(\tau) b] = \int p_{\theta}(\tau) \nabla_{\theta} \log p_{\theta}(\tau) b d\tau = \int \nabla_{\theta} p_{\theta}(\tau) b d\tau = b \nabla_{\theta} \int p_{\theta}(\tau) d\tau = b \nabla_{\theta} 1 = 0 \quad (3.10)$$

The second major benefit is reducing the variance of the gradients. Using the correct baseline value forces the variance to converge to the global minima, which can be shown mathematically from

equations 3.7 and 3.8 as follows:

$$Var(\nabla_{\theta} J(\theta)) = E_{\tau \sim p_{\theta}(\tau)} [(\nabla_{\theta} \log p_{\theta}(\tau) [r(\tau) - b])^2] - E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) [r(\tau) - b]]^2 \quad (3.11)$$

It is known from equation 3.10 that baselines are unbiased in expectation, therefore the second term in equation 3.11 is equal to 0. In order to calculate the global minima of the variance the first derivative of equation 3.11 with respect to b needs to be calculated, which can be shown as follows:

$$\begin{aligned} \frac{Var(\nabla_{\theta} J(\theta))}{db} &= \frac{d}{db} E_{\tau \sim p_{\theta}(\tau)} [(\nabla_{\theta} \log p_{\theta}(\tau) [r(\tau) - b])^2] \\ &= \frac{d}{db} (E[(\nabla_{\theta} \log p_{\theta}(\tau))^2 r(\tau)^2] - 2E[(\nabla_{\theta} \log p_{\theta}(\tau))^2 r(\tau) b] + b^2 E[(\nabla_{\theta} \log p_{\theta}(\tau))^2]) \\ &= -2E[(\nabla_{\theta} \log p_{\theta}(\tau))^2 r(\tau)] + 2bE[(\nabla_{\theta} \log p_{\theta}(\tau))^2] = 0 \end{aligned} \quad (3.12)$$

$$b = \frac{E[(\nabla_{\theta} \log p_{\theta}(\tau))^2 r(\tau)]}{E[(\nabla_{\theta} \log p_{\theta}(\tau))^2]} \quad (3.13)$$

According to the equation 3.13, it can be demonstrated that the optimal baseline corresponds to the expected value of the reward, which is weighted by the magnitude of the gradient. Additionally, a closer examination reveals that the baseline b is contingent on the gradients. Consequently, the baseline vector's dimensionality aligns with that of the parameters, signifying that each parameter should have a distinct baseline to minimize overall variance.

Calculating the baseline for each parameter of the policy π_{θ} as suggested by the optimal baseline in equation 3.13 can be an infeasible approach. Hence, a more practical solution is to use a general function approximator (e.g. deep neural network) that produces some advantage estimation of taking an action a in a state s . Such function approximator can be denoted as an advantage function $\hat{A}(s_t, a_t)$. Estimating an advantage value can be done through learning the Value function $V(s_t)$ and Q-function $Q(s_t, a_t)$ as the following approximation suggests:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \approx r(s_t, a_t) + V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \quad (3.14)$$

There are two primary approaches to learn the value function $V(s_t)$. The first method is based on Monte Carlo single sample estimation [7, chapter 5.2], which aims to approximate the value function with the reward-to-go, using a supervised regression method. The approximation can be

formulated as:

$$\begin{aligned} V^\pi &\approx y = \sum_{t'=t}^T \gamma^{t'-t} r(s_{t'}, a_{t'}) \\ L(\phi) &= \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi - y_i\|^2 \end{aligned} \quad (3.15)$$

In the event of non-episodic tasks, where the number of time-steps is infinite (i.e. $T = \infty$), the estimated values of $\hat{V}_\phi^\pi$ may tend to approach infinity. To tackle this issue, an extra discount factor, γ (where $\gamma \leq 1$), has been introduced in the aforementioned equation to account for the notion that the reward-to-go should be primarily influenced by current rewards, as opposed to future rewards. It should also be noted that current actions continue to affect the rewards obtained in the future. The discount factor alters the Markovian decision process (MDP) of the environment by introducing a terminal state on each step, in which the agent can land with a probability equal to $1 - \gamma$. Monte Carlo estimation approach is effective in reducing variance, since neural networks exhibit flexibility in recognizing that similar states correspond to the same value. This characteristic is particularly evident in continuous action spaces, where it is unlikely for a given state to be visited more than once.

Recalling the original value of the value function V from section 2.4.4 another way of approximating the value function can be derived as follows:

$$\begin{aligned} y_{i,t} &= \sum_{t'=t}^T E_{\pi_\theta}[r(s_{t'}, a_{t'}) | s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \approx r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) \\ L(\phi) &= \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi - y_i\|^2 \end{aligned} \quad (3.16)$$

The learning approach described above is known as temporal difference (TD) learning [7, chapter 6], which offers several advantages over the Monte Carlo estimation method. First, the neural network can be updated at each time step, a process referred to as bootstrapping, as opposed to waiting until the end of the episode, as required by Monte Carlo estimation. This can be particularly useful in scenarios where the episode length is infinite. Additionally, since the estimated value function is used to update the same neural network, the estimated value will eventually converge to an average, thereby yielding a reduced variance of the estimated values.

Before summarizing A2C, it is important to address a problem related to the use of neural networks as universal approximators. The accuracy of their approximation heavily depends on the nature of the data used for training and the targets that need to be approximated. If the target values are inaccurate, as is the case in TD Learning, the neural networks can become biased, while their variance is typically low. In contrast, Monte Carlo single sample estimation, which uses rewards from the environment as target values, is unbiased but has a high variance. Therefore, it is necessary to strike a balance between both approaches by combining them. To reduce the bias, target values from multiple steps should be used to train the estimator instead of using only one step as in TD Learning. This approach is known as the n-step bootstrapping [7, chapter 7.1] and the corresponding

Algorithm 2: Advantage Actor Critic(A2C)

Data: $N \geq 0, M \geq 0, \alpha > 0, 1 \geq \gamma > 0, 1 \geq \lambda > 0$

```

1  $i \leftarrow 0$ ;
2 while  $i < N$  do
3   Take M steps using  $\pi_\theta(a|s)$  and collect  $(s, a, s', r)$  for each step;
4   Bootstrap current  $V_\phi^\pi(s)$  and next  $V_\phi^\pi(s+1)$  value for each step;
5   Calculate GAE  $A_{GAE}^\pi$  for each step using  $V_\phi^\pi(s+1)$ ;
6    $y \leftarrow A_{GAE}^\pi + V_\phi^\pi(s)$ ;
7   update  $V_\phi^\pi$  using targets  $y$ ;
8    $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A_{GAE}^\pi(s, a)$ ;
9    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$ ;
10 end

```

equation used to train the neural networks can be expressed as follows:

$$\hat{A}_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) \right] - \hat{V}_\phi^\pi(s_t) + \gamma^n \hat{V}_\phi^\pi(s_{t+n}) \quad (3.17)$$

The concept of n-step return can be extended to incorporate all n-steps that are weighted by a factor λ , resulting in a generalized approach known as generalized advantage estimation (GAE) [24]. This approach allows for a trade-off between bias and variance of the neural networks, with smaller n values resulting in lower variance but higher bias and vice versa.

$$\begin{aligned} \hat{A}_{GAE}^\pi &= r(s_t, a_t) + \gamma \left((1-\lambda) \hat{V}_\phi^\pi(s_{t+1}) + \lambda \left(r(s_{t+1}, a_{t+1}) + \gamma \left((1-\lambda) \hat{V}_\phi^\pi(s_{t+2}) + \lambda (r(s_{t+2}, a_{t+2}) + \dots) \right) \right) \right) \\ &= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'} \end{aligned} \quad (3.18)$$

where $\delta_{t'} = r(s_{t'}, a_{t'}) + \gamma \hat{V}_\phi^\pi(s_{t'+1}) - \hat{V}_\phi^\pi(s_{t'})$. Algorithm 2 summarizes the steps required to train an agent using A2C.

3.1.3 Proximal Policy Optimization (PPO)

Proximal policy optimization (PPO) [25] was developed as a remedy to the intricacies associated with trust region policy optimization (TRPO) [26], that was introduced before as one of many optimizations to policy-based algorithms since the first policy based algorithm in the last century. TRPO introduced for the first time the concept of trust region as a constraint to enhance the expected advantage of the updated policy over its predecessor, since the introduction of policy gradient methods at the close of the preceding century. Trust Region based algorithms have since remained preminent and have been the preferred algorithms in policy gradient-based RL algorithms

for control and Atari problems. The motivation for trust region based algorithms stems from the equation that relates the improvement in the policy's objective function to the expected value of the trajectories generated by the updated policy distribution, denoted as $\pi_{\theta'}$ in which $\theta' = \theta + \Delta\theta$, with respect to the advantage of the previous policy π_{θ} . This equation (proof in the TRPO paper [26]) is represented as follows:

$$J(\theta') - J(\theta) = E_{\tau \sim p_{\theta'}(\tau)} \left[\sum_t A^{\pi_{\theta}}(s_t, a_t) \right] \quad (3.19)$$

The right-hand side of the equation presented above can be reformulated in terms of samples of states and actions from the state distribution $p_{\theta'}(s_t)$ and policy $\pi_{\theta'}(a_t|s_t)$ respectively, as follows:

$$J(\theta') - J(\theta) = \sum_t E_{s_t \sim p_{\theta'}(s_t)} \left[E_{a_t \sim \pi_{\theta'}(a_t|s_t)} [A^{\pi_{\theta}}(s_t, a_t)] \right] \quad (3.20)$$

By employing importance sampling [27], the sampling process from the updated policy $\pi_{\theta'}$, which has not been obtained yet, can be transformed into a sampling process from the current policy, π_{θ} . This can be achieved by rolling out the current policy in the environment. This transformation can be mathematically expressed as follows:

$$J(\theta') - J(\theta) = \sum_t E_{s_t \sim p_{\theta'}(s_t)} \left[E_{a_t \sim \pi_{\theta}(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} A^{\pi_{\theta}}(s_t, a_t) \right] \right] \quad (3.21)$$

Sampling from the state distribution $p_{\theta'}(s_t)$ is still challenging because it is not feasible to determine the state distribution of the updated policy without conducting a sampling procedure. As a consequence, the equation presented above can be reformulated, but this transformation involves introducing an approximation error denoted as $\mathcal{L}(\theta + \Delta(\theta))$, which is known as surrogate advantage loss.

$$J(\theta') - J(\theta) \approx \sum_t E_{s_t \sim p_{\theta}(s_t)} [E_{a_t \sim \pi_{\theta}(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} A^{\pi_{\theta}}(s_t, a_t) \right]] = \mathcal{L}(\theta') \quad (3.22)$$

The approximation error can be expressed with respect to the maximum Kullback Leibler (KL) divergence between the two policies, as it is the case in the approximated posterior of variational autoencoder (VAE) [28]. Thus, we can denote the following inequality:

$$J(\theta') - J(\theta) \geq \mathcal{L}_{\pi_{\theta}}(\theta') - \mathcal{CD}_{KL}^{max}(\pi_{\theta} || \pi_{\theta'}) \quad (3.23)$$

In the TRPO paper, the penalty factor $\mathcal{C}$ is obtained through analytical methods, resulting in a penalty on divergence. Therefore, by enhancing the right-hand side while decreasing the KL divergence term or keeping it constant, we can ensure an improvement in the left-hand side as well.

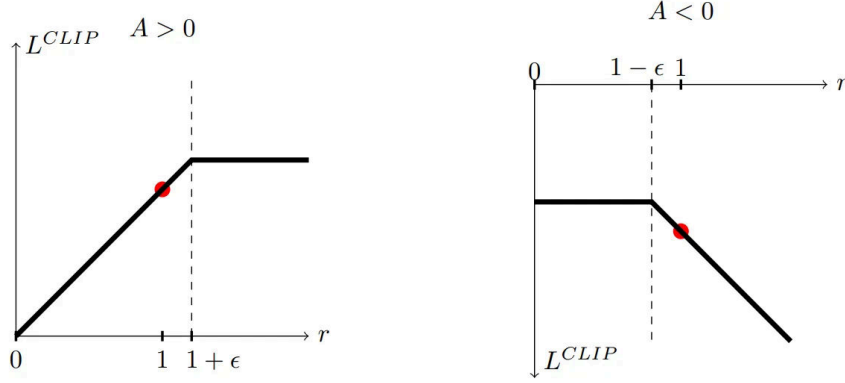


Figure 3.1: In PPO, the surrogate advantage curve slope becomes 0 beyond the trust region. [25]

Particularly, if the surrogate advantage loss $\mathcal{L}_{\pi_{\theta}}(\theta')$ surpasses the maximum KL divergence term $\mathcal{CD}_{KL}^{max}(\pi_{\theta}||\pi_{\theta'})$, we can guarantee an enhancement in the objective.

TRPO suffers from several computational drawbacks, primarily due to the calculation of the penalty factor $\mathcal{C}$. The adaptive $\mathcal{C}$ computation involves the Fisher matrix for the policy parameters, which has a time complexity of $\mathcal{O}(N^3)$. In the case of policies induced by neural networks, which can have numerous parameters, it is not feasible to compute the Fisher matrix. Additionally, TRPO is a complicated method, which is challenging to explain, implement, and debug, making it an unfriendly RL algorithm for users.

That is why PPO was later developed, as it builds upon the principles of TRPO to mitigate these complexities. PPO avoids the computational complexity of calculating the penalty factor $\mathcal{C}$ for each iteration by constraining its value to a fixed range within which the policy can vary. In PPO, updates to the policy outside the clipping range are not used for updating purposes, which incentivizes the policy to stay close to the existing one. The objective function in PPO still depends on the surrogate advantage, but it is modified to incorporate the clipping range, producing a new surrogate loss function that can be expressed as follows:

$$\mathcal{L}_{\pi_{\theta}}^{CLIP}(\pi_{\theta'}) = E_{\tau \sim p_{\pi_{\theta}}(\tau)} \left[\sum_{t=0}^T \left[\min \left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A_t^{\pi_{\theta'}}, \text{clip} \left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) A_t^{\pi_{\theta'}} \right) \right] \right] \quad (3.24)$$

In the equation above, both terms $(1 - \epsilon)$ and $(1 + \epsilon)$ do not depend on θ , resulting in a gradient of 0. Therefore, any samples that fall outside the trusted region are effectively ignored, discouraging updates that are too large as shown in figure 3.1. As a result, PPO does not directly limit the policy update, but rather disregards advantages that come from policies that deviate too much from the current policy.

Algorithm 3 shows the steps required to train a PPO agent.

Algorithm 3: Proximal Policy Optimization(PPO)

Data: $N \geq 0, M \geq 0, \epsilon > 0, \alpha > 0, 1 \geq \gamma > 0, 1 \geq \lambda > 0$

```

1  $i \leftarrow 0$ ;
2 while  $i < N$  do
3   Take M steps using  $\pi_\theta(a|s)$  and collect the transition tuple  $(s, a, s', r)$  for each step;
4   Bootstrap current  $V_\phi^\pi(s)$  and the next  $V_\phi^\pi(s+1)$  value for each step;
5   Calculate GAE  $A_{GAE}^\pi$  for each step using  $V_\phi^\pi(s+1)$ ;
6    $y \leftarrow A_{GAE}^\pi + V_\phi^\pi(s)$ ;
7   Update  $V_\phi^\pi$  using targets y;
8    $\mathcal{L}_{\pi_\theta}^{CLIP}(\pi_{\theta'}) \leftarrow E_{\tau \sim p_{\pi_\theta}(\tau)} \left[ \sum_{t=0}^T \left[ \min \left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} A_t^{\pi_{\theta'}}, \text{clip} \left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) A_t^{\pi_{\theta'}} \right) \right] \right]$ ;
9    $\nabla_\theta (J(\theta + \Delta_\theta) - J(\theta)) \approx \nabla_\theta \mathcal{L}_{\pi_\theta}^{CLIP}(\pi_{\theta'})$ ;
10   $\theta \leftarrow \theta + \alpha \nabla_\theta (J(\theta + \Delta_\theta) - J(\theta))$ ;
11 end

```

3.2 Value-Based Methods

Drawing on the methodology of policy gradient algorithms, especially actor-critic algorithms discussed in section 3.1.2, it is understood that the aim of these algorithms is to enhance the policy by training an advantage function $\hat{A}_\phi^{\pi_\theta}$ that evaluates the actions taken by the policy. Based on this evaluation, the policy is adjusted to increase or decrease the probability of taking the same action in similar situations in the future. In contrast, value-based methods differ from policy gradient methods by completely discarding the policy gradient. Instead, they implicitly select actions based on the evaluation of the advantage function $\hat{A}_\phi^{\pi_\theta}$. This enables the policy to be reformulated as follows:

$$\pi(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = \text{argmax}_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{otherwise} \end{cases} \quad (3.25)$$

From the above expression, we can see that the policy sets a probability of 1 to actions that the advantage function $\hat{A}_\phi^{\pi_\theta}$ believes to be the best out of set of possible actions, instead of just improving the probability as in policy gradient methods.

3.2.1 Deep Q Network (DQN)

Essentially, training value based algorithms requires training the advantage function $\hat{A}_\phi^{\pi_\theta}$, which can be expressed as before as follows:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t) \quad (3.26)$$

Algorithm 4: fitted value Iteration

Data: $N \geq 0, \gamma > 0$

```

1  $i \leftarrow 0$ ;
2 while  $i < N$  do
3    $y_i \leftarrow \max_{a_i} (r(s_i, a_i) + \gamma E[V_\phi(s'_i)]);$ 
4    $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|V_\phi(s_i) - y_i\|^2;$ 
5 end
```

In order to evaluate the advantage function, it is required to learn the value function $V^\pi(s_t)$. This can be done using TD learning, which utilizes dynamic programming in its core. It is known from section 3.1.2 that value function can be defined as follows:

$$V^\pi(s_t) = E_{a \sim \pi(a|s)}[r(s, a) + \gamma E_{s' \sim p(s'|s, a)}[V^\pi(s')]] \quad (3.27)$$

The outer expectation term can be omitted, knowing that the policy is deterministic according to equation 3.33. Thus, the TD learning target can be simplified to a more convenient expression as follows:

$$V^\pi(s_t) = r(s, \pi(s)) + \gamma E_{s' \sim p(s'|s, \pi(s))}[V^\pi(s')] \quad (3.28)$$

Recalling the definition of the Q function from section 2.4.4 it can be argued that the following equation is true:

$$\operatorname{argmax}_{a_t} A^\pi(s_t, a_t) = \operatorname{argmax}_{a_t} Q^\pi(s_t, a_t) \quad (3.29)$$

Therefore, we can discard calculating the advantage function and instead calculate directly the Q function which can be calculated as follows:

$$Q^\pi(s, a) = r(s, \pi(s)) + \gamma E_{s' \sim p(s'|s, \pi(s))}[V^\pi(s')] \quad (3.30)$$

Calculating the Value function then can be done using supervised linear regression method, where targets and loss can be calculated as follows:

$$\begin{aligned} y_i &= \max_{a_i} (r(s_i, a_i) + \gamma E[V_\phi(s'_i)]) \\ L(\phi) &= \frac{1}{2} \sum_i \|V_\phi(s_i) - y_i\|^2 \end{aligned} \quad (3.31)$$

Improving the policy can be then done by iteratively improving the Q value. This algorithm is known as fitted value iteration [7, chapter 4.4] and has the following steps as shown in algorithm 4.

Algorithm 5: Q Iteration

Data: $N \geq 0, \gamma > 0$

```

1  $i \leftarrow 0$ ;
2 while  $i < N$  do
3    $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)$ ;
4    $\phi \leftarrow \operatorname{argmin}_\phi \frac{1}{2} \sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$ ;
5 end
```

However, the "max" term in the first step is problematic, because it requires knowing the transition dynamics of the environment, in order to be able to take multiple actions from the same state and select the best one. Instead of approximating V_ϕ , we can approximate directly Q_ϕ as it outputs the expected value of taking some action at the next timestep given the current state at once. These equation used to calculate these new targets is also known as Bellman equation. It can be expressed as follows:

$$y_i = r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i) \quad (3.32)$$

The algorithm that approximates the Q-function is known as Q Iteration [7, chapter 6.5] and can be summarized as shown in algorithm 5.

The Q-learning algorithm, a widely used RL method, has two main variants: an online variant that iteratively updates the policy after each step in the environment and another variant that collects a fixed number of steps K before updating the policy. However, both variants suffer from the issue of high correlation between sequential steps, which can lead to local overfitting of the policy to certain states, especially in continuous state settings where nearby states are likely to be similar. To mitigate this issue, an experience replay buffer can be used to store data sampled from a broad range of policies and a batch of (s, a, s', r) can be uniformly sampled from the buffer for K steps to update the policy.

Another problem with Q-learning is that the targets y_i are continuously changing with every policy update, resulting in a moving target issue and preventing the policy from converging to the optimal solution. To address this, two separate neural networks can be used to decouple the calculation of targets y_i and Q values, with one network dedicated to improving the policy with parameters ϕ and the other solely for calculating targets with parameters ϕ' .

Moreover, a deterministic policy may limit exploration and fail to consider other actions that could potentially yield better results. Therefore, it is recommended to introduce some level of exploration by randomly selecting actions with a probability ϵ . The policy $\pi(a_t|s_t)$ can be reformulated

Algorithm 6: Deep Q Network(DQN)

Data: $N \geq 0, K \geq 0, B > 0, 1 \geq \gamma > 0$

```

1 Initialize an experience replay buffer  $\mathcal{B}$  of size B;
2 Initialize an online network with parameters  $\phi$  and a target network with parameters  $\phi'$ ;
3  $i \leftarrow 0$ ;
4 while  $i < N$  do
5   Update target network's parameters:  $\phi' \leftarrow \phi$ ;
6    $j \leftarrow 0$ ;
7   while  $j < K$  do
8     take action  $a_i$  and observe state transition  $(s_i, a_i, s'_i, r_i)$  and add it to  $\mathcal{B}$ ;
9     Sample mini-batch  $\mathcal{M} (s_j, a_j, s'_j, r_j)$  from  $\mathcal{B}$  uniformly;
10     $y_i \leftarrow r(s_j, a_j) + \gamma \max_{a'} Q_{\phi'}(s'_j, a'_j)$  using target network  $Q_{\phi'}$ ;
11     $\phi \leftarrow \operatorname{argmin}_{\phi} \frac{1}{2} \sum_i ||Q_{\phi}(s_i, a_i) - y_i||^2$ ;
12  end
13 end

```

accordingly to accommodate this exploration strategy.

$$\pi(a_t|s_t) = \begin{cases} 1 - \epsilon & \text{if } a_t = \operatorname{argmax}_{a_t} A^{\pi}(s_t, a_t) \\ \frac{\epsilon}{(|A|-1)} & \text{otherwise} \end{cases} \quad (3.33)$$

where $|A|$ is the number of possible actions in the action space. Combining all these optimizations into a single procedure results in Deep Q Network (DQN) [29]. Algorithm 6 illustrates the required steps to train a DQN.

3.2.2 Rainbow

Since the development of DQN it was suffering from multiple issues regarding exploration, training the Q function and the estimated values provided by the Q function. However, over the following years multiple techniques were developed in order to solve each of these problems. Rainbow [30] combines all of these techniques into one algorithm that outperforms all of them individually.

I. Double Q-Learning

In accordance with Q-Learning, actions are chosen to maximize the anticipated return for a given state. However, this method is not without its drawbacks, as estimated Q-values are prone to

overestimation. This overestimation arises from the selection of the action that maximizes Q-values from a Q-Function, which can be thought of upon initialization as the true Q-Function and unbiased noise in both positive and negative directions. When calculating the targets by repeatedly selecting the action that maximizes Q-values, positive noise accumulates systematically, leading to an overestimation of the Q-Values. Since the Q-function is acquired by means of TD learning, a bootstrapping stochastic approach, this overestimation persists and may even worsen with each iteration.

To address this issue, Double Q-Learning [31] was developed, which utilizes two distinct networks to disentangle action selection from Q-Value estimation during target calculation y_i . Consequently, even if an overestimated Q-value is used for action selection, the network update is performed with the one that does not. For each iteration, one network is arbitrarily chosen for action selection and the other for update, thus disconnecting the selection and evaluation mechanisms. However, training two distinct networks can be computationally cumbersome, hence we use the online network Q_ϕ for action selection and the target network $Q_{\phi'}$ for Q-value estimation. The target calculation is then reformulated as follows:

$$y_i = r(s_i, a_i) + \gamma Q_{\phi'}(s', \argmax_{a'} Q_\phi(s', a')) \quad (3.34)$$

II. Prioritized Experience Replay

The DQN algorithm is an off-policy method, whereby its learning hinges on gathering data and storing it in a replay buffer for training purposes. However, data collection can be a costly process, particularly when training physical robots. Uniformly sampling training data from the replay buffer is not the optimal utilization of this data, given that different experiences hold varying degrees of importance. Some experiences may represent typical steps in the environment, while others may signify reaching a goal or receiving a penalty for making an error. To optimize the use of the sampled data, it is crucial to sample past experiences that resulted in the highest rewards or penalties and hence, the greatest value function errors, with the objective of repeatedly evaluating these experiences to reduce the error. Prioritized experience replay [32] assigns a priority to past experiences so we can selectively sample the most crucial experiences based on their priority. This priority can be expressed as follows:

$$P(i) = \frac{\mathcal{L}_{TD_i}^\alpha}{\sum_k \mathcal{L}_{TD_i}^\alpha} \quad (3.35)$$

In this expression, $P(i)$ denotes the priority assigned to a sample based on its TD loss, while alpha represents a parameter that determines the extent of prioritization. However, relying solely on

prioritization when selecting experiences may lead to a sampling bias, resulting in the network overfitting exclusively to these samples. To overcome this issue, the samples' losses are weighted by a decaying weight w_i , causing them to lose importance over time and therefore their sampling priority diminishes. The weights are defined as follows:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta \quad (3.36)$$

where N is the batch size and β is scaling parameter.

III. Dueling Networks

Recalling the definition of the advantage function from equation 3.14 this formula can be reexpressed in terms of Q^π as follows:

$$Q^\pi(s_t, a_t) = A^\pi(s_t, a_t) + V^\pi(s_t) \quad (3.37)$$

The fundamental concept of dueling networks [33] is to decompose Q^π into two distinct streams. This approach is motivated by the fact that, in some environments, it is unnecessary to know the value of each action at every time step. In certain states, taking any action would not make a difference due to the state's lower value, such as driving on a road with no obstacles. Consequently, it becomes less critical to understand the impact of each action in such states. The original DQN implementation fails to clarify whether the value provided by the Q function results from taking a valuable action or merely being in a valuable state, such as being close to the target. A dueling architecture enables the algorithm to identify good actions more quickly by discarding poor actions more efficiently and comparing comparable actions more rapidly. By decoupling the advantage value from the state value, as illustrated in figure 3.2, we obtain a more detailed perspective on the action value.

The Q-Value for some state can be calculated by combining the advantage and state estimation. Both share the same feature extractor but have different heads (i.e. different output streams) that are also parametrized differently. This estimation can be shown as follows:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \alpha) + \left(A(s, a; \theta, \beta) - \frac{\sum_{a'} (A(s, a'; \theta, \beta))}{|\mathcal{A}|} \right) \quad (3.38)$$

where θ , α and β are the parameters of the shared feature extractor, state estimation head and advantage head, respectively. The proposed architecture has been shown to quickly identify good actions, as identifying action values has become an explicit part of the learning problem, based on empirical results proposed by the original paper.

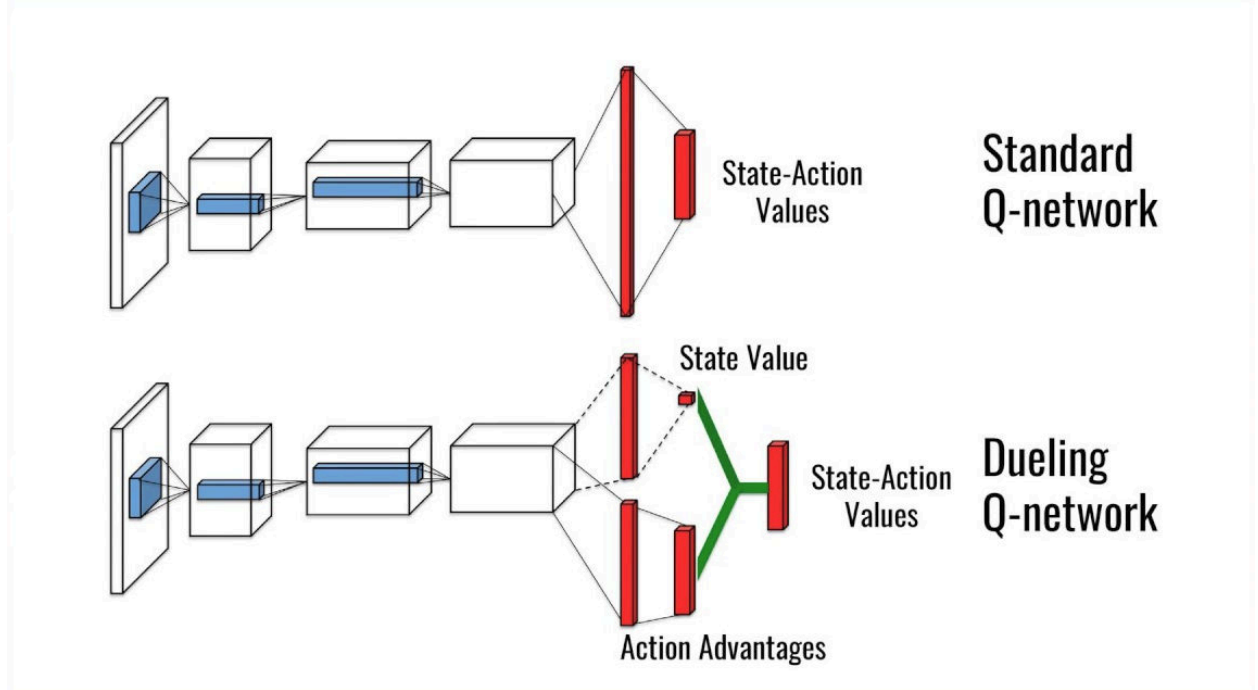


Figure 3.2: Comparing standard DQN and Dueling Network. Standard DQN outputs a value based on both actions and states. Dueling network decouples both action and value evaluation. [33]

IV. Multi-step learning

As explained in section 3.2.1 Q-learning uses a single step reward and then uses the greedy action to bootstrap the expected value of the the next state. However bootstrapping in general is biased –as proposed in section 3.1.2– , as the function used to bootstrap is biased by it is nature because it is an approximator. In order to reduce this bias and increase convergence speed to the optimar Q function multiple rewards from successive steps can be utilized to calculate the training targets. This multiple step reward can be defined as follows:

$$R_t^n = \sum_{k=0}^{n-1} \gamma_t^k R_{t+k+1} \quad (3.39)$$

where n represents the number of steps to estimate the return. To calculate the targets, the rest of the horizon is bootstrapped using the greedy action at time step $t + n$ and the bootstrapped value is added to the n -step reward. The calculation of targets can be expressed as follows:

$$y_i = R_t^n + \gamma_t^n \max_{a'} Q_\phi(S_{t+n}, a') \quad (3.40)$$

For many problem settings, multi-step learning yields a substantial improvement over single step estimation, however as the the number of steps increases the estimation’s variance also increases.

V. Distributional Reinforcement Learning

The Q function, as utilized by RL algorithms such as DQN, endeavors to prognosticate the anticipated return value, which corresponds to the mean of the targets that the algorithm has been trained on. Nevertheless, this approach may not exhibit optimal generalization capabilities in an environment where stochasticity is prevalent. Instead, distributional techniques endeavor to apprehend the complete distribution of expected values, such as the categorical or normal distribution. To achieve this, the training targets necessitate reformulation in order to enable updating of the entire distribution concurrently, for example, by way of determining the probability of categories in a categorical distribution. Accordingly, the revised targets may be formulated as follows:

$$D(s, a) = r(s, a) + \gamma D(s', a') \quad (3.41)$$

where D is the reward distribution of a state and action combination. However, enhancing the distributional representation of rewards cannot be accomplished via conventional linear regression loss. Rather, a more appropriate loss function for this task is the KL divergence. This particular divergence is a measure of the dissimilarity between two probability distributions. Thus, the process of updating the reward distribution entails reducing the discrepancy between the optimal reward distribution and the learned distribution, effectively minimizing the KL divergence between the two.

VI. Noisy Nets

The conventional DQN algorithm relies on ϵ -greedy exploration to probe for new actions beyond those favored by the Q-function, thereby facilitating the discovery of uncharted states and actions that may yield higher rewards. However, in environments characterized by a multitude of states that require exploration (e.g., sparse-reward environments), an excessive number of actions may need to be executed in order to discover the rewards. To address this issue, Rainbow adopts the utilization of noisy linear layers [34], enabling the generation of novel actions at the outset, which may be mathematically expressed as follows:

$$y = (b + Wx) + (b_{\text{noisy}} + (W_{\text{noisy}} \circ w)x) \quad (3.42)$$

The noisy linear layer can be conceptualized as a normal distribution with mean values of b and W , as well as variances of b_{noisy} and W_{noisy} , respectively, that are sampled from a unit Gaussian distribution. However, as the training progresses, this variance progressively approaches zero, enabling the neural network to make deterministic decisions regarding the optimal actions after the initial exploration of novel states and actions.

3.2.3 Soft Actor-Critic (SAC)

Soft actor-critic (SAC) [35] is a form of RL that bridges both Q-Learning and policy gradient approaches. However, it differs from both of these methods in several key aspects. Policy gradient methods are on-policy, meaning that the gathered data is used for training only once, resulting in inefficiency in memory utilization. By saving transitions in an experience buffer, SAC allows the algorithm to learn from prior experiences and avoid overfitting to newly collected data. While it may not be feasible to train a policy gradient algorithm on outdated data induced by an outdated or alternative policy, the policy does not directly learn from the experience sampled from the buffer. Instead, the policy is designed to optimize the Q-values generated by concurrently trained Q-functions. Since Q-functions are trained to satisfy the Bellman equation 3.45, which is valid for all transitions, it does not matter if they are trained on outdated experiences.

The fundamental characteristic of SAC is the incorporation of an entropy regularization term in the training process. By maximizing the reward income while simultaneously maintaining a high entropy of the actions' distribution, the algorithm aims to strike a balance between exploration and exploitation. Entropy can be broadly defined as a measure of how broad or diverse a distribution is and it can be mathematically calculated using the following formula:

$$H(P) = -E_{x \sim P} [\log P(x)] = - \sum_x P(x) [\log P(x)] \quad (3.43)$$

Maintaining high entropy in addition to optimizing the expected reward ensures that the agent is trained to reach the goal from a variety of different states, thereby enhancing its robustness. By preventing the agent from overfitting to a single trajectory, it is encouraged to explore the state space more broadly and develop a more comprehensive understanding of the environment. This makes the agent more capable of handling unseen or unexpected scenarios, thereby improving its overall performance.

SAC consists of three networks: one actor and two separate Q-networks. The reason for training two Q-functions simultaneously is to avoid overestimating the Q-values, as described in section 3.2.2. The two Q-networks learn concurrently to predict the optimal Q-values by trying to minimize the mean squared error of predicted values with the Bellman targets. The Bellman targets are

Algorithm 7: Soft Actor-Critic (SAC)

Data: $N \geq 0$, $K \geq 0$, $B > 0$, $1 \geq \gamma > 0$, $\alpha > 0$

```

1 Initialize ring replay buffer  $\mathcal{B}$  of size B;
2 Initialize two online Q-networks with parameters  $\phi_{1,2}$ ;
3 Initialize two target Q-networks with parameters  $\phi'_{1,2}$ ;
4 Initialize a policy network with parameters  $\theta$ ;
5  $i \leftarrow 0$ ;
6 while  $i < N$  do
7   Update target network's parameters:  $\phi' \leftarrow \phi$ ;
8    $j \leftarrow 0$ ;
9   while  $j < K$  do
10    Take an action  $a_i$  and observe state transition  $(s_i, a_i, s'_i, r_i)$  and add it to  $\mathcal{B}$ ;
11    Sample mini-batch  $\mathcal{M} (s_j, a_j, s'_j, r_j)$  from  $\mathcal{B}$  uniformly;
12    Calculate Q-Networks targets.;
13     $y_i \leftarrow r(s_i, a_i) + \gamma \left( \min_{k=1,2} Q_{\phi,k}(s'_i, a'_i) - \alpha \log \pi_\theta(a'|s') \right)$ ;
14    Update Q-Networks Parameters.;
15     $\phi_{1,2} \leftarrow \operatorname{argmin}_{\phi_{1,2}} \frac{1}{2} \sum_i ||Q_\phi(s_i, a_i) - y_i||^2$ ;
16    Calculate policy network's gradients.;
17     $\nabla_\theta J(\theta) \leftarrow \nabla_\theta [\pi_\theta(s) [\min_{k=1,2} Q_{\phi,k}(s_i) + \alpha \log \pi_\theta(s)]]$ ;
18    Update policy network's parameters.;
19     $\theta \leftarrow \theta + \nabla_\theta J(\theta)$ ;
20   end
21 end

```

calculated by bootstrapping the least value for each action from both Q-functions and adding an entropy regularization term. The entropy regularization term can be calculated as follows:

$$y_i = r(s_i, a_i) + \gamma \left(\min_{k=1,2} Q_{\phi,k}(s'_i, a'_i) - \alpha \log \pi_\theta(a'|s') \right) \quad (3.44)$$

where α is a regularization factor that can either be set as a constant or learned through a linear regularizer, which helps to ensure that the action distribution converges to a fraction of the maximum entropy.

On the other hand, the policy is trained to maximize the expected future return plus maintaining the entropy, the gradients for training the policy can be calculated as follows:

$$\nabla_\theta J(\theta) = \nabla_\theta [\pi_\theta(s) [\min_{k=1,2} Q_{\phi,k}(s_i) + \alpha \log \pi_\theta(s)]] \quad (3.45)$$

Algorithm 7 summarizes the following training steps of a SAC agent.

4 Model-Based Reinforcement Learning

Model-based RL constitutes a distinct class of algorithms that not only learns a policy but also acquires knowledge about the environment’s transition dynamics and reward mechanism. These algorithms are considerably more efficient in terms of data consumption compared to model-free algorithms. Learning the environment’s dynamics can be an optimal choice in settings where acquiring data is costly, such as in the case of real robot samples. However, this knowledge acquisition can lead to significant computational and memory burdens. Various techniques can be employed to leverage the acquired model, such as using it for data sampling to train the agent, similar to model-free methods [36]. Additionally, the acquired model can be employed for planning purposes using classical algorithms like model predictive control [37]. A combination of both model-free and model-based techniques can also be adopted, as in the case of Dreamer [38, 39, 40] algorithm, which we introduce in the subsequent section.

4.1 Dreamer

Dreamer is a recent model-based RL algorithm that has exhibited exceptional performance across a diverse range of environments, setting a state-of-the-art score in several of them. Drawing inspiration from both model predictive control and conventional model-free algorithms, Dreamer leverages a learned dynamics model known as world to forecast multiple future observations. Based on these predictions, Dreamer trains concurrently a meta RL algorithm (e.g actor-critic) to select the most optimal action as shown in figure 4.1.

The world model assimilates the latent representation of observations as an autoencoder. This latent encoding is then utilized to forecast forthcoming states and rewards by means of a recurrent state-space model (RSSM).

The RSSM serves as the foundation of the world model, which comprises several function approximators (i.e. deep neural networks), each designed for a specific task. The first of these is the

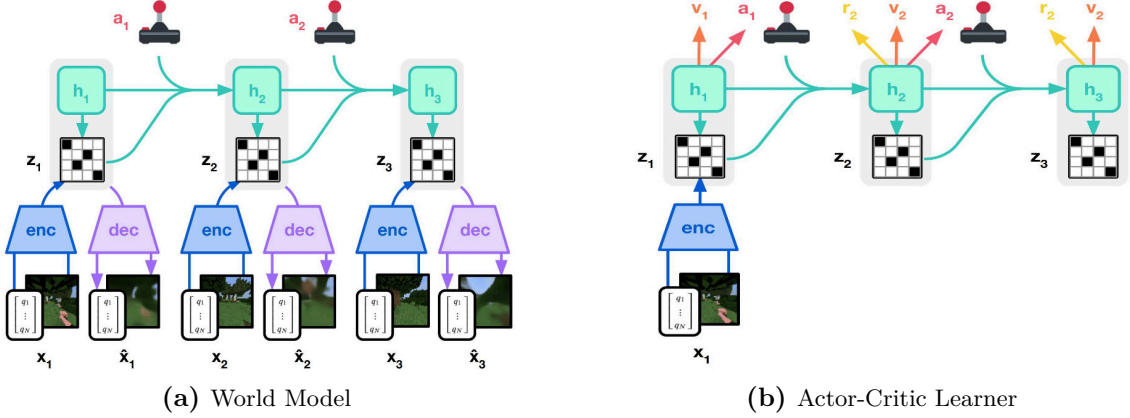


Figure 4.1: Dreamer consists of two models; world model and actor-critic learner. World model encodes observations to a latent representation that is used to predict future observations and rewards using a recurrent net. The actor-critic learner is trained using this latent encoding and reward prediction of the world model to take an optimal actions. [40]

encoder, represented by $z_t \sim q_\phi(z_t | h_t, x_t)$. It predicts a deterministic distribution of the latent state representation z_t of sensory data, such as images and data vectors and the recurrent state of the world model, h_t .

This deterministic distribution, z_t , is then combined with a stochastic distribution, $\hat{z}_t$, provided by the dynamics predictor, represented by $\hat{z}_t \sim p_\phi(\hat{z}_t | h_t)$, which takes only the recurrent state as an input. This stochastic distribution is intended to handle environments with high levels of stochasticity and prevent overfitting to local states.

Furthermore, the RSSM is equipped with a sequence model, represented by $h_t = f_\phi(h_{t-1}, z_{t-1}, a_{t-1})$, which predicts the recurrent state of the world model, h_t , given the latent state representation z_{t-1} , the previous timestep’s action a_{t-1} and the previous recurrent state h_{t-1} . This sequence model is intended to enable long-term state predictions in the future.

In addition to the RSSM, the world model incorporates several other function approximators to enhance its predictive capabilities. These include a reward predictor, represented by $\hat{r}_t \sim p_\phi(\hat{r}_t | h_t, z_t)$, which learns the rewarding system of the environment. There is also a terminal state predictor, represented by $\hat{c}_t \sim p_\phi(\hat{c}_t | h_t, z_t)$, that simulates the conditions necessary for the environment to terminate. Finally, the world model features a decoder, similar to an autoencoder, represented by $\hat{x}_t \sim p_\phi(\hat{x}_t | h_t, z_t)$. This decoder is designed to retain only the most relevant features of the sensory data and ignore those that do not contribute significantly to the data reconstruction process.

To train the world model, all the networks included, such as the encoder, decoder, and reward predictor, are trained simultaneously. The training process involves minimizing the negative log

likelihood loss, which is equivalent to the mean squared error loss, for the encoder, decoder, and reward predictor. Meanwhile, the terminal state predictor is trained using the binary classification loss. To summarize these various losses into a single function, we can define a general loss function, which can be expressed as follows:

$$\mathcal{L}_{\text{pred}}(\phi) = -\ln p_{\phi}(x_t|z_t, h_t) - \ln p_{\phi}(r_t|z_t, h_t) - \ln p_{\phi}(c_t|z_t, h_t) \quad (4.1)$$

In addition to the losses for the other networks, training the sequence model, and dynamics model of the RSSM involves two additional loss terms. The first is the dynamics loss, which is trained by minimizing the KL divergence between the representation provided by the encoder, and the dynamics predictor. This loss can be expressed as follows:

$$\mathcal{L}_{\text{dyn}}(\phi) = \max(1, KL[sg(q_{\phi}(z_t|h_t, x_t)) || p_{\phi}(z_t|h_t)]) \quad (4.2)$$

The "sg" in the equation represents "stop gradient," which is used to prevent gradients from flowing through $q_{\phi}(z_t|h_t, x_t)$ and update only $p_{\phi}(z_t|h_t)$. This is done to avoid any interference from the encoder during the training of the dynamics predictor.

The last loss term is the representation loss, which is similar to the dynamics loss. The purpose of the representation loss is to provide an alternative method for the world model to predict a reasonable latent space representation if the dynamics model is not powerful enough to do so. This allows us to have a good representation early on in the training process for training the actor. The representation loss can be expressed as follows:

$$\mathcal{L}_{\text{rep}}(\phi) = \max(1, KL[q_{\phi}(z_t|h_t, x_t) || sg(p_{\phi}(z_t|h_t))]) \quad (4.3)$$

The total loss of the world model is a weighted sum of all the loss terms discussed above. It can be expressed as follows:

$$\mathcal{L}(\phi) = E_{q_{\phi}} \left[\sum_{t=1}^T (\beta_{\text{pred}} \mathcal{L}_{\text{pred}}(\phi) + \beta_{\text{rep}} \mathcal{L}_{\text{rep}}(\phi) + \beta_{\text{dyn}} \mathcal{L}_{\text{dyn}}(\phi)) \right] \quad (4.4)$$

The actor-critic learner exclusively learns from the latent representation, reward, and terminal state provided by the world model. The world model supplies the actor-critic learner with a horizon of n-steps consisting of latent representations, rewards, and terminal signals. The actor's objective is to select a series of actions that optimize the overall return for this given sequence. Otherwise, it learns in a manner that is similar to the method described in section 2.

Dreamer comprises multiple versions, with DreamerV1 [39] being the initial version. The aforementioned exposition comprehensively covers the algorithm, except for the KL balancing strategy, which incorporates reconstruction and dynamics losses, introduced for the first time in DreamerV2 [38]. This technique led to a marked advancement in the algorithm’s performance. In comparison to DreamerV2, DreamerV3 [40] has no fundamental distinctions. Instead, DreamerV3 differs from DreamerV2 primarily due to modifications in model architecture, involving activation layers, output distributions, and input preprocessing, as well as hyperparameter adjustments, rendering it appropriate for a broad range of environments without additional tuning. These improvements have been corroborated by prior research studies, which confirm the efficacy of the Dreamer algorithmic series in a diverse range of applications.

5 Exploration

RL poses a distinctive quandary that involves balancing the trade-off between exploration and exploitation. The primary objective of any RL agent is to optimize reward by prioritizing previously successful actions. Nonetheless, in order to distinguish such actions, the agent must also experiment with unselected options. To acquire a reward, the agent must take advantage of its existing knowledge while also undertaking exploration to make more sophisticated decisions in the future. This challenge is further intensified in environments where the reward is rare or nonexistent. Consequently, this research endeavors to present two novel methods of exploration that provide intrinsic rewards to the agent for exploring unexplored states.

5.1 Intrinsic Curiosity Module (ICM)

Intrinsic curiosity module (ICM) [41] is an exploration technique that provides intrinsic rewards to the agent based on its level of familiarity with the environment’s transition dynamics. As the agent becomes more acquainted with a particular section of the environment through multiple visits, it becomes better equipped to anticipate the outcomes of its actions. Consequently, the intrinsic reward associated with this familiarity decreases. While the agent’s approximations of the environment’s dynamics cannot predict the next state with complete accuracy, the error associated with this prediction can be employed as a gauge of the novelty of the visited state.

ICM consists of two neural networks each of them has a specific task. The first is an inverse dynamics model. The inverse model possesses the capacity to convert states into a latent feature representation by predicting the action taken given the current and the next state. That is justified by the fact that attempting to predict the next state is a complex undertaking that involves several factors beyond the agent’s actions, such as environmental factors like gravity in arcade games. Therefore, a valuable state embedding should only encompass elements that the agent can exert control over. This behavior is similar to that of denoising autoencoders [42], which endeavor to learn a latent space that disregards the noise to reconstruct images that are free from noise.

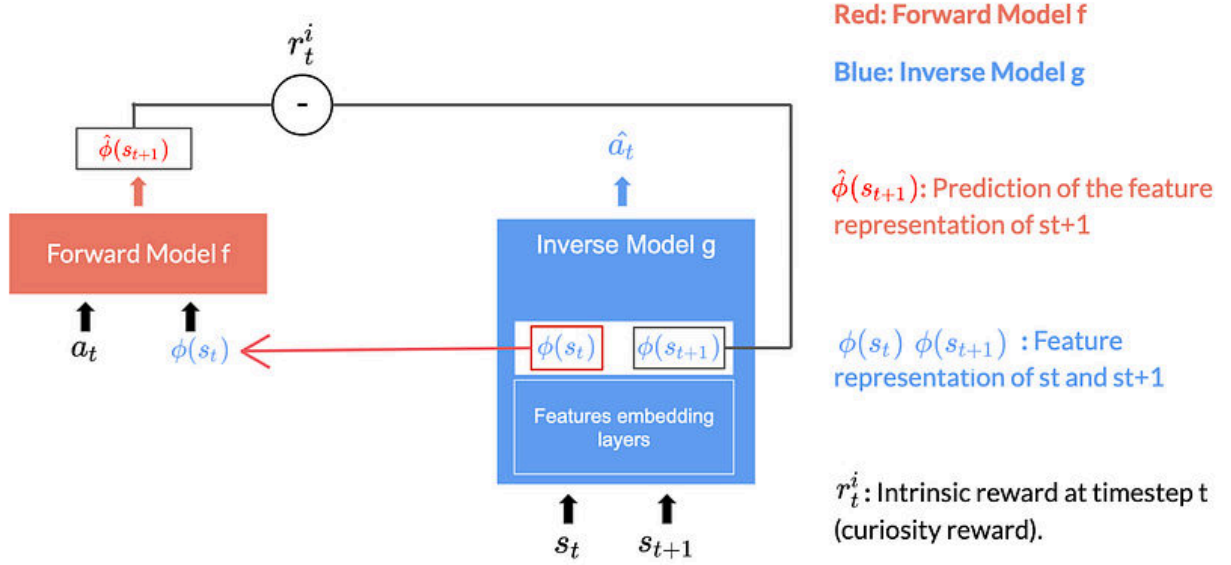


Figure 5.1: ICM consists of two neural networks; a inverse dynamics to predict the the taken action and a forward model that encodes the next state. [43]

In addition to the inverse model, ICM incorporates a forward model that learns a feature space of the next state based on the embedded current state and action. The ICM algorithm calculates the mean squared error between the predicted next state representation by the forward and inverse models, enabling it to provide intrinsic rewards to the agent based on this loss that can be calculated as follows:

$$r_t = \eta ||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2 \quad (5.1)$$

where η is a scaling factor of this quantity. As a result, the agent is encouraged to explore uncharted territory in order to gain a better understanding of the environment's dynamics.

In order to train ICM, it is necessary to calculate the loss for both the inverse and forward models independently. The inverse loss is computed as the cross-entropy loss between the predicted actions, $\hat{a}_t$ and the actions taken by the agent, a_t , which are represented in one-hot encoding. Mathematically, this loss can be expressed as follows:

$$\mathcal{L}_{Inverse} = H(\hat{a}_t|a_t) \quad (5.2)$$

where H is cross entropy loss function. On the other hand, forward loss is the mean squared error of the predicted encoding of the next state given by inverse model $\hat{\phi}(s_{t+1})$ and the forward model

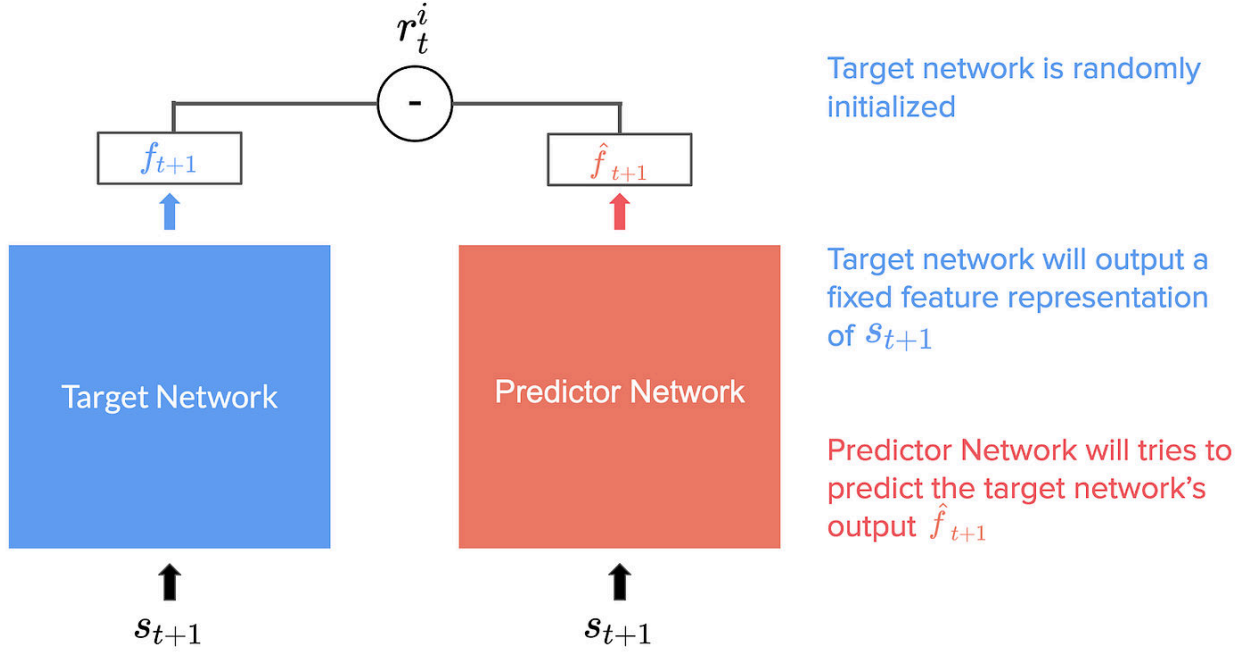


Figure 5.2: RND consists of two neural networks; The predictor tries to match its values to the randomly fixed target. [45]

$\hat{\phi}(s_{t+1})$. this can be calculated as follows:

$$\mathcal{L}_{Forward} = \frac{1}{2} \|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\|_2^2 \quad (5.3)$$

ICM is trained by minimizing the total loss of both the inverse and forward loss. The inverse loss helps the agent learn to map observations to actions, while the forward loss helps the agent learn to map observations and actions to future observations.

In stochastic environments, the ICM can struggle to predict the next state accurately, resulting in the agent receiving false rewards even for standing still. This can be a problem in scenarios like the noisy TV problem [44].

5.2 Random Network Distillation (RND)

Random network distillation (RND) [44] is an alternative method for intrinsically rewarding an agent that does not require predicting the next state. Instead, RND can be conceptualized as a density model that generates rewards based on the mean squared error between a target's embedding and a predictor's embedding of the next state as shown in figure 5.2.

In RND, the predictor network is trained to make predictions about a predetermined set of fixed, randomly generated targets generated by the target network, while the target network remains constant. The discrepancy between the predictor and target networks is measured using the mean squared error, which is utilized both to determine the reward and to train the predictor network. Specifically, when a state is visited frequently, the predictor embedding gradually becomes more closely aligned with the target embedding, which results in a lower reward.

By prioritizing the embedding of the next state over attempting to predict it, RND facilitates exploration and may exhibit reduced vulnerability to the problem of false rewards that may arise with the ICM.

6 Implementation Details

From a broader perspective, this endeavor entailed the adaptation of a two-dimensional simulation environment, the implementation of RL algorithms and the introduction of additional variations to render them operative. In this chapter, we furnish a comprehensive explication of the pragmatic exertions undertaken in this thesis.

6.1 Search And Rescue Environment

The search and rescue (SAR) environment is a two-dimensional simulation environment that emulates UAV systems tasked with locating targeted objects on a floor surface. The scenario also involves potential hazards, as depicted in figure 6.1.

The agent’s perception of the world is based on a point cloud representation of its surroundings, captured by a laser sensor as illustrated in figure 6.2. Additionally, it utilizes readings from an internal measurement unit (IMU) that includes longitudinal, lateral and radial velocities, as well as the agent’s radius. However, the agent lacks knowledge of its own position, the location of targets and how to maximize rewards, which constitutes a challenging exploration problem.

The task of working on this environment necessitated various modifications, such as integrating wrappers to tailor the observations and actions to conform to the standard OpenAI Gym API, addressing glitches and introducing a host of additional features.

6.1.1 Wrappers

The SAR environment is constructed using the Megastep [18] engine, which operates on a graphics processing unit (GPU). By leveraging GPU acceleration, the environment is capable of generating a considerable number of vectorized samples from a diverse range of environments per second. However, this diverges from the standard OpenAI Gym environments, upon which all open-source RL algorithms are built, which operate on central processing units (CPUs) and produce one sample

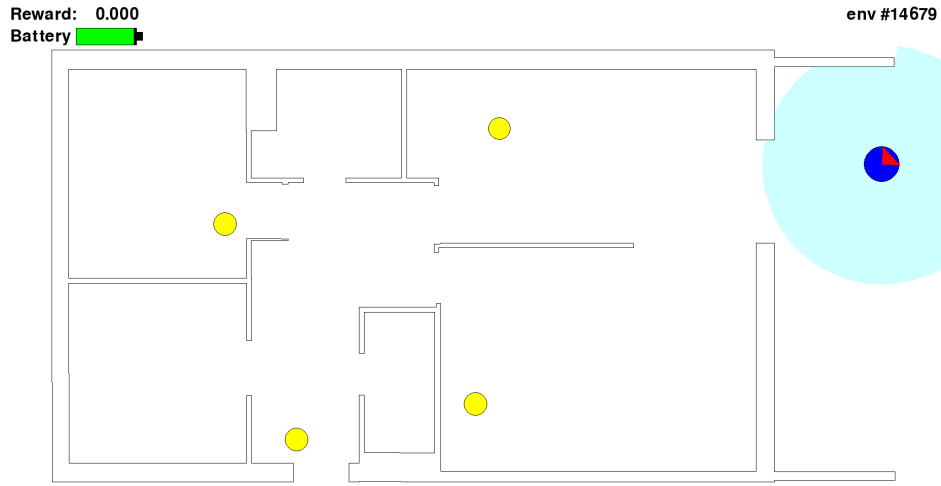


Figure 6.1: In the SAR environment, the UAV system (blue) is aimed to find the target objects (yellow), while avoiding collisions with the hazards and walls

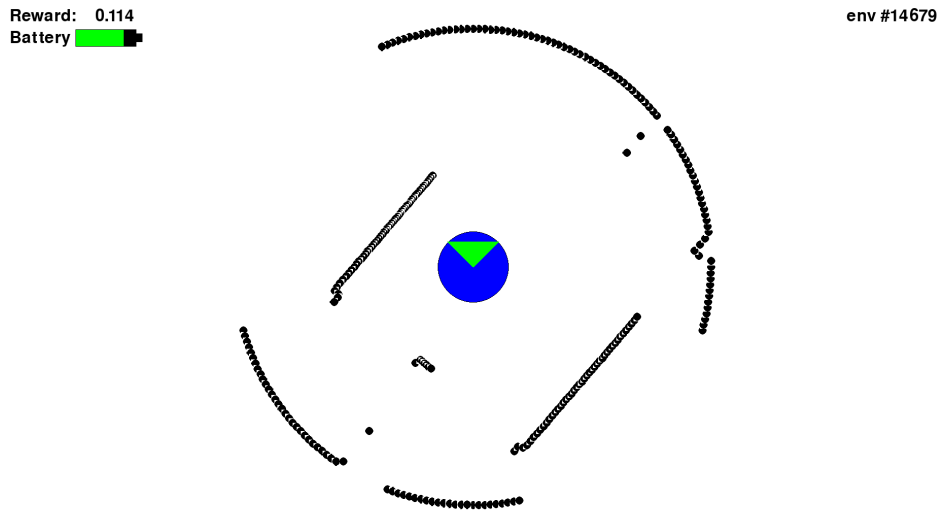


Figure 6.2: Point cloud of the surroundings captured by a laser sensor.

at a time. To enable the incorporation of Megastep-based environments into all RL algorithms, two potential solutions arose.

The initial and comparatively simpler solution is to serialize the simulation environment’s output and transfer it to the CPU. Nevertheless, this approach is inefficient and fails to utilize the full potential of the Megastep simulation engine, resulting in a bottleneck when moving data between the CPU and GPU.

The alternative solution that was implemented involves creating multiple wrappers for the environment to preserve the vectorized character of the simulation output. The primary wrapper, based on OpenAI Gym, performs the vital function of converting the SAR environment’s observations into more manageable tensors. This wrapper has several other features, including the ability to render frames of a single environment using its index, normalizing the observations and converting laser readings from polar to Cartesian coordinates.

A further wrapper is responsible for transforming the policy’s action tensors into a Megastep-compatible class. Additionally, the last crucial wrapper is responsible for simultaneously recording information for all environments, such as episodic length and reward.

6.1.2 Issues Fixed

The original state of the simulation environment was plagued by several bugs that had a detrimental impact on the training process.

The foremost and critical issue pertained to the spawning of agents on top of other objects, such as walls and hazards, as depicted in figure 6.3. This defect resulted in either immediate penalization due to collisions or restricted agent movement in any direction, adversely affecting the training process.

The underlying mechanism involves representing the inside of the building and walls through the use of a mask, whereby the agent must respawn outside this mask. To address the spawning issue, the walls’ masks were morphologically dilated to increase their area, ensuring that the agent would respawn sufficiently far away from both the walls and any hazards that were in close proximity to the walls.

Likewise, the target objects suffered from random and inconsistent respawn locations, often appearing on top of each other or walls. To mitigate this issue, the walls’ masks were again morphologically dilated and the respawn locations were limited to the convex hull of the remaining point cloud. As a result, the target objects were more likely to respawn in the center of the rooms, farther apart from each other and motivating the agent to explore and navigate more efficiently within the building.

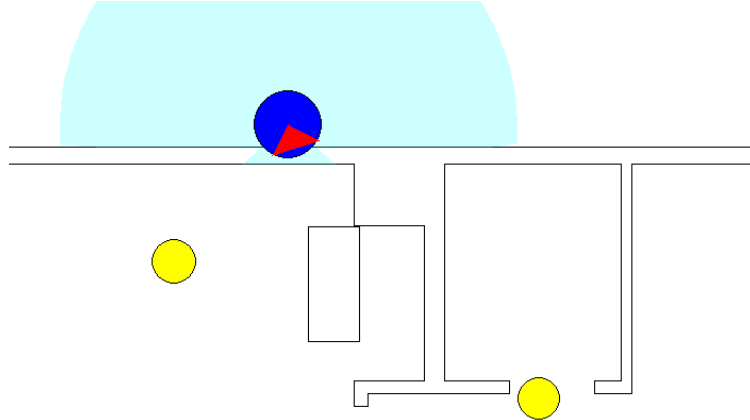


Figure 6.3: Illustration showing an example of a bad spawn on the top of a wall.

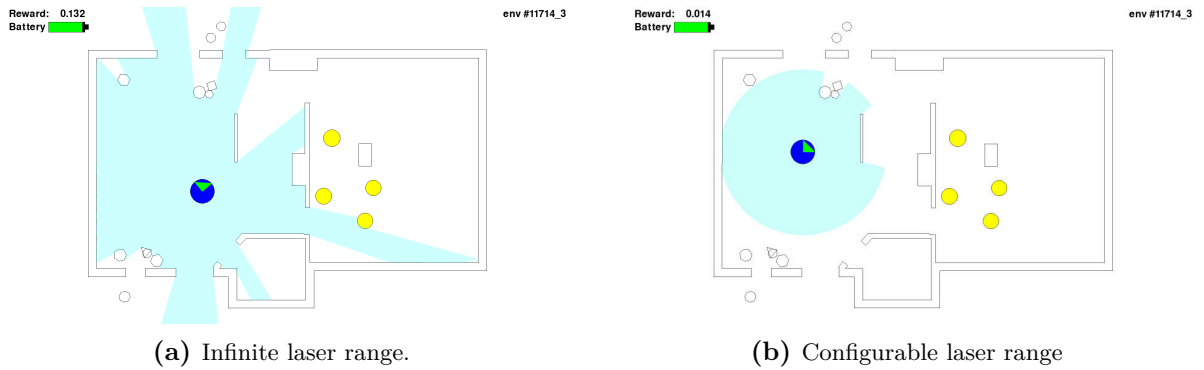


Figure 6.4: Comparison of the laser range before and after fixing an associated bug.

A further issue was that in the initial implementation, if the agent moved too far away from the building, a reset condition was triggered and the agent would be penalized accordingly. However, this condition was not reset when starting a new episode, resulting in the new episode terminating prematurely and potentially impacting learning negatively. Rectifying this issue was instrumental in improving performance.

Finally, the laser sensor functions by emitting laser rays, which bounce off objects to allow the agent to measure their distance. The maximum range of these rays was intended to be adjustable by setting a variable in the environment's class. However, in the initial implementation, the agent was able to detect target objects even beyond the laser range limit. This issue resulted in incorrect behaviors such as the agent standing still outside the building, where it could still detect target objects or not exploring inner rooms due to the same reason. By addressing this issue, the laser range became conveniently configurable through the configuration file, as depicted in figure 6.4.

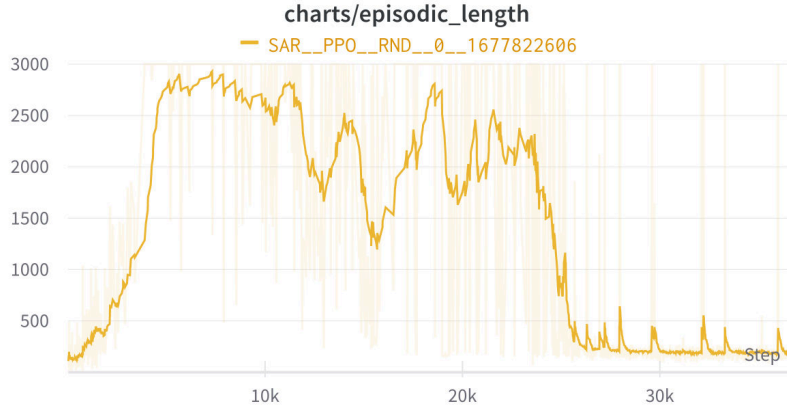


Figure 6.5: After solving an environment, the agent tends to reduce the episodic length to reduce the total penalty on taking actions.

6.1.3 Additional Features

The first minor modification was made by incorporating dots per inch (DPI) and figure size variables into the environment’s class. This allows for convenient adjustment of the size of recorded video frames from a single configuration file.

Additionally, observation variables typically include noise, but the standard deviation of this noise was previously hard-coded within the environment. Making this standard deviation configurable from a separate configuration file, has made it more convenient to adjust the noise level as needed.

Moreover, the SAR environment included initially a battery state among other observations, but this proved computationally expensive and led to inconsistent values that disrupted training. To address this, the battery class was removed and replaced with a step penalty, which incentivizes the agent to solve the environment as quickly as possible with minimal penalties. Analysis shows that after finding all target objects, the agent tends to take the shortest path towards the end of an episode, as shown in figure 6.5.

Furthermore, in order to train an off-policy algorithm, it is necessary to preserve transition tuples consisting of the current state s_t , the chosen action a_t , the subsequent state s_{t+1} , the reward received r_t and the termination indicator d_t in a designated replay buffer. However, it is worth noting that the final step of an episode does not signify the actual termination of the process, but rather the first state of the following episode, as demonstrated in figure 6.6. As such, to mitigate this predicament, the terminal state was incorporated into the system as supplementary information, which serves to supply the accurate following state in the final transition tuple of an episode.

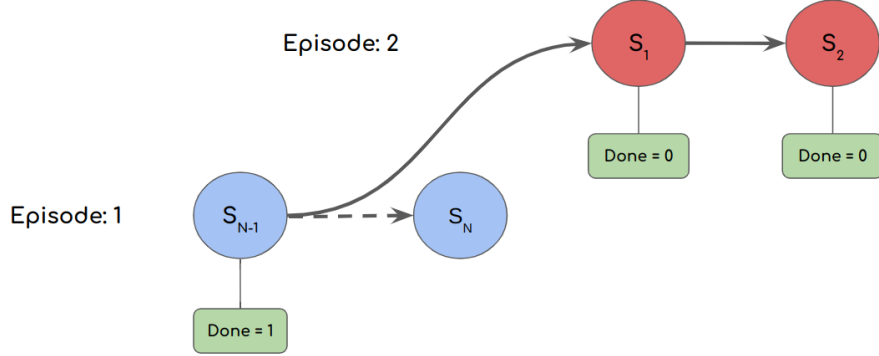


Figure 6.6: Illustration showing the states when the terminal condition is set to true.

Apart from the changes above, in a non-episodic environment, the agent is required to locate all objects or collide before exiting an episode, which may lead to instances where the agent remains in uninformative states (e.g. stationary) that could adversely affect the training process. To overcome this issue, an episode limit is introduced, whereby the environment resets automatically once the limit is reached. However, implementing an episodic limit violates the Markovian property of the environment, thereby resulting in incorrect Bellman target calculations for off-policy algorithms. As a solution, the time-limit termination criterion is retained and exceeding the time-limit is regarded as non-terminating for off-policy algorithms.

A further limitation of the initial SAR environment was its large action space. When dealing with large action spaces, deterministic algorithms such as DQN and Rainbow can be adversely impacted, as they rely on learning the value of each action in all possible states to train effectively. The curse of dimensionality arises when the action space is significantly large, making it increasingly difficult to explore the effects of each action in all possible states, including neighboring and similar ones. To mitigate this issue, the action space was simplified by minimizing it into only seven discrete actions, each of which represents a single move in one direction instead of an action space consisting 81 discrete actions consisting of multiple moves, thus facilitating the exploration of the effects of each action in all possible states.

Moreover, one downside of the simulation environment is that it provides sparse rewards, which hinders the agents' ability to learn their task effectively, especially when they lack any prior knowledge of the environment. Consequently, the agents were exhibiting peculiar behaviors such as standing still or moving aimlessly outside the building. To address this issue, a reward was added to incentivize the agent to enter the building's area. This modification yielded positive results, as the agent was observed to be more drawn towards entering the building. This, in turn, increased

the chances of the agent locating the target objects, thus improving its performance.

Finally, the simulation was initially configured to run multiple disparate environments (i.e. floorplans) simultaneously, with the aim of training the agent on a diverse set of scenarios to improve its generalization capabilities. However, given the sparsely-rewarded nature of the environment and the agent’s limited perception of the world, exploration proved challenging. Specifically, the agent’s limited sensor range and the similarity of observations across different floorplans meant that exploration algorithms were susceptible to confusion, with rewards being given in one state and not in a similar state in a different floorplan. Furthermore, parameters such as *gamma* are highly environment dependant. Bigger environment require bigger γ values to address bigger episodic lengths and vice versa. Therefore it is sub-optimal to train on different environments using a single gamma value. To address this, we simplified the task to training on a single floorplan and reduced the agent’s sensor range to create a more local perception of the environment.

6.2 Baselines

The primary objective of this thesis is to conduct a comparative analysis of state-of-the-art RL algorithms. The study commenced by modifying the Stable Baselines 3 library [46] to suit the SAR environment. This involved relocating all buffers to the GPU and adjusting the algorithms and policy to manage the output of the environment, which consists of dictionary classes containing GPU tensors. Although the library was functional and capable of executing the algorithms, it was deemed impractical to proceed with it owing to its intricate modular design and lack of several algorithms. These shortcomings made it challenging to debug and integrate additional functionalities such as exploration, recurrence or stacking observation.

CleanRL [47] stood out as a more suitable open-source RL library for the purpose of this thesis. It offers a unique paradigm in which each algorithm variant is implemented in a self-contained single-file. This approach simplifies the implementation by reducing the complexity of the codebase, making it more approachable and easier to comprehend. Additionally, this methodology allows for the fine-grained control over all aspects of the algorithm in a single file, facilitating the rapid prototyping of novel features. Although not all algorithms are implemented in CleanRL, its uncomplicated paradigm allowed for easy integration of additional algorithms and features, making it a valuable tool for the study and comparison of state-of-the-art RL algorithms.

6.2.1 Algorithms

The CleanRL library offers a comprehensive implementation of a wide range of RL algorithms, including PPO (section 3.1.3), DQN (section 3.2.1) and SAC (section 3.2.3). However, this library does not include all RL algorithms currently available. In this study, we aimed to expand the library by implementing DreamerV3 (section 4.1), A2C (section 3.1.2) and Rainbow (section 3.2.2), while maintaining the same implementation paradigm as CleanRL. Although a single-file implementation is ideal for prototyping, it leads to redundancy in the code by repeating multiple parts across multiple files. Thus, we improved the library by modularizing common elements, such as preprocessors, wrappers, configurations, distributions, buffers and utilities.

The original implementation of Rainbow algorithm used the C51 variant of the categorical distribution [48], which relies on a pre-determined number of categories defined by equidistant values. However, this approach requires extensive knowledge of the environment’s hyperparameters, such as the discount factor, as constraining the distribution within certain limits can negatively impact the neural network’s performance if the actual values exceed these limits. To overcome this issue, this work proposes the use of the Implicit Quantile Network (IQN) [49] that utilizes a categorical distribution based on percentile calculations. This approach allows the network to independently calculate the categories with the number of categories corresponding to the number of percentiles calculated, without any constraints on predefined values.

In addition, the hard exploration problem, which arises in sparsely rewarded environments with limited feedback on the agent’s actions, was addressed by implementing two advanced exploration algorithms in the library. These algorithms, RND (section 5.2) and ICM (section 5.1), reward the agent intrinsically for exploring new areas within the environment. To facilitate integration into any algorithm, these algorithms were implemented as modular components, encapsulating the network, optimizer, buffers, training and logging methods and intrinsic reward calculation procedures within a single class.

Initially, the intrinsic reward was simply added to the reward provided by the environment and the policies were trained in the usual manner. However, this approach was found to be flawed and resulted in poor performance. Intrinsic rewards differ from extrinsic rewards in that they are non-episodic, so the value function or Q function where exploration is integrated should estimate the extrinsic and intrinsic values separately, with these two heads trained independently using different target values. Unlike the target values for the extrinsic head, the bootstrapped values used to train the intrinsic head are not multiplied by the terminate flag. Moreover, since different environments require different hyperparameter tuning, this issue was partially mitigated by normalizing both the inputs of the exploration algorithm and the intrinsic reward using exponential moving average (EMA). These approaches resulted in consistent rewards provided by RND.

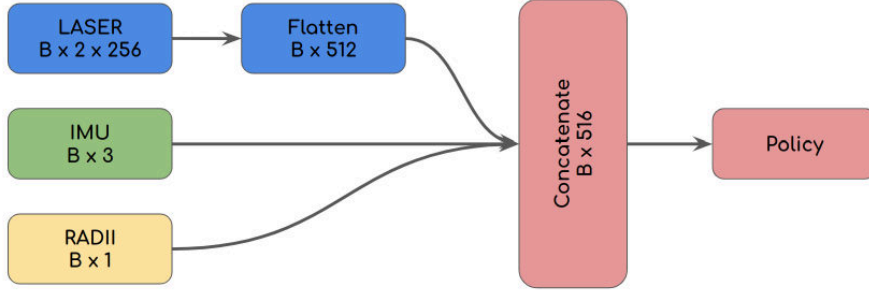


Figure 6.7: An illustration showing processing the SAR environment’s observations by flattening the laser readings and concatenating it with IMU and radii data.

In order to improve the performance of off-policy algorithms, a Polyak update [50] was incorporated into the implementation of all such algorithms. This update involves smoothly updating the target network by exponentially averaging over multiple steps, rather than simply substituting the weights of the target networks. This approach results in more stable and accurate training, especially when dealing with environments with non-stationary dynamics.

6.2.2 Buffers

CleanRL utilizes existing buffers from Stable Baselines 3, namely replay buffers and rollout buffers, which only support CPU tensors. This may result in a performance bottleneck if the observations are constantly converted between GPU and CPU. Furthermore, these buffers are not designed to handle dictionary observations, which are used in SAR. Therefore, in this study, several SAR-compatible buffers were developed for the different RL algorithms employed. These include a rollout buffer for on-policy algorithms, an experience replay buffer for off-policy algorithms, an n-step return and prioritized experience replay buffer for Rainbow algorithm and an episodic buffer for Dreamer algorithm. All of these buffers are also capable of handling vectorized environments, which allows for the simultaneous training of multiple environments and accelerates the training process.

6.2.3 Data Preprocessors

In cases where observations consist of multiple tensors, it is common practice to feed them into the policy’s network by converting each tensor into a single vector and then concatenating all of these vectors together. This process can be illustrated in figure 6.7.

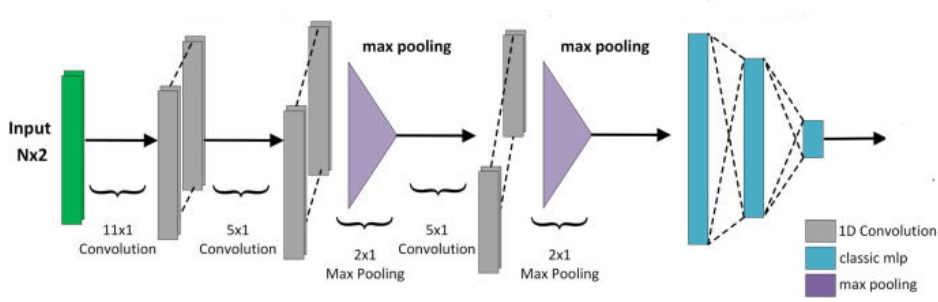


Figure 6.8: Similarly to AlexNet, 2DLaserNet consists of convolutional layers followed by batch normalization, activation layer and max pooling. [51, p. 7]

This approach did not yield satisfactory results in the SAR environment. This can be attributed to the diverse nature of the observation data, where laser observations are raster data similar to images, while IMU and radii data are vector data. As the high dimensionality of raster data makes it computationally expensive to process using multi-layer perceptron (MLP) and the performance was found to be suboptimal in practice, a new preprocessor was implemented to handle the observations. The first preprocessor separates all observation tensors from each other and laser observations are processed using a convolutional neural network, while IMU and radii observations are processed using conventional MLPs. The outputs from these processors are then concatenated and fed into the policy network.

To pre-process the laser data, the 2DLaserNet [51] architecture was initially adopted, leveraging multiple stages of 1D convolutional layers, subsequently followed by batch normalization and an activation layer. This multi-stage architecture aligns with the fundamental principles of AlexNet [52], which was one of the first most successful and influential networks in computer vision. However, given that the laser data is inherently in polar form and originates from the environment, it requires preliminary conversion into a point cloud in Cartesian coordinates to align with the processing pipeline of 2DLaserNet. This preprocessing step enables effective utilization of the architecture’s inherent processing capabilities, as demonstrated in figure 6.8.

In view of the resemblance of 2DLaserNet to the well-established AlexNet and the recognition of the advancements that residual connections introduced, as demonstrated in ResNet [53], the 2DLaserNet was modified by us via the inclusion of residual connections. Subsequently, the network underwent further refinement through the incorporation of channel 6.9a and spatial attention gates 6.9b, as exemplified by the convolutional block attention module (CBAM) [54]. Convolutional neural networks (CNNs) primarily learn the weights of kernels that capture distinctive features of a raster data; given that some features have greater relevance than others in a given context, the inclusion of channel attention gates was deemed necessary. Spatial attention gates are equally significant, given that the importance of objects in a scene can differ significantly based on their positional

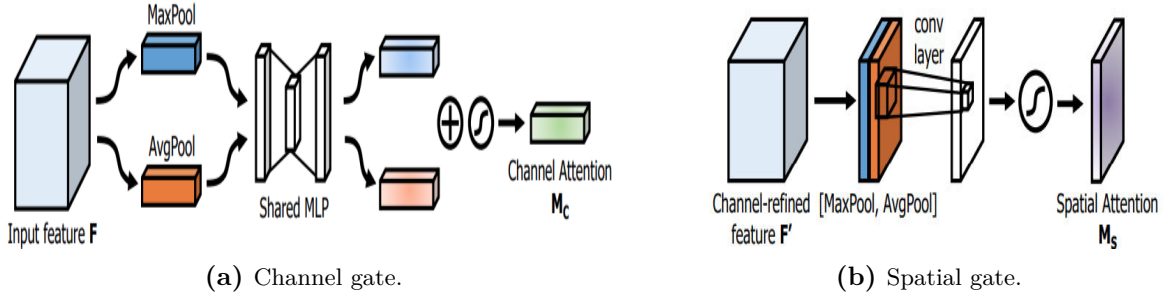


Figure 6.9: Overview of spatial and channel gates of CBAM. [54, p. 5]

arrangement. Consider for example a situation comprising a self-driving and a pedestrian. The pedestrian moving on the pavement would not warrant an application of brakes; however, if the pedestrian is located in the middle of the road, it would necessitate the car to come to a stop. Spatial gates operate similarly to the channel gates by applying a weight to the positions of the raster data across all channels, contingent on their significance.

6.2.4 Recurrence

The SAR environment, as depicted, exhibits a severely limited perception of the surrounding world. Specifically, the agent can only perceive objects within its laser range, lacks awareness of its location and the target's location or identity and has no information regarding its prior movements. For example, upon discovering the first target object, the agent cannot determine whether to pursue the second target or revisit the first target due to the absence of information about the past. These constraints render the environment partially observed and designate the state action transition process as a partially observed Markovian decision process (POMDP). To overcome this challenge, the agent must efficiently leverage all available resources at its disposal.

In order to overcome the challenge mentioned earlier, a policy was initially developed that utilized not only the current observation but also a history of $n - 1$ past observations. The processing of these observations can be accomplished through various methods, ranging from convolutional networks to individual processing followed by a recurrent network such as a gated recurrent unit (GRU) [55] or long short-term memory (LSTM) [56].

Subsequently, the recurrent preprocessor was upgraded to a two-layered GRU model. This was motivated by the computational efficiency of GRU compared to LSTM and its superior performance. The first layer of the GRU model processes the laser data, while the hidden states generated by this layer are concatenated with the IMU and radii data and passed through the second GRU layer, as illustrated in figure 6.10. It is worth noting that the laser sensor captures only one side of objects in

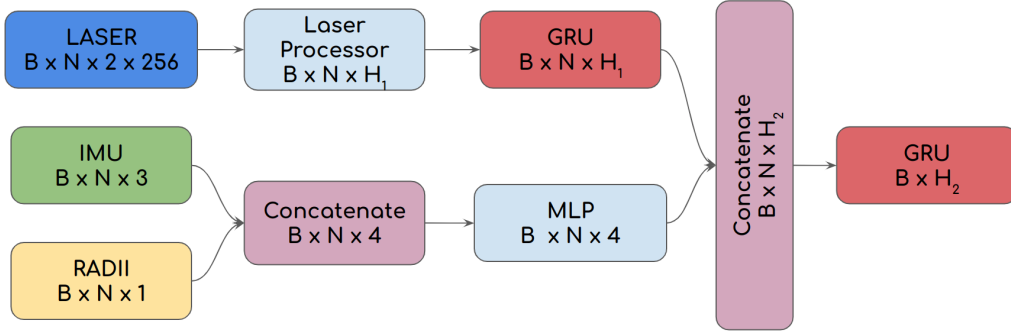


Figure 6.10: Illustration showing the neural network used in this work to process the SAR environment observations.

the environment. By utilizing a recurrent network and segregating the laser processing, the agent can scrutinize the objects captured by the laser sensor from multiple angles, leading to improved object classification in the environment.

Therefore, in this work, an approach was proposed to handle the challenge posed by POMDPs. Specifically, a wrapper was developed that could deliver a stack of n observations using a double-edged queue, which was then processed using the previously described recurrent preprocessor. This method resulted in a significant improvement in the agents' movement capabilities. However, this approach also had some limitations.

One of the main drawbacks of the proposed approach was its high memory consumption, which increased by a factor of n equal to the number of frames stacked. This was particularly problematic for off-policy algorithms. To address this issue, we exploited the fact that the experience replay buffer was a ring buffer that maintained the correct order of observations and separated observations from different environments. As a result, the buffers were adapted to deliver n frames, which allowed us to train the recurrent units on even longer sequences while minimizing memory usage. By leveraging this adaptation, we were able to significantly reduce the memory overhead associated with stacking frames.

Another downside of the frame-stacking approach was its computational inefficiency. Specifically, the policy on each training step needed to be executed n times to train the recurrent net on a sequence of n steps. Additionally, the last hidden state of the sequence was deprecated and not exploited to train the next sequence. Instead, an initial hidden state of zeros was used and the agent's memory was limited to only $n - 1$ steps in the past.

To address the limitations of the frame-stacking approach on on-policy algorithms, we saved the

hidden states after each step and used them to both take actions of the next step and train the recurrent unit. This approach successfully solved the SAR environment. However, this solution is not applicable to off-policy algorithms. The hidden states saved in the experience replay buffer differ each time the recurrent unit is updated, making it necessary to re-calculate all hidden states in the buffer after each policy update. This process is computationally infeasible, limiting the use of this approach for off-policy algorithms.

6.2.5 Transfer Learning

At the start of the training process, the agent is randomly initialized and has no knowledge of its position, the target’s position, what the target is or even how to obtain the target. Additionally, the agent receives limited feedback from the environment, making it difficult to evaluate the quality of its actions. One way to address this challenge is through transfer learning, which is a popular research direction that has been used to train Dota agents in OpenAI Five [3] and achieve an impressive superhuman score in difficult environments such as Montezuma’s Revenge [57].

Transfer learning involves leveraging the knowledge and experience of pre-trained agents to solve a different or similar task by fine-tuning the agent using conventional RL techniques on the new task. This differs from imitation learning, which aims to replicate the behavior of another agent by training the agent on a dataset of actions produced by that agent using supervised learning techniques. However, imitation learning is often challenging and less robust than transfer learning, as even small changes in actions can result in significant shifts in the state distribution, a problem known as distributional shift [58].

In this study, a wrapper was developed to gather state-action pairs from the environment. An agent was then pre-trained to mimic the actions in this dataset. After that, several agents were initialized using the pre-trained agent’s weights and fine-tuned with various RL algorithms. This strategy is only feasible for policy gradient approaches since the policy is explicit and not implicit, unlike in value-based methods, where computing Q-values for all actions in all states is necessary to choose the right actions. Although conservative Q-learning (CQL) [59] can be employed to assess the Q values of an offline dataset, offline techniques are beyond the scope of this research.

7 Experiments

To compare the various RL methods, we conducted a series of experiments in a simulated SAR environment, which were divided into three sets. The first set focused on testing the ability of different approaches to solve the SAR environment. The second set aimed to compare the effects of modifications added to all policies. The third set demonstrated how RL algorithms can be sensitive to parameters. During all experiments, we had to determine the best hyperparameters through empirical experimentation due to the inherent differences among the approaches being examined and the computational complexity of training individual agents in the SAR environment.

7.1 Algorithms

The experiment aimed to test and compare different RL approaches, including their performance in comparison to human performance. The evaluation was conducted rigorously by subjecting each methodology to 50 episodes, with the agent’s starting position varied across episodes. During each episode, the agent’s trajectory, the corresponding reward earned and the duration were recorded. The results of the experiment are presented in table 7.1.

	Reward		Length		Reward per Step	Steps per Object
	Min/Max	Mean±Std	Min/Max	Mean±Std		
PPO + RND	-2.622/14.795	13.327±4.699	40/187	109.120±26.476	0.116	38.261
A2C + RND	-4.497/-0.230	-2.797±0.495	12/799	138.960±133.960	-0.042	-
A2C + TL	13.707/14.764	14.657±0.151	94/517	136.980±60.603	0.116	45.660
DQN + RND	-6.095/-1.250	-2.864± 0.838	5/1500	205.64±413.958	-0.121	-
SAC + RND	-3.75/-0.635	-2.120±0.903	14/1500	494.84±324.042	-0.014	-
Rainbow	-3.750/-3.750	-3.750±0.000	1500/1500	1500±0.000	-0.002	-
DreamerV3	-2.702/-2.507	-2.544±0.040	3/81	17.7±16.082	-0.266	-
Human Operator	-2.722/14.595	9.744±7.354	34/472	222.320±120.809	0.047	91.122

Table 7.1: Comparison of RL different approaches performance metrics.

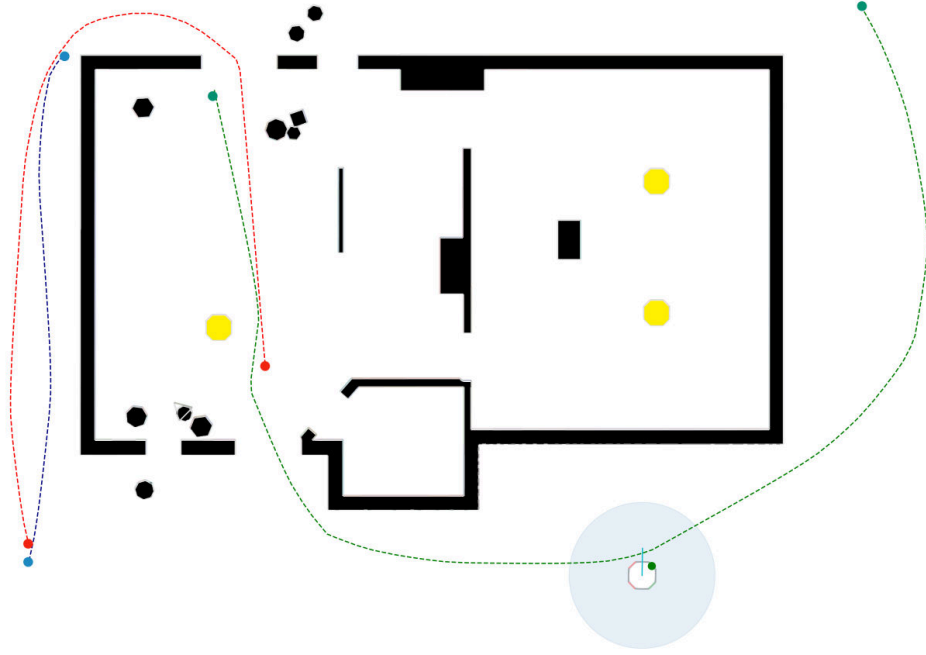


Figure 7.1: Trajectories of RL agents trained using PPO and RND on the SAR environment.

7.1.1 PPO + RND

The PPO agent underwent training for 50 million steps, utilizing a discount factor of 0.991, a GAE lambda of 0.95, an entropy regularization factor of 0.001, and a surrogate clip coefficient epsilon of 0.162. During each iteration, the agent collected 128 consecutive steps from the environment and was trained on this batch for four epochs. Additionally, we employed the RND as an exploration algorithm, with an output size of 16, a curiosity coefficient of 0.33, and a non-episodic discount factor of 0.975. Notably, this approach enabled the agent to successfully navigate the environment while minimizing step penalties by maintaining a small episodic length. Furthermore, the agent was able to traverse optimal paths, as illustrated in figure 7.1.

7.1.2 A2C + RND

We trained the A2C agent for a total of 165 million steps, employing a discount factor γ of 0.992, a GAE λ of 0.95, and an entropy regularization coefficient of 0.001. Additionally, we used RND technique as an exploration algorithm, with a contribution factor of 0.5 towards the overall advantage,

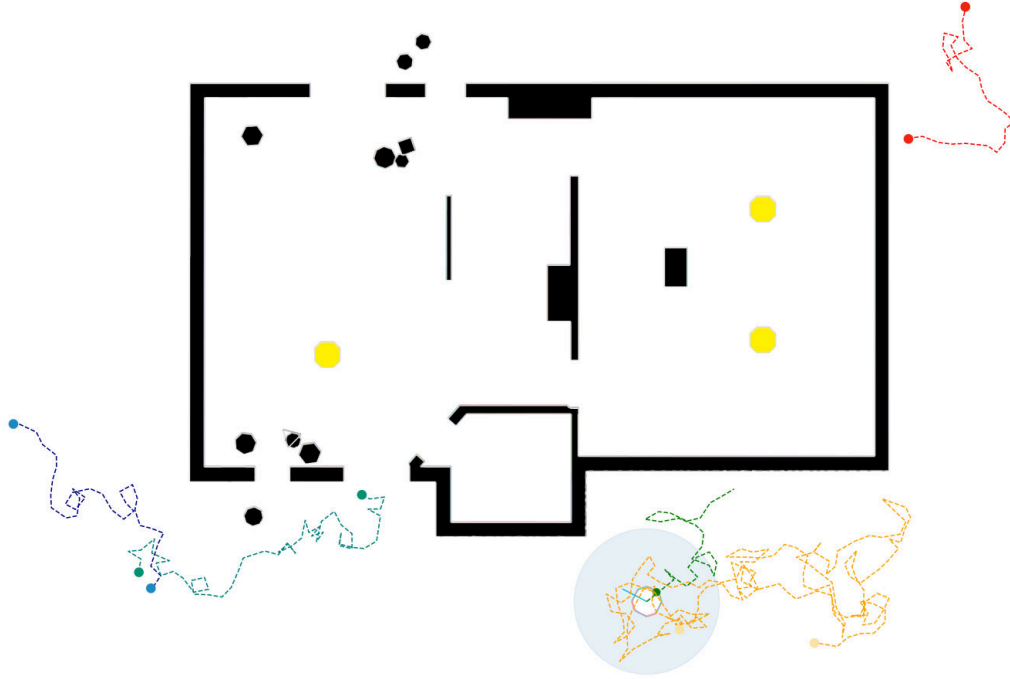


Figure 7.2: Trajectories of RL agents trained using A2C and RND on the SAR environment.

and a non-episodic discount factor of 0.95. However, this approach resulted in suboptimal scores, with the agent seemingly following non-straight paths, as depicted in figure 7.2.

7.1.3 A2C + Transfer Learning

In this particular approach, we employed an A2C algorithm to fine-tune an agent that had previously been trained on a dataset using supervised learning regression. The A2C algorithm had a relatively low learning rate of $2.5e-5$ to ensure convergence to the global peak and we utilized a discount factor γ of 0.99, and a GAE λ of 0.95. This approach yielded an agent that successfully solved the environment, as demonstrated in the corresponding table, while maintaining a low episodic length to avoid incurring step penalties. Additionally, figure 7.3 provides examples of the different trajectories that the agent was able to traverse.

7.1.4 DQN + RND

We trained a DQN agent for 32 million steps on a replay buffer consisting of 250 thousand. In this training, we set the discount factor γ to be 0.995 and trained the DQN on a stack of 16 frames.

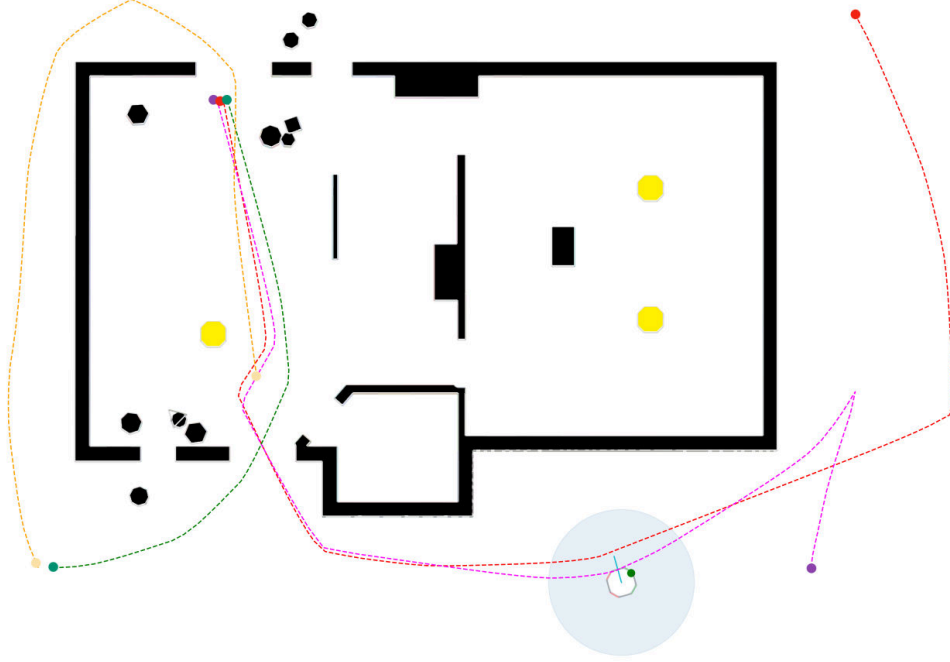


Figure 7.3: Trajectories of RL agents fine-tuned using A2C on the SAR environment.

The online network was trained on a batch of size 512 every 16 steps, while the target network was updated every 32 thousand steps. To explore the environment, we also employed RND as an exploration algorithm with a discount factor of 0.925 and curiosity factor of 0.1. However, this approach was unable to solve the environment and locate the target objects, as evidenced by the corresponding table. Also, the agent was not able to traverse optimal paths, as illustrated in figure 7.4.

7.1.5 SAC + RND

In this approach, we trained an agent using the SAC algorithm for 30 million steps, while also having an experience replay buffer of size 250 thousand to store and learn from past experiences. The discount factor, which determines the importance of future rewards, was set to 0.995, and the networks were trained on a sequence of 16 observations. To encourage exploration, the agent was also equipped with RND, with a discount factor of 0.95, and an intrinsic reward coefficient of 0.05.

The SAC algorithm is designed to maintain a high level of entropy in the actions taken by the agent, which is important for continuous action spaces. However, maintaining such high entropy in a discrete setting can result in random movements. To avoid this, we linearly regulated the entropy

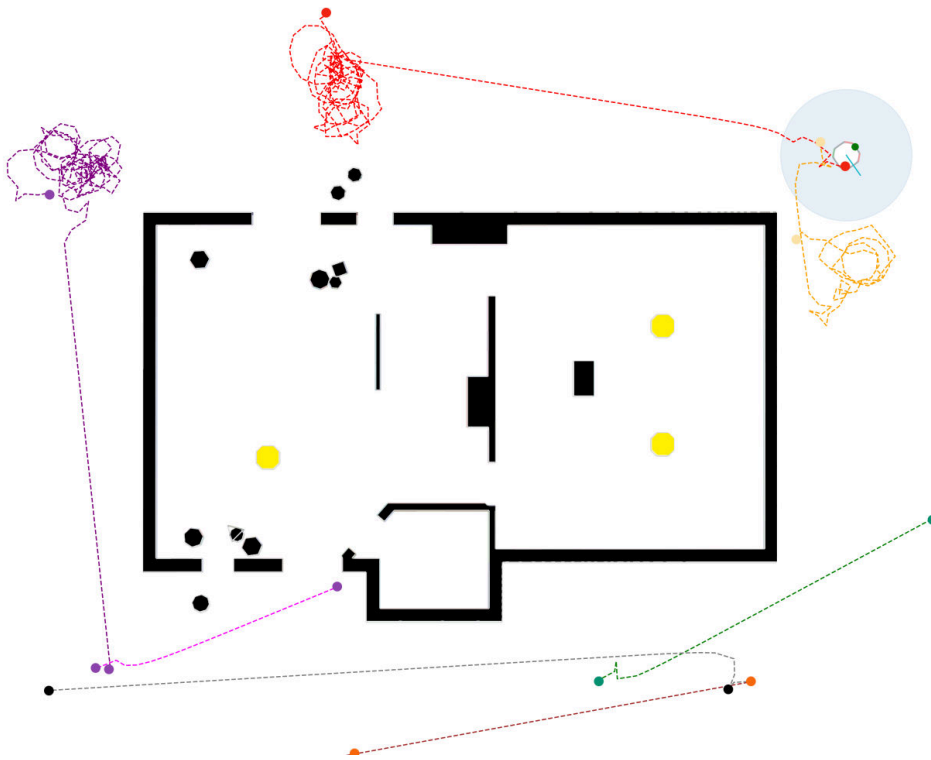


Figure 7.4: Trajectories of RL agents trained using DQN and RND on the SAR environment.

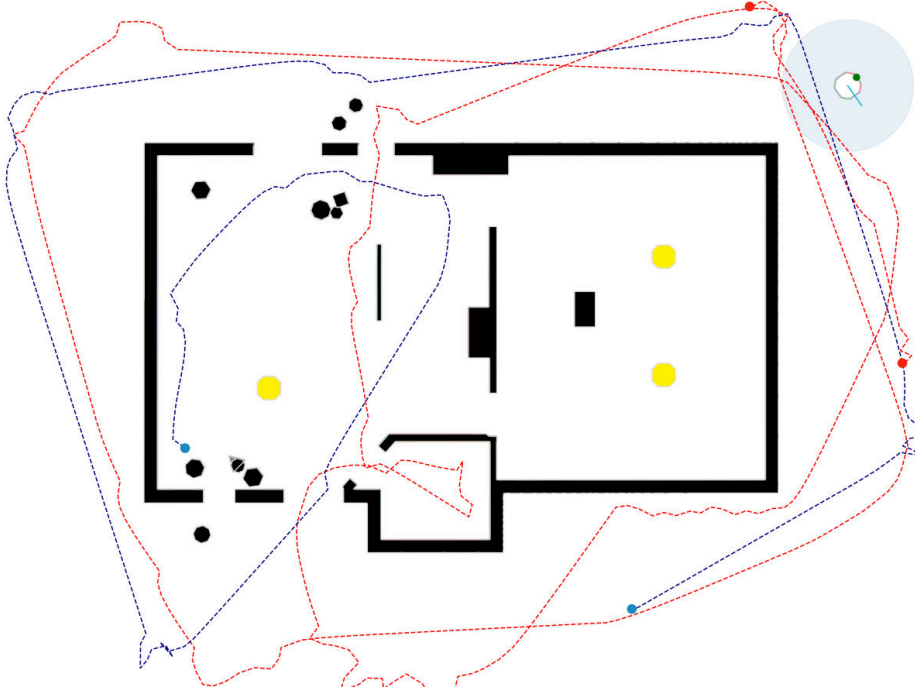


Figure 7.5: Trajectories of RL agents trained using SAC and RND on the SAR environment.

coefficient to keep the actor entropy at 0.55 of the maximum value. While this approach did not result in optimal reward gaining, the agent demonstrated good movement by avoiding collisions with objects and moving in straight lines around the building, as well as traversing around the buildings, as shown in figure 7.5.

7.1.6 Rainbow

The Rainbow agent was trained for 5 million steps using a replay buffer of size 250 thousand, and a discount factor of 0.991 on sequences of 16 frames. Instead of using C51, the agent was trained with IQN and the number of percentiles was set to 8. However, this approach led to the agent getting stuck in a stationary state, neither exploring nor colliding with objects, which resulted in poor performance. Furthermore, the agent did not move and showed a stationary behavior as shown in the figure 7.6.

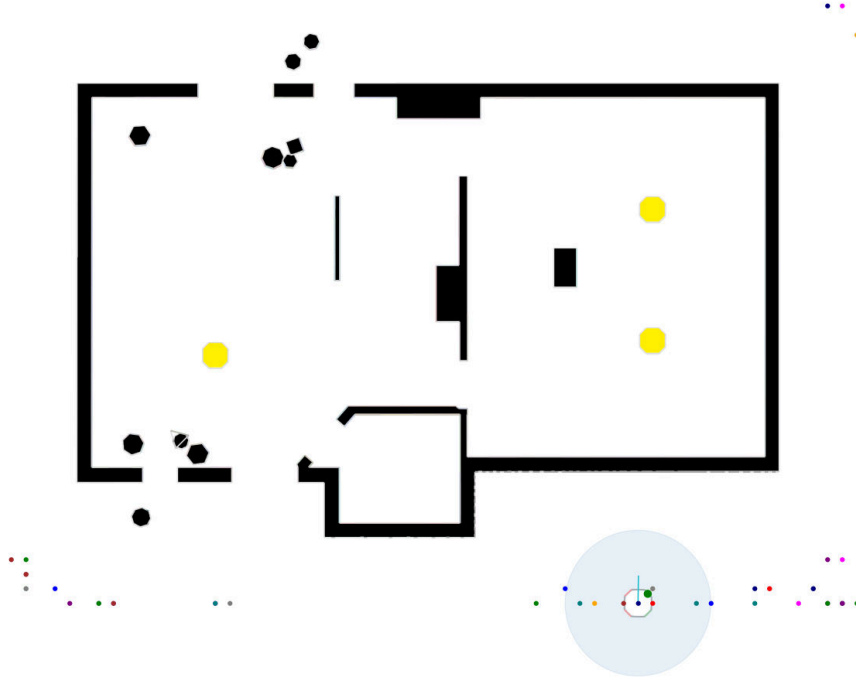


Figure 7.6: Positions of RL agents trained using Rainbow on the SAR environment.

7.1.7 DreamerV3

The algorithm chosen for this experiment was DreamerV3, which is a model-based approach that encompasses various techniques such as training a world model and using the learned latent state representation to train the actor-critic learner. The hyperparameters used were adapted from the original paper, which were designed to be suitable for a wide range of environments. In this experiment, the environment was set up to not penalize the agent for colliding with objects or walls, as penalization led to the agent becoming stationary, which limited its ability to explore and the world model’s ability to learn on a diverse range of states. Despite these efforts, the agent’s behavior was still suboptimal and it mainly traversed short random paths, as shown in figure 7.7.

7.1.8 Human Operator

In this experiment, actions were taken by a human operator given only the same laser observations that are given to the RL agents. The evaluation of this experiment conversely to others, was conducted on only 25 episodes. Figure 7.8 shows samples from the trajectory traversed by the UAV system.

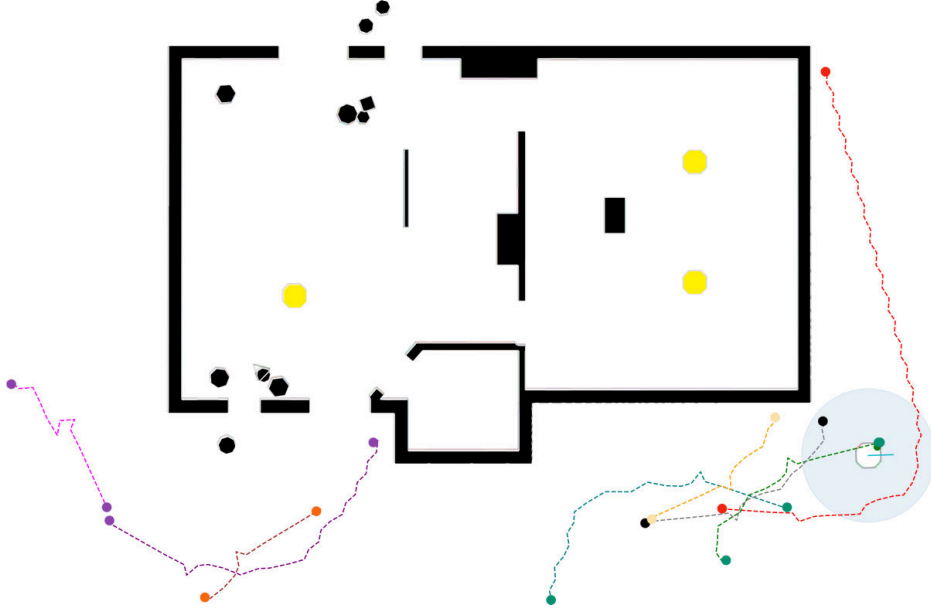


Figure 7.7: Trajectories of RL agents trained using DreamerV3 on the SAR environment.

7.2 Modifications

In this experiment, we compared modifications made to all preprocessors on PPO exclusively. All experiments were conducted using the same set of hyperparameters, environment, seed number, and number of neurons and layers. For this experiment, we set the discount factor γ to 0.991, the GAE λ to 0.95, and the clip fraction ϵ to 0.162. All experiments were trained on 30 million frames, with an annealing learning rate of 0.004. We also used RND as the exploration approach, with a discount factor of 0.975, and curiosity coefficient of 0.33. The convergence of different approaches is shown in charts 7.9.

Additionally, we evaluated all different approaches on 50 episodes and almost all of them were well performing as shown in table 7.2. Figures 7.10 also show different trajectories traversed by each agent.

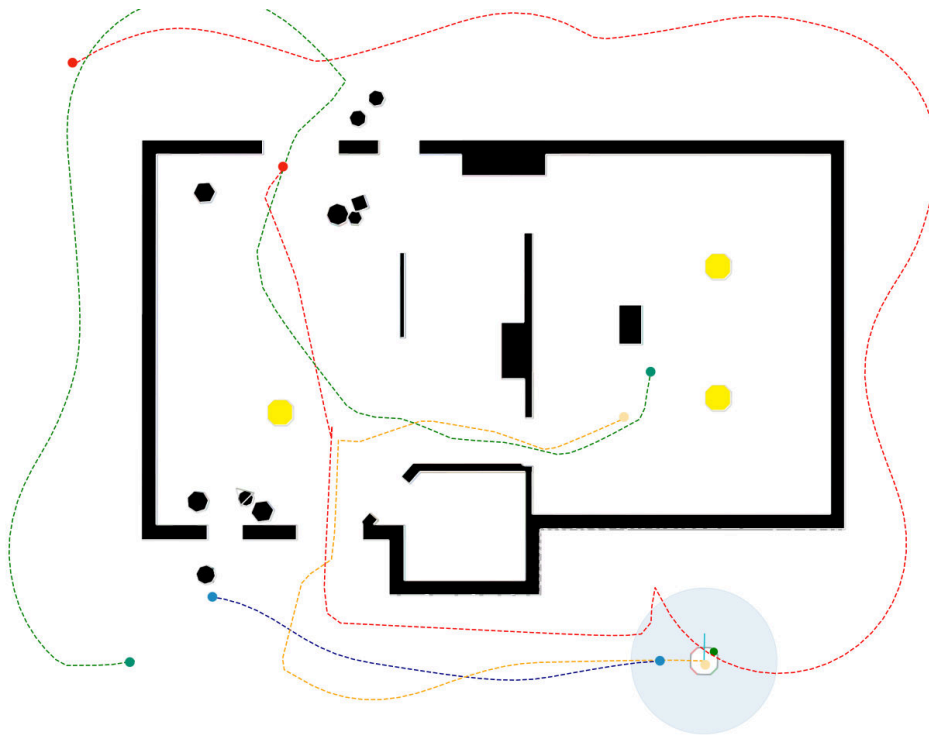


Figure 7.8: Trajectories of the UAV system by a human operator on the SAR environment.

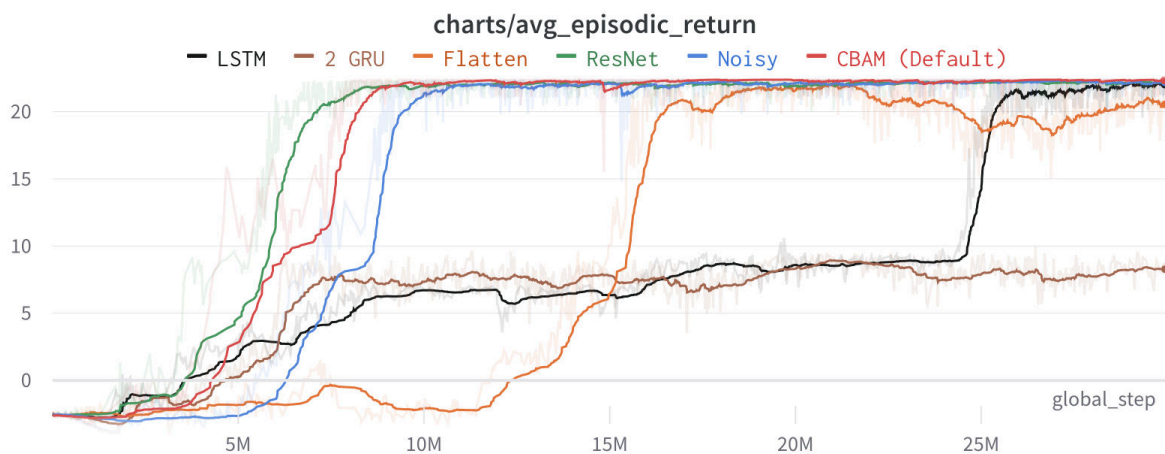


Figure 7.9: Chart showing the effect of different preprocessor modifications on reward convergence.

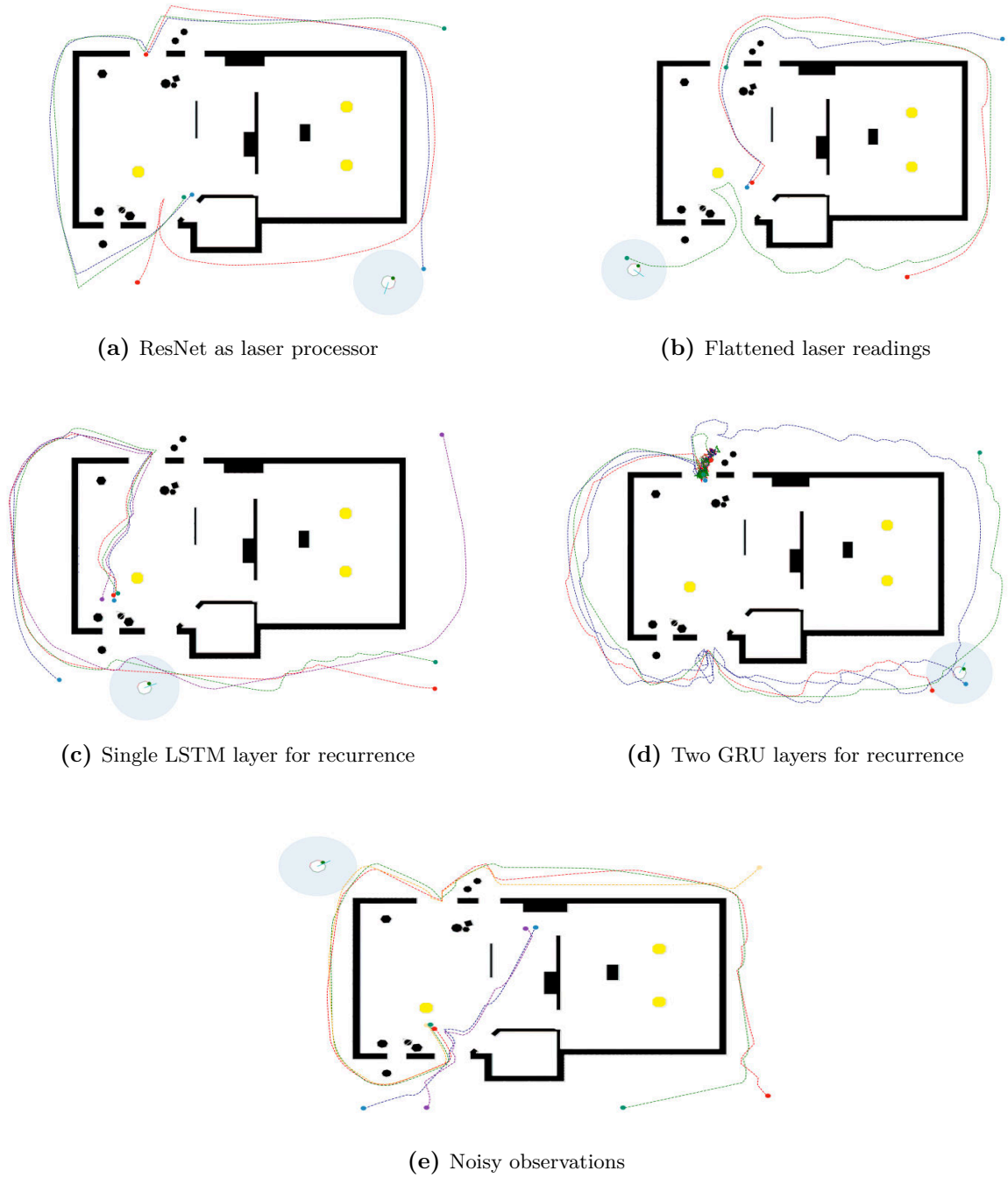


Figure 7.10: Trajectories of PPO agents with different policy implementations and observations.

	Reward		Length		Reward per Step	Steps per Object
	Min/Max	Mean $\pm$ Std	Min/Max	Mean $\pm$ Std		
ResNet	13.617/14.517	14.415 $\pm$ 0.120	193/553	234.020 $\pm$ 48.018	0.063	78.007
Flattened	11.785/14.557	14.088 $\pm$ 0.631	177/1286	364.600 $\pm$ 252.401	0.051	121.533
LSTM	13.387/14.615	14.387 $\pm$ 0.166	154/645	245.060 $\pm$ 66.490	0.061	81.686
2 GRUs	1.289/4.164	3.535 $\pm$ 0.627	334/1500	1445.68 $\pm$ 189.552	0.002	-
Noisy	14.429/14.777	14.528 $\pm$ 0.113	89/228	188.680 $\pm$ 45.163	0.083	62.9

Table 7.2: Comparison of different PPO policy implementations performance metrics.

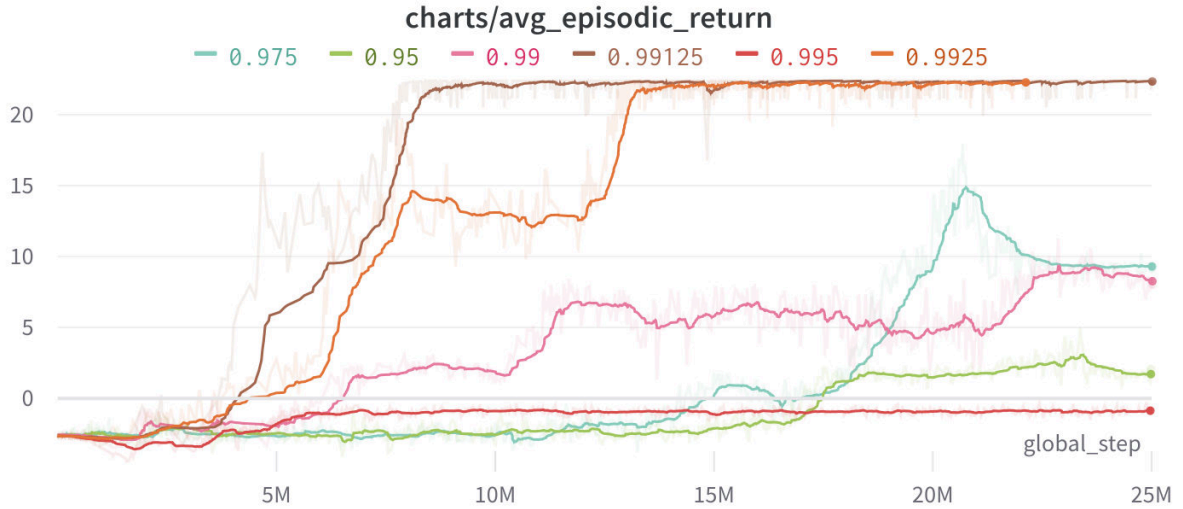


Figure 7.11: Charts showing the convergence for different PPO agents with different discount factors.

7.3 Hyperparameters

In this set of experiments we demonstrate the effect of hyperparameters on the convergence of RL agents. We trained PPO and RND as the RL and exploration methods with a fixed set of hyperparameters while altering the discount factor γ , where $\gamma \in \{0.95, 0.975, 0.99125, 0.9925, 0.995, 0.999\}$. The GAE λ , annealing learning rate, surrogate clip coefficient, and curiosity fraction were set to 0.95, 0.004, 0.1625, and 0.33 respectively. Figure 7.11 shows the reward charts for training for 25 million frames.

8 Discussion

In the previous chapter, we first conducted experiment to evaluate the efficacy of different policy-based methods in solving the SAR environment. Specifically, we compared the performance of A2C and PPO and found that PPO outperformed A2C. Policy-based algorithms typically have a policy loss that oscillates around 0, where negative values attempt to increase the probability of selecting a particular set of actions, while positive values aim to decrease the probability of choosing actions in a given batch. Our results revealed that A2C exhibited highly variable unrestricted updates, as depicted in figure 8.1a, whereas PPO updates were comparatively less variable, as illustrated in figure 8.1b. This difference in the updates' variability and overall performance can be attributed to PPO's ability to restrict parameter updates within a reliable region defined by the KL divergence, while A2C's updates may overshoot and miss global peaks or become permanently stuck.

In addition, we observed that the entropy charts of all policy gradient methods decreased, as illustrated in Figure 8.2. This indicates that the policies become more confident in the actions they take, which highlights the need for a regularization term to maintain an exploratory behavior during the early stages of training.

In an attempt to improve A2C's performance, we employed transfer learning to equip the agent with a prior belief about the environment. This approach has shown promise in enhancing agent



(a) A2C policy loss.



(b) PPO policy loss.

Figure 8.1: Policy loss of PPO agents are less variant than the unrestricted loss of the A2C agent.

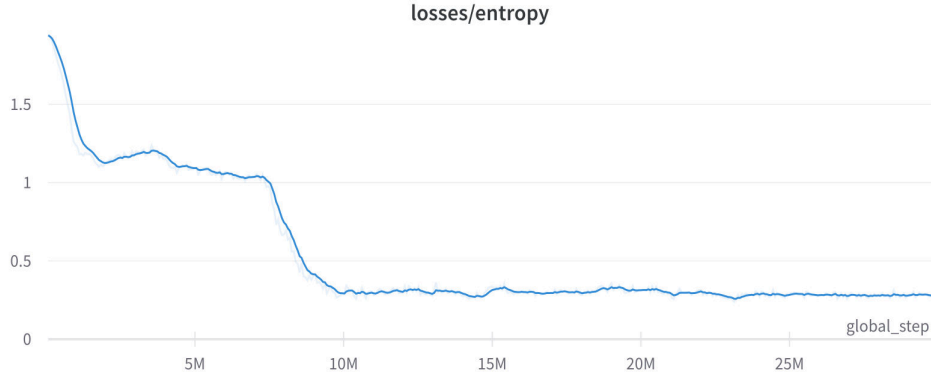


Figure 8.2: As the agent becomes more confident with the actions it takes, the entropy of on-policy methods decreases.

performance by boosting the agent’s initial performance and enabling efficient exploration and learning. By further fine-tuning the policy using the agent’s own experience, the agent becomes more resilient against distributional shifts and other issues associated with imitation learning. Overall, this approach improved the agent’s performance and robustness.

Subsequent to testing value-based approaches within the SAR setting, it was found that on-policy algorithms exhibited greater efficacy than their off-policy counterparts. Although the latter manifested non-random and optimal movements, they were not able to accomplish a successful resolution of the environment. The reason behind this limitation stems from the fact that recurrent layers within off-policy algorithms were trained on a restricted number of frames, whereas on-policy algorithms were trained on the entirety of episodes. This key difference enabled the agent to recollect past actions and thereby mitigate the inherent partial observability of the environment. In contrast, the memory of off-policy methods was restricted to the number of frames on which they were trained. To train the recurrent layers of an off-policy algorithm on episodic sequences, the hidden states must be preserved within the replay buffer. This process, however, can be computationally onerous to update after each policy iteration.

In addition, our observations revealed the anticipated occurrence of the DQN algorithm overestimating Q values as it transitions from the exploration phase to exploiting actions, which is depicted in figure 8.3.

Furthermore, all prior methodologies employed RND as an exploration technique, which displayed a typical behavior of generating low rewards for familiar locations intermixed with significantly rewarding peaks of novel observations, as illustrated in figure 8.4.

On the contrary, ICM was not incorporated in our experiments owing to its intricate computational complexity and propensity to generate intrinsic rewards that are extremely large and variable, as



Figure 8.3: DQN overestimates the values of the actions as it starts exploiting the best actions for each state.

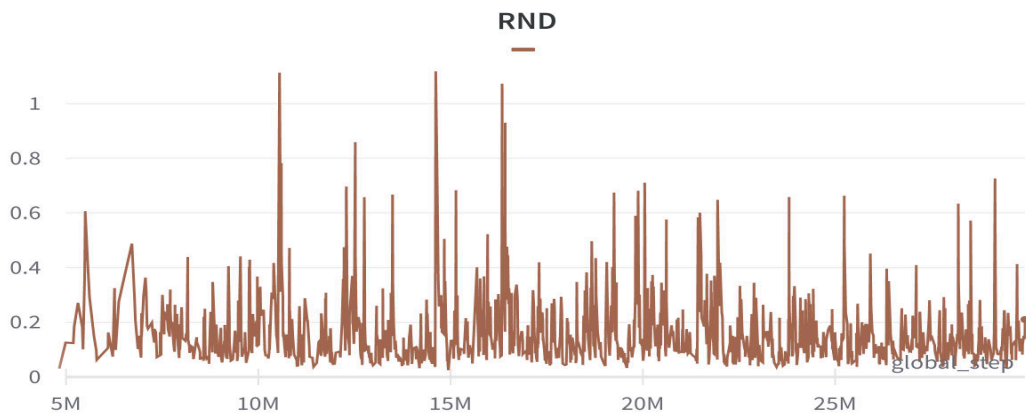


Figure 8.4: Intrinsic rewarding behavior of RND shows a pattern of low rewards for familiar states with occasional high peaks of reward for novel observations.

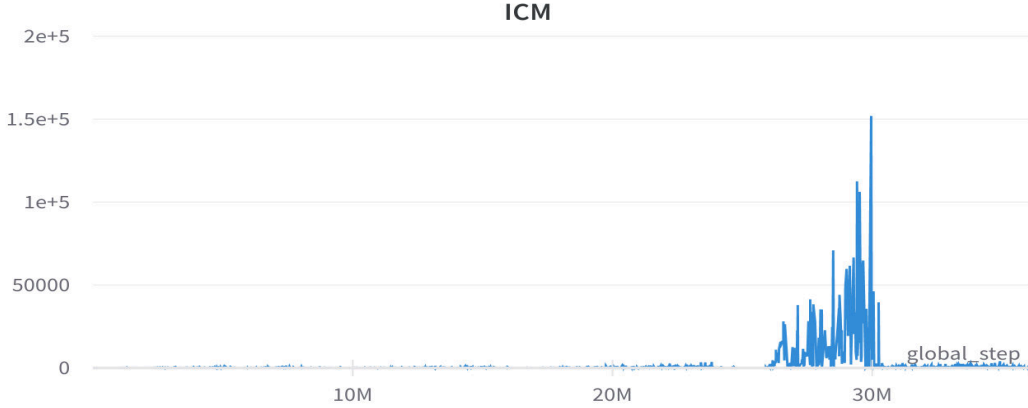


Figure 8.5: Intrinsic rewarding behavior of ICM shows very high intrinsic rewards the led to training collapse.

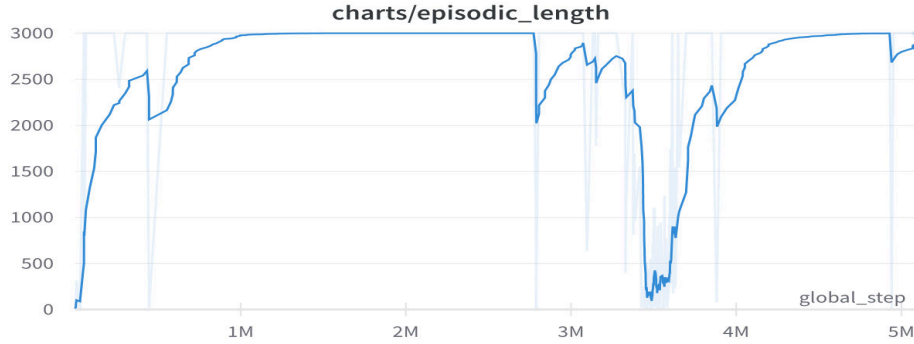


Figure 8.6: Rainbow converges quickly to a stalling behavior owing to the absence of instant rewards.

depicted in figure 8.5. These rewards tend to overshadow both extrinsic and primary rewards, thereby resulting in training instability and, in some cases, collapse. The process of regulating these rewards involves adjusting the scaling factor η , which is highly dependent on the environment and infeasible due to computational constraints.

Subsequently, we advanced our experiment by assessing Rainbow, a model that lacked both extrinsic and instantaneous feedback from the environment and intrinsic rewards from an external exploration method. As demonstrated in the episodic length chart 8.6, the Rainbow model displayed a static behavior by endeavoring to evade collisions with objects and walls. This conduct adversely affected the agent’s capacity to explore the environment effectively. The experiment highlighted the critical role of instantaneous feedback through exploration algorithms, which foster inquisitive behavior, assist the agent in escaping local optima and ultimately converge to the global optima and optimal behavior.

Subsequently, we employed a DreamerV3 agent within the experimental setup, which presented erratic conduct throughout numerous trials. This was attributed to the recurrent layer within the world model, which captures past information pertinent to reconstruct forthcoming observations. However, these historical data may not necessarily be relevant to the decision-making process of the actor, since the world model is trained independently from the actor-learner. Consequently, the actor is confronted with the issue of partial observability of the latent space generated by the world model. In addition to the recurrence challenge, assuming a world model that flawlessly replicates the environmental behavior and experiences no loss of information in its latent space, the meta learner would presumably perform similarly to an A2C agent owing to its unrestricted updates.

Ultimately, we conducted a comparative analysis of the aforementioned methods with respect to the performance of a human operator. The human operator exhibited a high success rate in identifying target objects in nearly all trials. This is primarily due to the absence of partial observability in humans and the ability of humans to obtain prior knowledge about the task and environment, unlike the RL agents. However, proficient RL agents such as PPO were able to surpass the human results in terms of velocity, navigation and control, which are crucial factors in real-world scenarios.

In the second experiment, following the comparison of various approaches, we examined the impact of the alterations imposed on all policies. We initially observed that employing ResNet for processing laser observations achieved a marginally faster convergence rate compared to CBAM. This is due to the added overhead introduced by the spatial and channel gates required to be learned with CBAM. However, in more intricate environments that entail a diverse range of objects and features, the use of spatial and channel gates can be advantageous.

Conversely, employing an MLP for processing the laser data resulted in a very slow and unstable convergence rate. This outcome can be attributed to the low number of laser points, which is computationally feasible for MLPs. However, for high-dimensional data, training MLPs would not be possible. In addition to computational efficiency, convolutional networks learn generic features that are relevant for a broad range of state distributions. On the other hand, MLPs may tend to overfit to the distribution of data on which they were trained.

Regarding recurrence, we conducted a comparison between utilizing a single LSTM layer and two stacked GRU layers against the default implementation, which includes separate GRU layers. In the default implementation, the first GRU layer processes the laser data, reducing its dimensionality to higher-level features. Subsequently, these features, along with other observation variables, are passed through the second GRU layer. Our adopted approach, which utilizes separate GRU layers, achieved superior results compared to the LSTM layer and two stacked GRU layers. Specifically, it was able to converge after 25 million steps for the former, while convergence was not achieved for the latter.

Reward: 6.047

env #11714_3

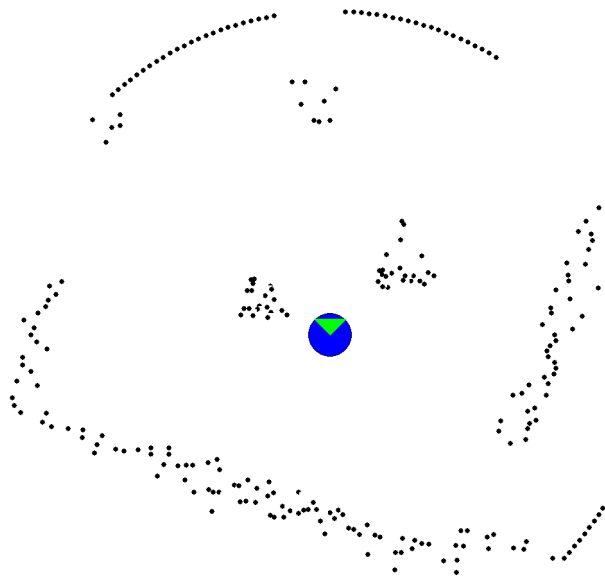


Figure 8.7: Illustration of noisy laser readings using Gaussian noise.

Next, we endeavored to simulate real-world scenarios by training the agent on noisy observations, as illustrated in figure 8.7. Impressively, the agent demonstrated its robustness by converging to optimal behavior in under 10 million steps. Nonetheless, this outcome was predictable since the noise introduced to the observation followed a Gaussian distribution. Eventually, the agent learned to disregard this noise and handle the mean of the Gaussian distribution, which is similar to conventional regression problems.

Finally, we conducted a set of experiments on hyperparameters to illustrate the sensitivity of RL methods to such parameters. Specifically, we altered the discount factor γ , while keeping all other parameters fixed and demonstrated how a minor adjustment in a single hyperparameter can result in drastically different outcomes. This sensitivity can be attributed to the high stochasticity of RL algorithms and the nature of the discount factor γ , which is contingent on both the environment and the algorithm used.

9 Conclusion and Outlook

In this thesis, we commenced by providing an introduction to RL, its components and the training paradigm of RL methods. Following that, we examined a broad range of RL methods, which differed from policy-based methods to value-based methods as well as model-based RL. We thoroughly explained all of these algorithms by presenting the motivation and theory behind each method, thereby making it easy for beginners to understand.

Next, we assessed multiple open-source RL libraries and selected CleanRL owing to its simplicity and straightforward implementation paradigm. This library was adapted to our task and further enhanced by adding several RL methods while maintaining the same simple implementation paradigm.

Moreover, we introduced the SAR environment and addressed several issues associated with its initial implementation and adapted it to behave like a standard RL simulation environment. We revamped the rewarding system by introducing a reward for entering the buildings, leading to more guided learning. In future works, this concept can be further extended further by introducing rewards for entering different rooms inside the building, thus allowing even more guidance due to reward rareness and incentivizing the agent to explore more rooms, which may eventually increase the probability of finding the target objects.

Furthermore, the issue of reward sparsity was further addressed by incorporating intrinsic reward algorithms into our agents, which ensured their motivation to explore uncharted territories and demonstrated how stalling behavior can result from the absence of both intrinsic and extrinsic rewards. In upcoming research, the exploration algorithms can be enhanced by incorporating recurrent layers, enabling even more robust intrinsic rewards with access to past states and preventing deception into believing that similar states are identical. Additionally, the feasibility of incorporating other exploration techniques, such as information gain or pseudo-counts, will be investigated, offering promising avenues for future research.

Following this, the RL policies were augmented with proficient attention-based processors, which facilitate enhanced identification of objects and a better understanding of the environment through laser sensors. This is achieved instead of the utilization of fully connected layers, which are unsuitable for processing raster data and entail significant computational expenses. Our work established the

superiority of CNNs over MLPs for the handling of raster data, owing to their ability to capture a broad range of generic features and prevent overfitting.

Afterwards, the partial observability of the environment was addressed by extending the networks of the policies with recurrent units. This enabled the agents to gain a more comprehensive understanding of their surroundings and the current state, utilizing both current observations and past states. As a result, their ability to navigate the environment and solve tasks was improved.

Afterwards, a evaluation of different approaches was conducted and it was observed that policy-based algorithms performed better than value-based methods, owing to their capability to maintain past information throughout the entire episode. In future work, there is an opportunity to investigate more advanced reinforcement learning algorithms that integrate multiple improvements from policy gradient algorithms, such as Muesli [60], as well as tree-search-based RL algorithms such as MuZero [61]. By incorporating these enhancements, it may be possible to further enhance the performance of RL algorithms and to solve increasingly complex problems.

Furthermore, our work emphasized the advantages of transfer learning in enhancing algorithmic performance, which was only applicable to policy-based methods. Therefore, it would be interesting to explore the potential benefits of equipping value-based algorithms with prior knowledge using offline RL methods, such as CQL, to improve their performance and enable more effective identification of target objects.

Lastly, an evaluation was conducted on DreamerV3 as an instance of model-based algorithms. Despite the underwhelming results, model-based methods remain a favorable option for tasks that require costly sampling, such as robotic scenarios or 3D simulations. Future work can improve DreamerV3’s performance by incorporating alternative architectures and recurrent layers in the world model, such as LSTM and equipping the actor-learner with recurrent nets, allowing it to retain relevant information for decision-making. Moreover, using more advanced policy-based algorithms like PPO or Muesli as the actor-learner instead of A2C could be considered in future research. Due to the model’s complexity, a rigorous hyperparameter tuning process may also be necessary.

In conclusion, building an end-to-end efficient and effective RL algorithm for exploring wide variety of environments and finding target objects is a challenging task, but one that can be tackled with the right approach. Breaking down the task into simpler sub-tasks like navigation and planning and using reinforcement learning in combination with other techniques such as general planning algorithms can help overcome some of the challenges faced in this task.

By breaking down the task into simpler sub-tasks, we can create a more manageable and densely rewarding environment for the navigation RL algorithm to be trained in. This can lead to better performance and more robust behavior in a wider range of environments. Additionally, incorporating

general planning algorithms into the system can help it to better adapt to new environments without the need for explicit training.

As we continue to develop and refine UAV systems for exploration and target detection, it is important to keep in mind the various challenges that must be overcome and to continue to innovate new approaches and techniques to address these challenges. By doing so, we can create more capable and effective UAV systems that can be used for a variety of applications, including search and rescue, disaster response and environmental monitoring.

Bibliography

- [1] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: <http://arxiv.org/abs/1712.01815>.
- [2] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (2016), pp. 484–489.
- [3] Christopher Berner et al. “Dota 2 with Large Scale Deep Reinforcement Learning”. In: *CoRR* abs/1912.06680 (2019). arXiv: 1912.06680. URL: <http://arxiv.org/abs/1912.06680>.
- [4] Yuxi Li. “Reinforcement Learning Applications”. In: *CoRR* abs/1908.06973 (2019). arXiv: 1908.06973. URL: <http://arxiv.org/abs/1908.06973>.
- [5] OpenAI et al. “Learning Dexterous In-Hand Manipulation”. In: *CoRR* abs/1808.00177 (2018). arXiv: 1808.00177. URL: <http://arxiv.org/abs/1808.00177>.
- [6] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: *CoRR* cs.AI/9605103 (1996). URL: <https://arxiv.org/abs/cs/9605103>.
- [7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.
- [8] Amir Masoud Sefidian. *Reinforcement Learning agent*. [Online; accessed March 13, 2023]. URL: <https://www.sefidian.com/2022/06/06/reinforcement-learning-q-learning-numerical-example/>.
- [9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.
- [10] H B Barlow. “Unsupervised learning: introduction”. In: *Unsupervised Learning: Foundations of Neural Computation*. Ed. by Geoffrey E. Hinton and Terrence Joseph Sejnowski. Bradford Company Scituate, MA, USA, 1999, pp. 1–17. URL: <https://doi.org/10.7551/mitpress/7011.001.0001>.
- [11] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. “Transfer Learning in Deep Reinforcement Learning: A Survey”. In: *CoRR* abs/2009.07888 (2020). arXiv: 2009.07888. URL: <https://arxiv.org/abs/2009.07888>.

- [12] Mathworks. *Reinforcement Learning*. [Online; accessed March 13, 2023]. URL: <https://www.mathworks.com/discovery/reinforcement-learning.html>.
- [13] Sergey Levine. *The anatomy of reinforcement learning*. [Online; accessed March 13, 2023]. URL: <https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-4.pdf>.
- [14] Vijay R. Konda and John N. Tsitsiklis. “Actor-Critic Algorithms”. In: 2000, pp. 1008–1014. URL: <http://papers.nips.cc/paper/1786-actor-critic-algorithms>.
- [15] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. “Model-based Reinforcement Learning: A Survey”. In: *CoRR* abs/2006.16712 (2020). arXiv: 2006.16712. URL: <https://arxiv.org/abs/2006.16712>.
- [16] Emanuel Todorov, Tom Erez, and Yuval Tassa. “MuJoCo: A physics engine for model-based control”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033.
- [17] Greg Brockman et al. “OpenAI Gym”. In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: <http://arxiv.org/abs/1606.01540>.
- [18] Andy L Jones. *megastep*. Version 0.1. July 7, 2020. URL: <https://andyljones.com/megastep>.
- [19] Eamonn Keogh and Abdullah Mueen. “Curse of Dimensionality”. In: *Encyclopedia of Machine Learning and Data Mining*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2017, pp. 314–315. URL: https://doi.org/10.1007/978-1-4899-7687-1_192.
- [20] Hongzi Mao et al. “Resource Management with Deep Reinforcement Learning”. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*. Ed. by Bryan Ford, Alex C. Snoeren, and Ellen W. Zegura. ACM, 2016, pp. 50–56. URL: <https://doi.org/10.1145/3005745.3005750>.
- [21] Richard S Sutton et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.
- [22] Hongzi Mao et al. “Variance Reduction for Reinforcement Learning in Input-Driven Environments”. In: *CoRR* abs/1807.02264 (2018). arXiv: 1807.02264. URL: <http://arxiv.org/abs/1807.02264>.
- [23] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: <http://arxiv.org/abs/1602.01783>.
- [24] John Schulman et al. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: <http://arxiv.org/abs/1506.02438>.

- [25] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: <http://arxiv.org/abs/1707.06347>.
- [26] John Schulman et al. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: <http://arxiv.org/abs/1502.05477>.
- [27] Surya Tokdar and Robert Kass. “Importance sampling: A review”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2 (Jan. 2010), pp. 54–60.
- [28] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. URL: <https://arxiv.org/abs/1312.6114>.
- [29] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: <http://arxiv.org/abs/1312.5602>.
- [30] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *CoRR* abs/1710.02298 (2017). arXiv: 1710.02298. URL: <http://arxiv.org/abs/1710.02298>.
- [31] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: <http://arxiv.org/abs/1509.06461>.
- [32] Tom Schaul et al. *Prioritized Experience Replay*. cite arxiv:1511.05952Comment: Published at ICLR 2016. 2015. URL: <http://arxiv.org/abs/1511.05952>.
- [33] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. cite arxiv:1511.06581Comment: 15 pages, 5 figures, and 5 tables. 2015. URL: <http://arxiv.org/abs/1511.06581>.
- [34] Meire Fortunato et al. “Noisy Networks for Exploration”. In: *CoRR* abs/1706.10295 (2017). arXiv: 1706.10295. URL: <http://arxiv.org/abs/1706.10295>.
- [35] Tuomas Haarnoja et al. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: <http://arxiv.org/abs/1801.01290>.
- [36] Richard S. Sutton. “Dyna, an integrated architecture for learning, planning, and reacting”. In: *SIGART Bull.* 2 (1990), pp. 160–163.
- [37] Jong Woo Kim et al. “Model-based reinforcement learning and predictive control for two-stage optimal control of fed-batch bioreactor”. In: *Computers And Chemical Engineering* 154 (2021), p. 107465. URL: <https://www.sciencedirect.com/science/article/pii/S009813542100243X>.
- [38] Danijar Hafner et al. “Mastering Atari with Discrete World Models”. In: *CoRR* abs/2010.02193 (2020). arXiv: 2010.02193. URL: <https://arxiv.org/abs/2010.02193>.

- [39] Danijar Hafner et al. “Dream to Control: Learning Behaviors by Latent Imagination”. In: *CoRR* abs/1912.01603 (2019). arXiv: 1912.01603. URL: <http://arxiv.org/abs/1912.01603>.
- [40] Danijar Hafner et al. “Mastering Diverse Domains through World Models”. In: *arXiv preprint arXiv:2301.04104* (2023).
- [41] Deepak Pathak et al. “Curiosity-driven Exploration by Self-supervised Prediction”. In: *CoRR* abs/1705.05363 (2017). arXiv: 1705.05363. URL: <http://arxiv.org/abs/1705.05363>.
- [42] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *Journal of Machine Learning Research* 11.110 (2010), pp. 3371–3408. URL: <http://jmlr.org/papers/v11/vincent10a.html>.
- [43] Thomas Simonini. *Curiosity-Driven learning through next state prediction*. [Online; accessed March 13, 2023]. URL: <https://medium.com/data-from-the-trenches/curiosity-driven-learning-through-next-state-prediction-f7f4e2f592fa>.
- [44] Yuri Burda et al. “Exploration by Random Network Distillation”. In: *CoRR* abs/1810.12894 (2018). arXiv: 1810.12894. URL: <http://arxiv.org/abs/1810.12894>.
- [45] Thomas Simonini. *Random Network Distillation: a new take on Curiosity-Driven Learning*. [Online; accessed March 13, 2023]. URL: <https://medium.com/data-from-the-trenches/curiosity-driven-learning-through-random-network-distillation-488ffd8e5938>.
- [46] Antonin Raffin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: <http://jmlr.org/papers/v22/20-1364.html>.
- [47] Shengyi Huang et al. “CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms”. In: *Journal of Machine Learning Research* 23.274 (2022), pp. 1–18. URL: <http://jmlr.org/papers/v23/21-1342.html>.
- [48] Marc G. Bellemare, Will Dabney, and Rémi Munos. “A Distributional Perspective on Reinforcement Learning”. In: *CoRR* abs/1707.06887 (2017). arXiv: 1707.06887. URL: <http://arxiv.org/abs/1707.06887>.
- [49] Will Dabney et al. “Implicit Quantile Networks for Distributional Reinforcement Learning”. In: *CoRR* abs/1806.06923 (2018). arXiv: 1806.06923. URL: <http://arxiv.org/abs/1806.06923>.
- [50] Alexandre Piché et al. “Beyond Target Networks: Improving Deep Q -learning with Functional Regularization”. In: *arXiv preprint arXiv:2106.02613* (2021).
- [51] Burak Kaleci, Kaya Turgut, and Helin Dutagaci. “2DLaserNet: A deep learning architecture on 2D laser scans for semantic classification of mobile robot locations”. In: *Engineering Science and Technology, an International Journal* 28 (2022), p. 101027. URL: <https://www.sciencedirect.com/science/article/pii/S2215098621001397>.

- [52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.
- [53] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: <http://arxiv.org/abs/1512.03385>.
- [54] Sanghyun Woo et al. “CBAM: Convolutional Block Attention Module”. In: *CoRR* abs/1807.06521 (2018). arXiv: 1807.06521. URL: <http://arxiv.org/abs/1807.06521>.
- [55] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. cite arxiv:1412.3555Comment: Presented in NIPS 2014 Deep Learning and Representation Learning Workshop. 2014. URL: <http://arxiv.org/abs/1412.3555>.
- [56] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (1997), 1735–1780. URL: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [57] Tim Salimans and Richard Chen. “Learning Montezuma’s Revenge from a Single Demonstration”. In: *CoRR* abs/1812.03381 (2018). arXiv: 1812.03381. URL: <http://arxiv.org/abs/1812.03381>.
- [58] Olivia Wiles et al. “A Fine-Grained Analysis on Distribution Shift”. In: *CoRR* abs/2110.11328 (2021). arXiv: 2110.11328. URL: <https://arxiv.org/abs/2110.11328>.
- [59] Aviral Kumar et al. “Conservative Q-Learning for Offline Reinforcement Learning”. In: *CoRR* abs/2006.04779 (2020). arXiv: 2006.04779. URL: <https://arxiv.org/abs/2006.04779>.
- [60] Matteo Hessel et al. “Muesli: Combining Improvements in Policy Optimization”. In: *CoRR* abs/2104.06159 (2021). arXiv: 2104.06159. URL: <https://arxiv.org/abs/2104.06159>.
- [61] Julian Schrittwieser et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *CoRR* abs/1911.08265 (2019). arXiv: 1911.08265. URL: <http://arxiv.org/abs/1911.08265>.