A Study of the Efficacy of Generative Flow Networks for Robotics and Machine Fault-Adaptation

by

Zahin Sufiyan

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

In 2005, Opportunity, one of NASA's renowned Mars rovers, faced a dire situation. It was a moment that could end a mission that had already far outlasted its expected lifespan. After clambering out of the Victoria crater, the rover started to experience an abrupt current spike in its right front wheel. As a consequence, the wheel motor started to malfunction, causing the wheel to stop turning. NASA anticipated that the $400 million dollars of investment and most importantly invaluable scientific data regarding the Martian terrain was on the brink of remaining unexplored. With the immensity of space between Mars and Earth (140 million miles, to be specific), the engineers at NASA could only detect and diagnose the malfunction; however, human intervention in the maintenance of Opportunity was an impossibility. Nevertheless, human ingenuity again succeeded when the engineers at NASA's Jet Propulsion Laboratory came up with an unconventional workaround. They started to drive the rover backward and thus by doing so, they were able to redistribute the mechanical load and reduce the strain on the malfunctioning wheel. The impaired wheel now functioned as a rear wheel, allowing the fully functional wheels to lead and navigate the harsh and challenging surface of Mars. Due to this innovative approach, Opportunity continued to explore Mars and gathered some of the most invaluable data about the red planet for 15 Earth years instead of its initially predicted 90-day lifespan. This was a testament to human ingenuity, but also a stark reminder of the necessity for built-in machine fault adaptability in robotic systems.

Our research is a step towards adding hardware fault tolerance and fault adaptability to machines. In this research, our primary focus is to investigate the efficacy of generative flow networks (GFlowNets/CFlowNets) in robotic environments, particularly in the domain of machine fault adaptation. Generative Flow Networks is an emerging algorithm with the potential to be considered as a substitute approach to the prevalent reinforcement learning methods in continuous exploratory tasks. In our work, the experimentations were done in a simulated robotic environment (Reacher-v2). This environment was manipulated and modified to introduce four distinct fault environments which are reduced range of motion, increased damping, actuator damage, and structural damage. Each fault replicates actual malfunctions that are generally witnessed in real-world machines/robots that render them inoperative. The empirical evaluation of this research indicates that continuous generative flow networks indeed have the capability to add adaptive behaviors in machines under adversarial conditions in the environment. Furthermore, the comparative analysis of CFlowNets with state-of-the-art RL algorithms also provides some key insights into the performance in terms of adaptation speed and sample efficiency. Despite a few algorithmic shortcomings, our experiments confirm that CFlowNets has the potential to be deployed in a real-world machine and it can demonstrate adaptability in case of malfunctions to maintain functionality. The thesis is motivated by the idea of transforming robots into more than just mere tools, making them capable entities which are capable of autonomously overcoming certain faults and failures, thus sustaining their operation while delaying the need for maintenance. Through experimentation in simulated robotic environments, the comparative study aims to contribute to the ongoing discourse on enhancing the adaptive capacities of automated systems and machines.

Preface

A major part of this thesis will be submitted as an extended paper to a journal.

"It is not the strongest of the species that survives, nor the most intelligent, but the one most ***adaptable*** *to change."*

-Charles Darwin

Acknowledgements

I would like to express my deepest gratitude and appreciation to my supervisor Dr. Osmar Zaiane, for his unwavering support, guidance, and patience throughout this journey. His encouragement and support always created a nurturing environment during challenging times, helping me to transform every obstacle into opportunities for growth. For me, he is not only my mentor but also a source of great inspiration which helped to push myself beyond my own limits to coordinate my project, especially in writing this thesis dissertation. I am honored to have embarked on this scholarly journey under the supervision of Dr. Zaiane.

Furthermore, I would also like to acknowledge, with heartfelt gratitude, the important role of Sheila Ann Schoepp, who has been like a guiding figure and another mentor to me during my research. Beyond her technical expertise, her moral support and friendship have been instrumental at every step of this journey.

I would like to extend my gratitude to Yoshihiro Mitsuka and Shotaro Miwa from Mitsubishi Electric Co. Their invaluable insights and profound suggestions were crucial in laying the foundation of this project. I am equally grateful to Mitsubishi Electric Co. for funding and providing me with the opportunity that empowered me to realize this research.

To my parents, Noman Sufiyan and Hasna Hena, and in loving memory of my late brother Tonmoy Sufiyan, I am here today because of your boundless faith in me and your immeasurable sacrifices.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI Artificial Intelligence.

CFlowNets Continuous Flow Networks.

DDPG Deep Deterministic Policy Gradient.

DRL Deep Reinforcement Learning.

GAE Generalized Advantage Estimator.

GB Gigabyte.

GFlowNets Generative Flow Networks.

MARS Markov Molecular Sampling.

MB Megabyte.

MCMC Markov Chain Monte Carlo.

MDP Markov Decision Process.

MSE Mean Squared Error.

NN Neural Network.

PPO Proximal Policy Optimization.

ReLu Rectified Linear Unit.

RL Reinforcement Learning.

ROM Range of motion.

SAC Soft Actor-Critic.

TD3 Twin Delayed Deep Deterministic Policy Gradient.

XML Extensible Markup Language.

Chapter 1

Introduction

Robotic technology has brought about substantial economic changes in several industries, particularly in manufacturing, and has immense potential to offer numerous benefits to society. A wide range of applications has been implemented using it. These include search and rescue operations, manufacturing, disaster response, healthcare, and transportation. Furthermore, they are indispensable tools for scientific research, enabling the exploration of remote areas such as planets and oceans. However, a significant hurdle to their widespread implementation in more diverse environments is their vulnerability, particularly for complex machines.

Robots, like any other machine, are susceptible to damage and failure, particularly when operating in a real-world environment. In outdoor or unstructured environments, these machines can be exposed to various sources of damage, including rough terrain, collisions with obstacles, and exposure to harsh weather conditions. Furthermore, machines can suffer from component wear and tear, battery depletion, software bugs, and other faults that reduce their performance and reliability or even cause them to fail altogether. It is also possible that damage to a robot or machine may pose a safety risk to humans in the area. Additionally, as a result of these failures, quick maintenance is essential to ensure continued operations. However, maintenance is not always readily available like in remote areas. To minimize the risk of damage, downtime and ensure the longevity of robots, it is essential to develop strategies for fault diagnosis, fault detection, identification, and adaptation.

Imagine a warehouse where an advanced fleet of robots and manufacturing machines all work together to assemble products from one assembly line to another. The robots are equipped with an extensive network of sensors and devices that enable them to detect and diagnose faults in real-time. If a robot experiences a malfunction, such as a broken arm or reduced range of motion, a supervisory artificial intelligence (AI) algorithm is able to detect and respond to the detected malfunction. The AI algorithm temporarily adjusts the movement of the affected robot and alerts other robots in the network about the fault that has occurred. At the same time, the algorithm processes the fault data gathered from the malfunctioning robot and adapts its task performance, such as using alternative routes or modifying its joint and limb movements to work around the fault. Once the robot has successfully adapted, the supervisory AI control system is notified, and the robot returns to its normal duties. A fault adaptation system such as the above example enables the fleet of robots to continue working efficiently and effectively, even when one of them experiences a fault or failure through the process named fault adaptation. With the ability to detect and recover from faults autonomously, robots can continue to operate effectively in challenging environments and achieve their intended tasks with minimal disruption. Consequently, developing robust fault adaptation techniques is a critical area of research for the continued advancement and deployment of robotics in the real world. In summary, the benefits of machines and robots that can adapt when a fault occurs are numerous and far-reaching. From increased productivity and reduced maintenance costs to improved safety and product quality, fault adaptation technology has the potential to revolutionize many industries. Furthermore, continued innovation in this field can lead to even more advancements and opportunities in the future.

1.1 Motivation

Across a wide range of industries, from manufacturing to transportation to energy, there is a growing trend toward automation and interconnectivity. These systems rely on a variety of technologies, including artificial intelligence, machine learning, reinforcement learning, and the Internet of Things (IoT), to create a seamless and highly efficient network of machines and devices.

As these machines and robots become more autonomous and move out of the controlled settings of factories and into the more dynamic and unpredictable environments of the natural world, it is inevitable that they will encounter situations that result in damage or other faults. Even inside controlled environments such as manufacturing industries, factory machines/robots may face adversarial conditions that can hinder their operations. These types of disruptions and machine faults may hamper these industries' manufacturing processes by causing downtime. In modern digital manufacturing, mechanical failures cause 79.6% of machine tool downtime. [1].

It might also be difficult for a maintenance team to repair a machine when a fault occurs in remote locations for research and exploration purposes. An example would be deep-sea exploration using autonomous underwater vehicles (AUVs) to map the seafloor, conduct surveys, and collect data on marine life. These vehicles can withstand the extreme pressures and temperatures of the deep oceans and can operate for weeks or even months at a time without human intervention. Such situations call for a highly automated, interconnected system that is capable of adapting to changing conditions whenever a fault occurs without the need for any manual intervention. By detecting and adapting to changing conditions, including machine faults, these systems can help prevent downtime, reduce waste, and improve overall performance.

The growing trend of fully automated and interconnective systems enables the utilization of a wide range of sensors and monitoring systems that collect data on the performance of the machine, which in turn helps with the identification of deviation from normal operating conditions or the detection of anomalies such as hardware failure. These techniques are commonly referred to as fault detection and fault diagnosis [2]. Detection and diagnosis mechanisms of this type are only concerned with detecting faults as soon as they arise but do not address the issue of post-fault detection.

As machines become increasingly complex and autonomous, it is becoming more important than ever for them to have the capability of some sort of tolerance and adaptability of their behavior in response to faults and errors. The first step towards such an approach to adding fault tolerance in machines is through hardware redundancy. Hardware redundancy refers to the duplication of essential machine components in order to replace similar broken components whenever a fault occurs [3]. However, even though the process of hardware redundancy has been quite an established method of fault tolerance, it does come with its own disadvantages. For instance, the duplication of components increases the size, weight, power consumption, and financial cost of the machine. Additionally, retrofitting redundant components into existing machines can also be challenging since the original blueprint of the machine may not allow such modification. Using more materials to build machines with redundant components also has a negative impact on the environment. As a result, researchers are exploring new approaches to fault tolerance that can address these issues and provide more efficient and sustainable solutions. Some of these approaches include software redundancy, self-healing systems, fault prediction and adaptation. These methods can be less costly and easier to implement than hardware redundancy while still providing the necessary level of fault tolerance to maintain machine safety and reliability.

An approach to making machines fault-tolerant that has received considerable attention in recent times is to add the capability of adaptability. One of the best sources of inspiration for creating adaptable machines is nature itself. Compared to natural animals, robots still lack the ability to create compensatory behaviors following an injury, limiting their capabilities. Like living organisms, machines require the ability to adapt to unexpected changes in their environment, whether that means adjusting to damage or finding a new way to complete a task. For instance, it is common for birds to change their normal behaviour in response to injuries [4]. If a bird has an injury to its wing, it may compensate by changing the way it flaps its wings or by altering the angles of its wings during flight. This enables the bird to maintain its balance and continue to fly. Despite its injuries, as a result of this adaptive behavior, the bird has modified its pattern to be able to function. This sort of adaptability may enable the bird to learn new flight patterns that are more efficient and effective for its altered physical condition. Machines can also take inspiration from nature by adopting a fault-tolerant approach, similar to how some animals and plants are able to survive and thrive in hostile environments. In fact, many researchers and engineers are now turning to biomimicry, the process of imitating nature's designs and processes, to create machines that are better able to handle unexpected challenges [5]. The approach entails building machines that are capable of recognizing and adjusting to variations in the environment, including the presence of faults, to ensure continued operation.

During the past few years, deep learning has gone from being a means of predicting machine faults to being a primary focus of the field. Rather than just predicting or diagnosing faults, the industry is now focusing more on post-fault phases. Recently, a number of research projects have already been conducted in the context of fault tolerance/fault adaptation using reinforcement learning. However, since fault adaptation tasks require diverse exploration and reinforcement learning has the issue of balancing exploration and exploitation for large-scale problems, new frontiers of research have been explored that can become an alternative method for standard reinforcement learning algorithms. This is where the emerging technology Generative Flow Networks (GFlowNets) provide a solution to this problem by compensating for the shortcomings of reinforcement learning in exploratory tasks. Through the generation of a distribution proportional to rewards over terminating states, generative flow networks enhance exploration capabilities [6]. One of the key differences between GFlowNets and reinforcement learning is that GFlowNets generate a distribution over all possible paths and sample from the most rewarding paths with a higher probability which makes them more sample-efficient than standard reinforcement learning algorithms because they tend to stick to the most rewarding path which can be a local optimum. However, it is worth mentioning that generative flow networks and RL have different objective functions and they solve different problems. While RL seeks to maximize the reward, GFlowNets emphasizes on generating outputs in proportion to the reward to produce a diverse set of high-reward solutions. A variant of GFlowNets that is primarily being investigated in this research is tokenized as CFlowNets, or Continuous Flow Networks. CFlowNets can be considered as an updated extension of the theoretical formulations of GFlowNets which is able to adapt to continuous control task that has a continuous state-action space [7]. In this study, GFlowNets and CFlowNets are mainly investigated for their possible implementation in robotics as well as to determine whether the generative flow networks demonstrate adaptive behavior in exploratory tasks. To the best of our knowledge, no research has yet implemented and evaluated the efficacy of generative flow networks in machine fault adaptation.

1.2 Thesis Objectives

The primary objective of this thesis is to test a hypothesis that addresses the question of whether GFlowNets/CFlowNets can be considered as a potential algorithm to generate candidate high-return solutions in the context of continuous exploratory tasks primarily concerned with exploration. Additionally, the thesis seeks to assess the performance of generative flow networks in comparison to RL. This will be achieved by conducting a comparative study of reinforcement learning algorithms and generative flow networks. Additionally, the implementation of generative flow networks provides a new perspective toward autonomous systems capable of self-adaptation without the need for human intervention. The ability of machines and robots to adapt to faults can be very beneficial to a variety of industries. By detecting and diagnosing faults in real-time, these machines can adapt to unexpected failures and malfunction which will result in increased productivity, reduced downtime, and ultimately, increased profits. Furthermore, machines can extend their lifespan and reduce maintenance costs by adapting to faults. It also reduces the need for expensive repairs, delays the need for repair/maintenance and maximizes resource efficiency.

GFlowNets/CFlowNets framework offers a potential solution for developing machines and robots that can adapt to hardware failures autonomously. This thesis involves a thorough evaluation of Generative Flow Networks (GFlowNets) and Generative Continuous Flow Networks (CFlowNets), in terms of their ability to enable adaptation to failures. Unlike previous research that focused on detecting malfunction [2], [8] and diagnosing malfunctions [9], [10], [11] in machines, this thesis assumes that the system already has fault detection and diagnosis capabilities where the system knows about the fault along with its specific limitations. This means that the focus is on how the system can adapt to the fault, instead of detecting it.

The primary objective of this thesis is to investigate the possibility and feasibility of using GFlowNets/CFlowNets in machine fault adaptation. The purpose of this study is to seek answers to the following questions:

- Can Generative Flow Networks be utilized in robotic applications?
- Is it possible to develop a Continuous GFlowNets Learning approach for machines well suited for continual interaction with environments? In other words, Can GFlowNets or CFlowNets theoretical formulations work with continuous action and state space?
- Can Generative Flow Networks adapt when a fault is introduced in the robotic

environment? In other words, can it be utilized for fault adaptation tasks?

  - If yes, how does it compare to reinforcement learning methods (PPO, SAC, TD3, and DDPG) in terms of adaptation speed and sample efficiency?

- How does the computational resource consumption of flow networks compare to standard RL algorithms?

- Does incorporating the transfer of task knowledge and storage content enhance performance efficiency for Generative Flow Networks?

The thesis will begin with a comprehensive review of relevant literature on fault adaptation in machines and the recent application of reinforcement learning algorithms for fault tolerance and adaptation. This will provide the tools to identify gaps and opportunities for further research, laying a theoretical foundation for the study. Using data collected from fault detection and diagnosis, flow network models will be developed and trained, and their performance will be tested as they adapt to hardware failures.

In order to assess the performance of flow networks, we perform experiments on a continuous control problem. We run our experiments in a simulated robotic environment Reacher-v2 in MuJoCo [12] OpenAI Gym [13]. For our empirical comparative analysis, we examine four reinforcement learning algorithms which are DDPG (Deep Deterministic Policy Gradient) [14], TD3 (Twin-delayed DDPG) [15], Proximal Policy Optimization (PPO) [16] and Soft-Actor-Critic (SAC) [17] and evaluate their ability to add fault tolerance compared to the flow networks. As part of the ongoing research on developing more efficient and reliable industrial systems, the study provides insight into the potential of the generative flow networks as novel frameworks for machine fault adaptation.

1.3 Manuscript Organization

The introduction chapter has explained fault tolerance, adaptation, and the motivation and objective for this research study. The remaining chapters are organized as follows: The second chapter focuses exclusively on providing the background knowledge on reinforcement learning. Chapter 3 delves into foundational information about GFlowNets and its variant, CFlowNets. Chapter 4 discusses related work, including prior research outcomes related to the field of machine fault adaptations. Chapter 5 extensively describes the design of the experimental setup and the fault environment details and also elaborates on the robot simulator that has been used to conduct our experiments. Chapter 6 reports our findings based on our multiple segments of experiments and discusses the interpretations of the results. Chapter 7 concludes with a summary of our thesis research, including strengths and limitations, as well as future research directions.

Chapter 2

Background on Reinforcement Learning

In this chapter, we provide background information on the contributions of this thesis. Since this study works as a comparative analysis of Reinforcement learning and Generative Flow Networks, we begin by discussing the fundamental concepts of reinforcement learning in this chapter. As a follow-up to discussing reinforcement learning's conceptual foundations, we review different types of reinforcement learning algorithms. Afterward, we describe four state-of-the-art reinforcement learning algorithms that are used in our comparative analysis.

2.1 Reinforcement Learning

In reinforcement learning [18], agents are trained to make sequential decisions in a given environment. The process can be compared to the process of training a dog to do something specific, like fetch a ball, for example. Reinforcement learning can be viewed as an interaction between the dog which can be considered an agent and its environment, which is the area where the dog can move around and interact with real-world objects. The reward signal in this example could be seen as the dog receiving a treat for fetching the ball successfully. Initially, the dog is not able to understand the right set of actions it has to do to perform the task. However, after a series of trials and errors, for instance, chasing the ball or barking at it, when it finally successfully retrieves the ball, it receives a treat. This treat is a reward the dog receives from the environment which helps reinforce the behavior and encourages similar efforts in the future. In this example, positive reinforcement which came in the form of a treat and the method of trial and error is used to teach the agent (dog) how to select the correct course of action to perform a desired task. In a similar manner, reinforcement learning involves an agent being given a set of allowed actions that they use in order to interact with the world and get positive or negative feedback depending on the selected action. In order to maximize rewards, agents learn to map aspects of the world to particular actions through a process of trial and error.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework that is generally used to describe the settings and environment of reinforcement learning (RL). It is used to model sequential decision-making problems in dynamic environments. The MDP framework consists of two entities which are the agent and the environment. The agent in this scenario works both as a learner and a decision-maker while the environment consists of everything that the agent interacts with. The environment is completely external to the agent. In a series of discrete time steps, $t \geq 0$, the interaction between the agent and the environment occurs. At every discrete set of time steps, the agent receives information about its current state provided by the environment. This current state is denoted by $S_t \in \mathcal{S}$, where $\mathcal{S}$ is a finite set of all possible states. The agent has the option of choosing an action $A_t \in \mathcal{A}(s)$ from the current state, where $\mathcal{A}(s)$ is the finite set of all possible actions that can be taken from that state. After the agent selects and executes the selected action, the environment updates its state $S_{t+1} \in \mathcal{S}$, and the agent gets a computed discrete numerical reward $R_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is a subset of real-valued numbers which is based on the agent's chosen action and current state.

The state and reward at the next time step (S_{t+1}, R_{t+1}) after the execution of an action are determined by the MDP dynamics probability function $\mathcal{P}$, where $\mathcal{P}$ is the probability operator:

$$\mathcal{P}(s', r|s, a) = Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{2.1}$$

The Markov property of the MDP framework is considered one of its most important features. The Markov property states that whenever an agent takes an action given a state, the probability of the resultant reward and state depends only on the previous state and action. In other words, it does not depend on the entire history of all the states and actions that preceded it. At a given time-step, as the future state is determined solely based on the present state and action the Markov property is sometimes also referred to as "Memoryless property".

An agent's interaction with the environment leads to a series of states, actions, and rewards at each time step. This is known as a trajectory:

$$S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{2.2}$$

Every trajectory begins with an initial state which is problem-specific or randomly selected. At the end of the trajectory, the final state transitions to zero rewards with a probability of one. This final state is also known as the terminal state. An episodic task is one that triggers a zero probability when it reaches a terminal state or a fixed number of timesteps or episodes. However, sometimes the trajectory continues indefinitely and the agent interacts with the environment without a fixed endpoint. Tasks of this nature are referred to as continuing tasks.

2.1.2 Returns, Policies, and Value Functions

The primary objective of any reinforcement learning method is to maximize the expected sum of rewards across a trajectory. The goal of an agent is to execute those set of actions for which the expected sum rewards are maximized which is referred to as expected returns. Return is denoted as G_t which is all the rewards combined in a trajectory:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T = \sum_{k=0}^{\infty} R_{t+1+k} \tag{2.3}$$

Since there are two types of tasks related to reinforcement learning trajectory: episodic tasks and continuing tasks, the returns corresponding to these two types of tasks are also different. In episodic tasks, where the agent interacts with the environment for a fixed number of timesteps and it contains a terminating state T, the return is defined as shown in Equation 2.3. On the other hand, for continuing tasks, the agent interacts with the environment indefinitely and as a result, the sum of rewards can be unbounded because there are no terminal states. In this scenario, the conventional return would not work because of the absence of terminal states. Rather we need to consider discounted return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

This type of discounted return takes into account the time value of the rewards. The rewards that are received by the agent during a trajectory at each time step are multiplied by γ which is referred to as the discount factor. Discount factors are usually raised to the power of the time step and are in the interval [0, 1] which specifies the amount of weight given to rewards in the distant future relative to those in the immediate future. If the discount factor γ is set to zero the agent does not consider the future rewards and puts more weight on immediate rewards which means the agent's decision-making is entirely dependent on immediate feedback. Conversely, $\gamma = 1$ signifies that the agent values the future reward as much as the immediate rewards which makes the agent's decision-making inclined towards maximizing the total sum of rewards over time.

One of the two critical concepts of reinforcement learning is policy and value function. In essence, the policy is a mapping of states to actions. Given a state, the policy determines the action to take by the agent. There are two types of policies in reinforcement learning. A deterministic policy maps each state to a particular action.

In other words, whenever an agent transits into a particular state, the same action is executed according to a deterministic policy. For instance, if a machine encounters a certain type of specific malfunction, such as facing an obstacle, it may have a deterministic policy to turn away given that particular state. The deterministic policy is represented as:

$$\pi(s) = a \tag{2.5}$$

In contrast, a stochastic policy is a mapping from states to a probability distribution over all allowed actions from that state. The stochastic policy is represented as:

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{2.6}$$

Value functions are a tool to assess the goodness of a state. It is a function that measures how good it is for an agent to be present in a state after following a policy π. By following a policy π, we define the state-value function that results from the policy as follows:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\Big] \tag{2.7}$$

Similarly, we can define the action-value function which is the value function with respect to state-action pairs as:

$$Q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{2.8}$$

2.1.3 Policy Gradient Methods

Policy gradient methods are a set of reinforcement learning methods that are different from the standard conceptual reinforcement learning methods. Whereas in the usual case, reinforcement learning methods model a policy by measuring the goodness of a current state, in other words by using value function, policy gradient methods learn a policy directly without utilizing value function as an intermediate step. Policy gradient methods are specifically advantageous for continuous action space. In this method, the policy is directly optimized, so an action-value function is not necessary.

The objective of policy gradient methods is to maximize the expected sum of rewards by optimizing a parametrized policy. The parametrized function:

$$\pi_\theta(a|s, \theta) = Pr(A_t = a|S_t = s, \theta) \tag{2.9}$$

One way to maximize the expected reward is by using gradient ascent. Gradient ascent generally updates the weights/policy parameters θ in the direction that maximizes the expected return which in turn increases high reward action probability and minimizes the action probabilities that result in low rewards.

There are several well-known policy gradient methods used in reinforcement learning, including REINFORCE, Trust Region Policy Optimization (TRPO), and Actor-Critic. However, the Actor-Critic method is a bit different because it combines elements of both policy-based and value-based methods. As policy gradient methods have evolved, various improvements have been made. For example, the Actor-Critic architecture has introduced a critic to reduce gradient estimate variances. By incorporating a value-based method (critic) along with a policy-based method (actor), learning stability and efficiency have been enhanced.

2.1.4 Exploration vs. Exploitation

As mentioned in the background section, the primary goal of a reinforcement learning algorithm is to maximize the expected sum of rewards across a trajectory over a period of timesteps. To achieve a higher expected return, it is necessary for an RL agent to learn to execute a series of actions that yield higher rewards. However, this task may prove challenging, as the agent has to strike a balance between exploration and exploitation.

The term exploitation refers to the RL agent's tendency to exploit the current knowledge by greedily choosing the greedy policy to get a high reward. It means exploiting the current information that the agent received from the environment and selecting actions that are already expected to produce the highest rewards based on prior knowledge. For instance, if you enjoy the movie genre sci-fi which never fails to entertain you, and then you decide to watch a movie which is only of the genre sci-fi then you are taking advantage of your prior movie experiences by exploiting your current knowledge. If an agent only exploits the environment, it may always get stuck in a suboptimal policy while better trajectories with higher rewards may remain hidden during the learning phase.

On the other hand, exploration refers to the scenario where the agent tries to improve its knowledge by taking random actions that are not guaranteed to yield high rewards. In other words, during exploration, the agent executes actions that are not yet well understood in the hopes of gaining new information about the environment and potentially discovering new action paths that may even produce trajectories with greater rewards. If you decide to watch a rom-com movie instead of always watching sci-fi, you might like it even more. That is why exploration is important because it may lead to better policy. However, since the agent is venturing toward uncharted territories during exploration, it may yield much lower rewards initially, but in the long run, it may generate more optimal policies.

2.1.5 On/Off-Policy Algorithms

Reinforcement learning methods can generally be categorized into two types: On-policy and Off-policy. The key difference between on-policy and off-policy lies in how these methods use data to update the policy. On-policy algorithms refer to the set of RL algorithms that update their policy by using the data that is collected under the current policy. In other words, this algorithm tries to improve and evaluate the same policy that is being used for the action selection procedure which means the behavior policy is the same as the target policy. Behavior policy means the policy that the agent uses to select an action given a current state, and the target policy is the policy that the agent is trying to learn to approximate and improve. Some commonly used on-policy algorithms are Proximal Policy Optimization (PPO), Sarsa, Advantage

Actor-critic (A2C), Vanilla Policy Gradient (VPG), etc.

On the other hand, off-policy algorithms make an update to their policy using data generated under different policies. A behavioral policy is used in the off-policy method to explore the environment and collect samples, which generates the agent behavior, and a second policy is learned, called the target policy. Q-learning is an off-policy method most commonly used. Some other examples of off-policy RL algorithms include Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic Policy Gradient (TD3), Soft Actor-critic (SAC), etc.

2.1.6 Proximal Policy Optimization (PPO)

Proximal Policy Optimation (PPO) [16] is one of the state-of-the-art reinforcement learning algorithms to date. It was first introduced in 2017 in an effort to train deep reinforcement learning agents with improved stability by avoiding too-large policy updates. Since then, it has been widely utilized because of its stability and faster convergence compared to other policy gradient algorithms in large-scale DRL tasks. PPO is an on-policy algorithm where an update to the policy is made using data collected by the current policy.

One of the key drawbacks of traditional policy gradient algorithms is that during their large policy updates the policy parameters are used to deviate too much from previous values. The policy update is regulated by the learning rate and whenever the learning rate is too small, it results in slow convergence, while if the learning rate is set too high, it leads to a large policy update. Large policy updates can cause the agent to transit into a bad policy and sort of fall off the cliff. In some cases, it may cause catastrophic forgetting and will be difficult or even impossible to return to a state of optimal policy once again. As a means of improving training stability and restricting large policy updates, PPO utilizes something called the surrogate objective function. With a clipped surrogate object L^{CLIP} or enabling the constrained size of the update, this algorithm approximates the policy gradient update. The clipped surrogate objective is defined as:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, clip\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\Big)A_t\Big)\right] \tag{2.10}$$

where the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the ratio of selecting an action given a state under a new policy to selecting an action under the old policy. This ratio is clipped from a specific range which is [1 - ϵ, 1 + ϵ], where ϵ regulates the size of the clipping. If the ratio is outside this range, it signifies that the updated policy is deviating too much and as a result, the update is clipped to maintain stability. A_t is the advantage function. The advantage function is a measuring tool for assessing the goodness of an action compared to the average action defined by the policy π and determines whether a policy change is necessary. The advantage function A_t is represented as:

$$A(s_t, a_t) = \hat{Q}(s_t, a_t) - V(s_t) \tag{2.11}$$

The PPO policy parameters θ are updated by applying gradient ascent on the surrogate objective function:

$$\theta_{\text{new}} = \arg\max_{\theta'} L^{\text{CLIP}}(\theta') \tag{2.12}$$

2.1.7 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) [17] is another one of the most popular state-of-the-art reinforcement learning algorithms that have been recently considered to be one of the most efficient algorithms for robotics applications. It is a model-free off-policy reinforcement learning algorithm that makes an update to their policy using data generated under different policies. This algorithm uses a maximum entropy RL with the objective of finding the optimal policy that maximizes long-term entropy and long-term reward. SAC seeks to maximize the expected return plus the entropy of the policy. Initially, the first term of the objective function can be considered as the expected value of the sum of rewards that the agent receives given a state and action. The objective function can be represented as:

$$J(\pi) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)] \tag{2.13}$$

Where $r(s_t, a_t)$ is the reward received by an agent after executing an action from state s_t. γ is the discount factor. The second term deals with the entropy bonus. As per the principle of maximum entropy, we aim to find the distribution with the maximum entropy. In many RL algorithms, there is a possibility that an agent may converge to a local optimal state. A maximum entropy objective function enables the agent to search for distributions with maximum entropy by adding the maximum entropy to the objective function. By incorporating the maximum entropy principle, the system has to search for the entropy as well, it enables more exploration by discouraging the policy from becoming too deterministic, and chances to avoid converging to local optima are higher. The SAC objective function with maximum entropy is:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) + \alpha H(\pi(.|s_t))\right] \tag{2.14}$$

Where α is the hyperparameter known as temperature which controls the relative importance of the entropy. $H(\pi(.|s_t))$ is the entropy of the policy. To optimize the objective function the SAC algorithm utilized three different networks:

In a **value network**, the state value function V, which is parameterized by ψ, is trained by minimizing the following error:

$$J_V(\psi) = \mathbb{E}_{s_t \sim D, a_t \sim \pi_\phi}\left[\frac{1}{2}\big(V_\psi(s_t) - Q_\theta(s_t, a_t) - \alpha \log \pi_\phi(a_t|s_t)\big)^2\right] \tag{2.15}$$

where V_ψ is the value function, Q is the state-action value function, α is the temperature parameter, π_θ is the policy function, and D is the replay buffer.

In a Q **network**, the following objective function is minimized:

$$J_{\mathrm{Q}}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim D}\left[\frac{1}{2}\left(Q_\theta(s_t, a_t) - (r(s_t, a_t)) + \gamma V_{\bar{\psi}}(s_{t+1})\right)^2\right] \tag{2.16}$$

For all state-action pairs in the replay buffer, the objective function of a Q Network minimizes the squared difference between the Q function prediction and the sum of the immediate reward and the discounted expected value of the next state.

In a **Policy network** π, training is done by minimizing the following error:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}\left[D_{KL}\left(\pi_\phi(.|s_t)\middle|\middle|\frac{exp(Q_\theta(s_t,.))}{Z_\theta(s_t)}\right)\right] \tag{2.17}$$

Where the D_{KL} is the. Kullback-Leibler Divergence. To put it simply, we want our Policy function distribution to resemble the exponentiation distribution of our Q function normalized by another function Z.

The policy network, parameterized by ϕ, is trained using the objective function:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}[\mathbb{E}_{a_t \sim \pi_\phi}[\alpha \log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t, a_t)]] \tag{2.18}$$

a_t is sampled by implementing the re-parametrization trick to make sure that sampling from the policy is a differentiable process:

$$a_t = f_\phi(\epsilon_t; s_t) \tag{2.19}$$

The final objective function yeilds to:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D, \epsilon_t \sim \mathcal{N}}[\alpha \log(\pi_\phi(f_\phi(\epsilon_t; s_t|s_t))) - Q_\theta(s_t, f_\phi(\epsilon_t; s_t))] \tag{2.20}$$

2.1.8 Deep Deterministic Policy Gradient (DDPG)

The deep deterministic policy gradient is another one of the most widely used reinforcement learning algorithms that are specifically optimized for its applications in control systems and robotics because of its robust performance while dealing with environments with continuous state and action space. It is an off-policy, online, and model-free algorithm that has an actor-critic reinforcement learning agent that searches for an optimal poly that maximizes the expected cumulative long-term reward [14].

As DDPG is an actor-critic method, it generally utilizes two neural networks to implement the actor-critic architecture and makes use of experience replay and target experience replay and target networks to stabilize training. The first neural network deals with the policy/actor-network. This network takes the states of the environment as input and simply generates a specific action given the states. The actor in this case is deterministic and aims to approximate the optimal policy deterministically as a function that maps states to action. Given a state s, the actor networks generate an output of action a:

$$a = \mu(s|\theta^{\mu}) \tag{2.21}$$

where μ is the policy function and θ^{μ} is the actor-network parameter.

The second neural network is the critic network with weights θ^{Q} which approximates the action-value function. In other words, the critic network takes a state-action pair as an input and outputs a Q-value which evaluates the quality of action taken by an agent given a state s, by following a policy μ:

$$Q(s, a|\theta^{Q}) \tag{2.22}$$

where, Q is the Q-value and θ^{Q} is the critic network parameters.

During the training phase of DDPG, the actor-network (policy) and and the critic network (value) are updated at each timestep. The critic update involves minimizing the loss between the Q-target values and the predicted Q-values. The target Q-values are computed using the following Bellman equation:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{2.23}$$

where r_i is the reward, γ is the discount factor, Q' is the target critic network, μ' is the target actor-network, $\theta^{Q'}$, and $\theta^{\mu'}$ are the target network parameters. After that, the goal is to minimize the mean-squared loss between the target Q-value and the predicted Q-value:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2. \tag{2.24}$$

Gradient ascent is performed on the expected Q-values to update the actor-network. The goal is to maximize the expected Q value by updating the actor parameters using the following policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i} \tag{2.25}$$

where J is the objective function, N is the batchsize, $\nabla_a Q$ is the Q-value gradient with respect to action, and $\nabla_{\theta^\mu}\mu$ is the gradient of policy with respect to its parameters.

Similar to other many RL algorithms, to stabilize training DDPG also utilizes an experience replay. This replay buffer is used to store experiences to update NN parameters. During each trajectory roll-out, all the transition tuples which include current state, current action, next state, and reward (s, a, s', r) are stored in a finite-sized replay buffer from where mini-batches of experiences are sampled to update the policy and value networks.

Additionally, DDPG also uses soft target updates for both its policy and value networks where the target networks are updated by slowly tracking the learned networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{2.26}$$

Where θ', θ are the target network parameters and learned network parameters respectively and τ is the soft update factor.

In the context of reinforcement learning with discrete action spaces, exploration is typically facilitated through the probabilistic choice of random actions, utilizing mechanisms like epsilon-greedy or Boltzmann exploration strategies. However, it is not the same case while dealing with continuous state-action space. In the environments where state and action spaces are continuous in nature, the exploration is achieved by adding noise of the action itself which serves as a means to explore more diverse actions. There are various processes to introduce noises to the action space however, the authors in the DDPG paper have employed the **Ornstein-Uhlenbeck Process** to integrate noise (N) into the action output, thus enhancing the model's exploration ability.

$$a_t = \mu(s_t|\theta^\mu) + \mathbf{N} \tag{2.27}$$

2.1.9 Twin Delayed DDPG (TD3)

The deep deterministic policy gradient (DDPG) approach displayed very good results with tasks involving continuous state and action spaces and has been applied in the field of robotics and control systems. However, DDPG has a major drawback of unstable training phases and to produce robust performance, it relies heavily on finding out the correct and optimal hyperparameters. The algorithm has a tendency to cause function approximation errors and overestimation bias of the Q-values of the critic network which in turn causes the agent to get stuck into a local optima. TD3 (Twin Delayed Deep Deterministic Policy Gradient) [15] algorithm is considered as the successor of the DDPG algorithm primarily because it addresses the issue of function approximation errors and overestimation bias of DDPG by introducing three new key features:

1. **Clipped Double-Q Learning:** TD3 algorithm's first major change from DDPG is that it employs two critic networks instead of one. During the critic update, TD3 uses this clipped double Q learning method to take the smallest value estimation between the two critic network predictions. By adding this feature, TD3 ensures a more stable function approximation by the underestimation

of Q-values. It results in underestimation bias, but this is not a problem since low values do not propagate through the algorithm, as opposed to overestimated values.

For each critic network:

$$y_i = r_i + \gamma \min_{j=1,2} Q'_j(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'}))|\theta^{Q'_j}) \tag{2.28}$$

where y_i is the target Q-value, Q'_j is the j-th target critic network, μ' is the target actor network, $\theta^{\mu'}$ and $\theta^{Q'_j}$ are the target networks parameters.

The loss function of each critic is as follows:

$$L_j = \frac{1}{N}\sum_i (y_i - Q_j(s_i, a_i|\theta^{Q_j}))^2 \quad \text{for } j = 1, 2 \tag{2.29}$$

2. **Delayed Updates:** The second feature that is introduced in TD3 is the delayed updates of the actor-network. In an actor-critic architecture, sometimes the policy overfits to the noise in value estimation. This happens when the value network (critic) is not stable enough and is used to update the policy network, thereby causing the agent's policy to continuously get worse since it is updating on states with a lot of errors. To solve this issue TD3 updates the policy network (actor) less frequently than the value function network (critic). This helps the value network to become more stable by reducing value estimation variance with every time step and then after a fixed number of steps for instance, after d steps, it is used to update the policy network.

3. **Target Policy Smoothing:** It is common for deterministic policy methods to produce target values with a high variance while updating the critic. TD3 employs a noise regularisation strategy which is called the target policy smoothing to address this issue. By adding noise to the target policy, and then averaging over mini-batches, the smoothing method reduces variances. This results in higher values for actions that are more robust. The target action is perturbed

by a clipped noise:

$$\tilde{a} = \mu'(s_{i+1}|\theta^{\mu'}) + \text{clip}(\epsilon, -c, c) \tag{2.30}$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$, c is a clip value.

The Q-value targets are then computed with this noise:

$$y_i = r_i + \gamma \min_{j=1,2} Q'_j(s_{i+1}, \tilde{a}|\theta^{Q'_j}) \tag{2.31}$$

Similar to DDPG, the actor-network is updated to maximize the expected Q-value but for TD3, with the minimum value of the two critic networks:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a \min_{j=1,2} Q_j(s, a|\theta^{Q_j})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i} \tag{2.32}$$

In summary, this chapter establishes foundational knowledge about some of the essential core concepts and methodologies for reinforcement learning that serve as a crucial backdrop for the upcoming chapters. The idea behind the balance between exploration and exploitation, along with a brief review of on/off-policy methods lays out a framework to explain some insights and findings about the results for comparing these traditional RL methods with generative flow networks that are discussed in the later chapters.

Chapter 3

GFlowNets and its Variants: A Comprehensive Overview

As a follow-up to the overview of reinforcement learning in Chapter 2, this chapter delves into the detailed background analysis of GFlowNets and one of its latest variants named CFlowNets. Furthermore, this chapter also contains a concise overview highlighting the difference between GFlowNets and Reinforcement Learning.

3.1 GFlowNets

Generative flow networks, or GFlowNets [6], is the next step in exploratory control tasks and have the potential to be used as a substitute for Reinforcement Learning. In its most basic form, it is a trained stochastic policy or generative model. GFlowNets forward sampling policy aims to generate distribution and sample different candidate objects through a sequence of constructive steps with probability proportional to rewards over terminal states. The reason why it is called a stochastic policy is that for each state in the environment, the agent does not have a precisely well-defined action to take from a given state; instead, the agent has a probability distribution to take from that given state. GFlowNets is an emerging technique for active learning diverse candidate sampling.

To illustrate how GFlowNets works in lay terms, Bengio et al. [6] compare it to constructing an object such as a Lego boat. While constructing a Lego boat, we go through a sequence of steps by adding each block to its proper place to finally get the final object. Let us compare it with the GFlowNets setting. The state of GFlowNets signifies the objects' current state (partially constructed object), and the trajectories define how the objects could have been built in many different ways. There are two kinds of sampling policies in GFlowNets, which are the forward-sampling policy and the backward-sampling policy. Adding Lego blocks one by one can be considered GFlowNet's forward-sampling policy for this example. On the other hand, a backward-sampling policy of GFlowNets removes one Lego block at a time, correcting our attempt to build a Lego boat. Finally, when the Lego object is done being constructed, in other words, when the terminal states of the trajectories have been reached, we will get a reward. The reward is high if the terminal object resembles the boat and lower if it does not. By utilizing GFlowNet's forward-sampling policy, we can generate an exponentially large number of boat-looking objects from different trajectories that can be sampled.

3.1.1 Definition of GFlowNets

The formal definition of GFlowNets is that it is a stochastic policy or generative model. The term "Flow" refers to the distribution of unnormalized probabilities across different states and state transitions of the network as it constructs objects. The terminology is derived from the concept of flow networks where water particles enter into the network through some initial state s_0 and exit through terminal nodes $x \in \mathcal{X}$. To clearly define GFlowNets, including its policy parameters and output, [19] outlines how the idea of GFlowNets can be specialized. It is a learning algorithm that contains the following features:

- A forward-sampling policy that provides the forward action distribution $P_F(-|s)$, where s is any state that is not terminal ($s < s_n$).

- A backward-sampling policy $P_B(-|s)$ that provides the backward action distribution using which one can sample backward from any given state s.

- The initial state flow estimator $Z = F(s_0) = \sum_x R(x)$, where s_0 is the initial state and $\sum_x R(x)$ is the sum of all the rewards i.e. the sum of all the terminal flows at the terminal states.

- The edge flow function $F(s \to s')$, which estimates the flow of the edge between two states i.e. the transition between states s to s'.

- $F(s)$ which is the state flow function that estimates the flow through any particular state.

- A self-conditional flow function $F(s|s')$ estimates flow through s if trajectories only pass through $s' < s$.

- A training objective function such as Flow-matching objective, Detailed balance objective, and Trajectory balance objective, etc.

3.1.2 GFlowNets Architecture

Directed acyclic graphs (DAGs) are used to represent trajectory state transitions in GFlowNets shown in the Figure 3.1. In the DAG, there are both state spaces $\mathcal{S}$ and action spaces $\mathcal{A}$. The nodes in the graph structure denote the different states $s_t \in \mathcal{S}$, and the edge that connects different states refers to the actions $a_t \in \mathcal{A}$. Typically, a neural network samples the forward-going constructive actions while constructing any sample candidate objects. The sequence of different actions $(a_0, a_1, a_3, \dots)$ taken by the agent creates a trajectory τ, which is essentially a sequence of states $(s_0, s_1, s_3, \dots)$. Each executed action given a state s in the trajectories transits to a next state s' with transition probability $s \to s'$ which represents the the probability that a water particle going through the state s will reach the next state s'. Every trajectory begins with an initial state s_0 (presented with a triangle shape in Figure 3.1) and ends with a terminal state s_n (if the object in question has n number of elements). A state can have multiple parents; in other words, there could be different trajectories τ that lead to the same state. As an example, you can combine Lego blocks in many different ways to achieve the same result. The ending nodes in a generated trajectory is called the terminal states s_n. When a sample object is constructed, it initiates an "exit" action T (red transition in Figure 3.1), which leads it to the terminal state, after which we can get a reward $R(x)$, and GFlowNets sample the terminal object with probability proportional to this reward. A basic difference between typical generative models and GFlowNets is that generative models have an objective of fitting finite datasets, while GFlowNets have an objective of matching reward functions where the flow has to be constrained at each terminal state x to be equal to $R(x)$.

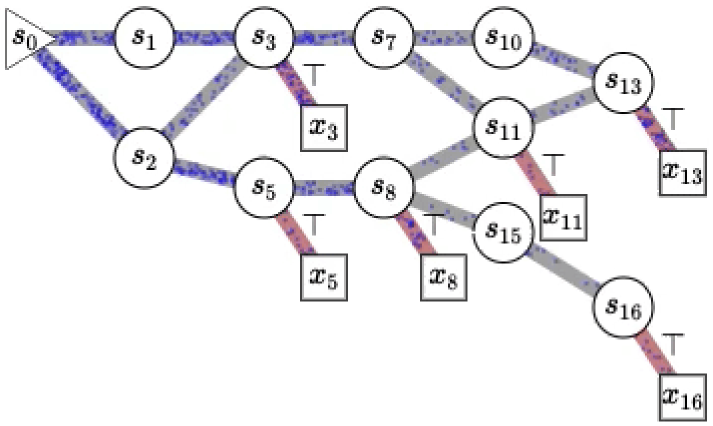


Figure 3.1: Flow Network DAG Illustration [20]

To represent the architecture of GFlowNets in terms of Neural Networks, we can look into Figure 3.2. A GFlowNet consists of a neural network that has the capability of outputting a stochastic policy $\pi(a_t|s_t)$, where $s_t \in \mathcal{S}$ is the current state of the object and $a_t \in \mathcal{A}$ is the set of allowed action from state s_t. Choosing the possible action from the state s_t leads to the next state, which is s_{t+1}. In other words, the policy will result in a forward transition probability $P_F(s_{t+1}|s_t)$. In every step, the same neural network is utilized again and it produces a stochastic output a_t, from

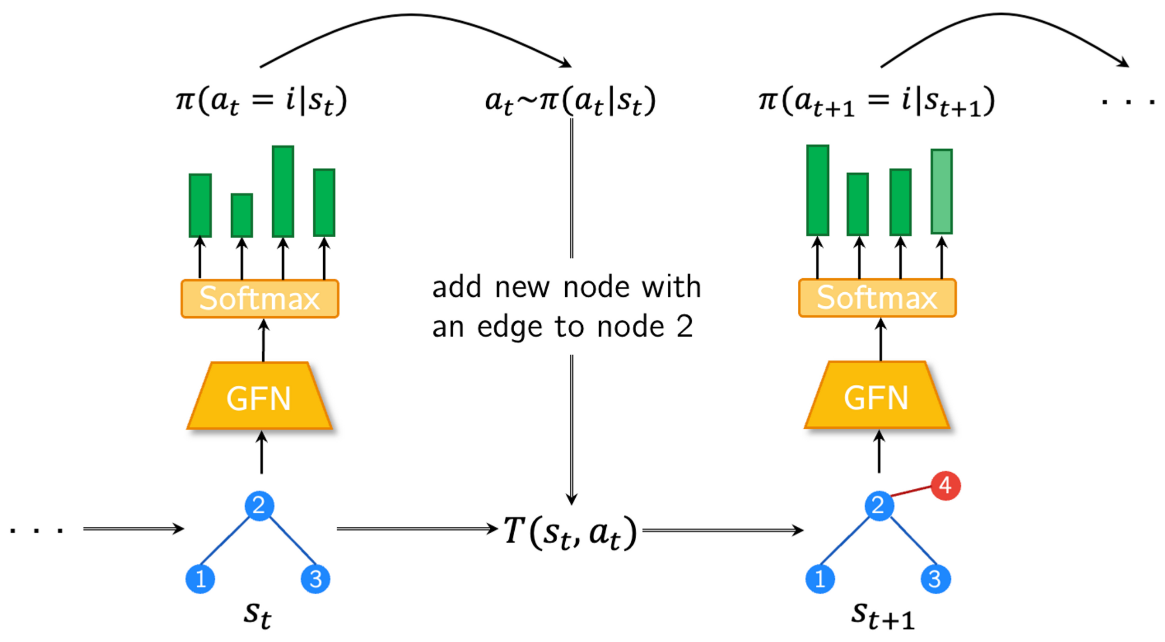


Figure 3.2: GFlowNets Architecture [20]

which the next state $s_{t+1} = T(s_t, a_t)$ is derived. In Figure 3.2, we can see that according to the policy, after choosing action a_t of adding a new node, given s_t, we get the forward transitional probability of the next state which is adding a new node 4 to the pre-existing node 2.

3.1.3 GFlowNets Training

As previously mentioned, the GFlowNets training objective involves matching a reward function $R(x)$ (to construct objects with a probability proportional to $R(x)$) rather than fitting a finite dataset like standard generative models. There is no target distribution for the GFlowNet's policy, and we only have access to the unnormalized probabilities $R(x)$, which is the reward function itself. Therefore, we must adjust the flow into the terminal state x to match the value of R(x). If the flows are fixed/matched in the terminal states, then we should have the state flow estimator as $Z = F(s_0) = \sum_x R(x)$, i.e., the terminal node flows should equal the initial flow at the root s_0 which is also called the "Flow matching constraint". In this case, the GFlowNets can be trained by estimating the state flow function $F(s)$ and edge flow function $F(s\rightarrow s')$ to satisfy the flow matching constraint given that we need to fix the desired flow in the terminal states, and then sampling with GFlowNets will generate terminal states x with probability $\frac{R(x)}{\sum_x' R(x')}$. We can sample the trajectories by utilizing the forward policy to achieve the forward transition probabilities:

$$\pi(a|s) = \pi(s\rightarrow s' = T(s,a)|s) = P_F(s'|s) = \frac{F(s\rightarrow s')}{\sum_{s''} F(s\rightarrow s'')} \tag{3.1}$$

In a similar manner, the backward trajectories are represented as:

$$P_B(s|s') = \frac{F(s\rightarrow s')}{\sum_{s''} F(s''\rightarrow s')} \tag{3.2}$$

The GFlowNets Foundation paper [6] proposed several training objectives for the GFlowNets training framework. Such as:

Flow-matching objective. Based on the convention that if no action turns $s\rightarrow s'$, the edge flow function is $F(s\rightarrow s') = 0$, so we can present the flow matching constraint into a loss that measures the mismatch between the sum of entering flows through a state and the sum of outgoing flows from the state:

$$\sum_{s'} F(s'\rightarrow s) = \sum_{s"} F(s\rightarrow s") \tag{3.3}$$

The GFlowNets empirical paper also used the following training objective:

$$(\log(\sum_{s'} F(s'\rightarrow s)) - \log(\sum_{s"} F(s\rightarrow s")))^2 \tag{3.4}$$

To avoid neural network numerical issues induced by large state flow functions and edge flow functions in the earlier portion of the trajectory, log is introduced in the training objective where rather than matching flows, we match their logarithms (matching relative probabilities). The final log-scale training objective for GFlowNets:

$$\mathcal{L}_{\theta,\epsilon}(\tau) = \sum_{s'\in\tau\neq s_0} \left(\log \left[\epsilon + \sum_{s,a:T(s,a)=s'} expF_\theta^{\log}(s,a) \right] \right.$$

$$- \log \left[\epsilon + R(s') + \sum_{a' \in \mathcal{A}(s')} expF_\theta^{\log}(s,a) \right] \Bigg)^2 \tag{3.5}$$

where ϵ is a small positive constant to prevent taking the logarithm of zero. Introducing the log scale gives balances the different magnitudes of flow values across the network. As it turns out, matching the logs of the flows equals making the incoming-to-outgoing flow ratio closer to 1. $R(s')$ is zero for internal states. Only in terminal states, does the inclusion of the reward generate samples from a distribution where the probability of sampling a state is proportional to its associated reward.

Detailed balance objective and backward policy. Another training objective proposed by Bengio et al. [6] is called the Detailed balance objective and backward policy. Instead of using a flow-matching constraint, a neural network outputs a softmax for all possible actions in each state, which yields in the forward sampling policy $P_F(s'|s)$. Here $s' = 0$ if no action is taken to make a transition to s'. It also enables a backward sampling policy $P_B(s|s')$, which provides a set of parent states $s_t \in \mathcal{S}$ of the current state s'. Flow matching is indirectly achieved by this objective as well:

$$F(s)P_F(s'|s) = P_B(s|s')F(s') \tag{3.6}$$

Trajectory Balance Objective. The trajectory balance training objective takes the concept of the detailed balance objective and extends it to entire trajectories instead of individual state transitions. This training objective states that the forward probability of an entire trajectory that starts from the initial state s_0 and ends at terminal state s_n has to be equal to the backward probability of that trajectory that starts from s_n to s_0.

$$F(s_0) \prod_{t=1}^{n} P_F(s_t|s_{t-1}) = R(s_n) \prod_{t=1}^{n} P_B(s_{t-1}|s_t). \tag{3.7}$$

where, the probability of trajectory under the forward policy is calculated by $F(s_0)\prod_{t=1}^{n} P_F(s_t|s_{t-1})$ and the backward probability which starts from the reward at terminal states is represented by $R(s_n)\prod_{t=1}^{n} P_B(s_{t-1}|s_t)$.

3.1.4 GFlowNets vs. RL

In this study, we are trying to make a comparative evaluation between generative flow networks and RL. GFlowNets and RL differ in that RL policies are designed to find paths that maximize return, whereas GFlowNets are trained so that they sample terminal states matching a desired unnormalized probability function from a distribution of all possible paths. Rather than seeking to maximize this function, GFlowNets aims to generate outputs in proportion to it, creating a diverse set of high-reward solutions. The problems they solve are, therefore, different. In addition, the value function in RL tries to measure the expected cumulative future reward that an agent can obtain from a particular state or state-action pair. In contrast, the flow function of GFlowNets estimates the proportion of all future rewards that can be attributed to passing through a particular state or transition. Through the network, it quantifies how much 'reward mass' should pass through a particular point.

Generally, standard return maximization techniques lead to a single sequence that maximizes returns. However, in some cases, we may want to sample a diverse set of high-return solutions. Typically, an RL agent aims to find a policy that maximizes the expected sum of rewards by maximizing the value function. As a result, because of this inclination towards higher rewards, RL sometimes tends to learn policies that exploit the known rewarding paths instead of exploring better and unknown paths. GFlowNets, on the other hand, aims to generate and sample from the entire distribution of solutions/outcomes in an active learning approach, giving more probability to high-reward paths while still maintaining diversity in the solutions. A reinforcement learning agent seeks out the most rewarding path from a set of possibilities. The question arises, however, if the path found by the agent is a local optima. One of the important problems in RL is getting stuck in locally optimal points due to the objective function (sum of cumulative reward). Let us consider the soft actor-critic algorithm, which is a fast DRL algorithm being applied to robotics. This algorithm takes advantage of entropy regularization in its policy, which helps trade-offs between exploration and exploitation. Nevertheless, when dealing with large-scale problems, we have to deal with balancing exploration strategies with exploitation so that the agent's policy does not become too deterministic. The idea of GFlowNets, as claimed by Bengio et al. [6], is to solve the problem of exploration in RL. GFlowNets generate a distribution over all possible paths and sample from the most rewarding paths with a higher probability which could make them more sample efficient than RL, while RL sticks to the most rewarding path which can be a local optimum. Because GFlowNets are not solely focused on the highest-reward path but rather on sampling across a range of high-reward paths, they are inherently more exploratory. This major edge GFlowNets has over RL could play a vital role in machine adaptability through proper exploration and could indeed be why GFlowNets may perform better than RL in a continual learning environment where a machine adapts to changing conditions.

3.2 CFlowNets

A number of new studies have been conducted on flow network-based implementations since GFlowNets were introduced. A potentially useful extension of GFlowNets has been tokenized as CFlowNets [7], which stands for generative Continuous Flow Networks. This paper discusses a shortcoming of GFlowNets in the context of exploratory continuous control tasks. To calculate the flow-matching loss, GFlowNets form a DAG and traverse each node's inflows and outflows. Each node in this DAG represents a state and the edges between the states represent the set of allowed actions. A GFlowNet formulation deals with discrete tasks, in which there are a limited number of state and action pairs, and each edge represents one discrete action. It is important to note, however, that real environments have continuous states and action spaces for most tasks. Additionally, multimodal reward distributions may be present in these environments, which entails more exploration of diversity. In the GFlowNets foundations paper, the authors did not discuss in detail how GFlowNets theoretical formulation can work in continuous state-action spaces. The only thing that was mentioned by Bengio et al. [6] related to continuous scenarios is that if the GFlowNets need to be implemented in continuous tasks, integrals can be used instead of sums for continuous variables and for deriving training objective function they advocated using integrable densities and detailed balance criterion. Nevertheless, this concept has not yet been experimentally tested, and GFlowNets have not yet been shown to handle continuous tasks in experiments. To solve this issue with CFlowNets, continuous control tasks can be addressed to create policies that are proportional to continuous rewards. In other words, Li et al. [7] claimed to have extended the theoretical formulation and flow-matching theorem of the GFlowNets foundation to make it appropriate for continuous scenarios.

3.2.1 CFlowNets Definition

In its most basic form, CFlowNets architecture resembles the architect of GFlowNets in many aspects. Since the CFlowNets are developed with continuous scenarios (S, A) in mind:

- CFlowNets state space is a continuous state space denoted as $\mathcal{S}$, and the continuous action space is represented with $\mathcal{A}$.
- The agent selects an action to make a transition $a_t : s_t \rightarrow s_{t+1} \in \mathcal{A}$.
- The continuous execution of actions results in a sequence of sampled elements of $\mathcal{S}$ which forms a trajectory $\tau = (s_1, s_2, s_3, ..., s_f)$. This trajectory is an acyclic one.
- The acyclic trajectory τ is as any sampled trajectory starting at s_0 and terminating at s_f. The initial root state is denoted by s_0, and the final state by

s_f.

- Terminating flow is represented as $F(s{\rightarrow}s_f)$ and a transition $s{\rightarrow}s_f$ is defined as the terminating transition.

- Continuous flow networks are composed of the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{F})$.

3.2.2 Continuous Flow Definitions

Taking into account that the sampled trajectory τ is acyclic, the authors defined the parent set $\mathcal{P}(s_t)$ which contains the set of all the parent states of s_t. By parent state of s_t, we mean all the states that can make a transition by selecting an action $a \in \mathcal{A}$ to the state s_t. In a similar manner, the child states $\mathcal{C}(s_t)$ are defined as the set of resulting next states after taking an action from the state s_t. Based on the assumption that the flow functions $F(s, a)$ is *Lipschitz continuous* and given a state pair (s_t, s_{t+1}), only a unique action a_t can make that transition i.e., $s_t = g(s_{t+1}, a_t)$ where $g(s_{t+1}, a_t)$ is a transition function, Li et al. [7] formulated the following continuous flow definition:

1. **Continuous State Flow:** In a continuous state flow, $F(s)$ is the integral of the trajectory flows that pass through state s:

$$F(s) = \int_{\tau: s \in \tau} F(\tau) d\tau \tag{3.8}$$

2. **Continuous Inflows**: Given a state s_t, the continuous state inflows are the integral of flows originating from its parent states:

$$\int_{s \in \mathcal{P}(s_t)} F(s{\rightarrow}s_t) ds = \int_{s: T(s,a)=s_t} F(s, a) ds = \int_{a: T(s,a)=s_t} F(s, a) da \tag{3.9}$$

3. **Continuous Outflows:** Given a state s_t, the continuous state outflows are the integral of flows that pass through s_t and go outwards towards its child states after selecting from a set of allowed action $a \in \mathcal{A}$:

$$\int_{s \in \mathcal{C}(s_t)} F(s_t{\rightarrow}s) ds = \int_{a \in \mathcal{A}} F(s_t, a) da \tag{3.10}$$

Based on these three definitions, Li et al. [7] defined the forward transition probability as the ratio of the flow from state s_t to s_{t+1} denoted as $F(s_t \to s_{t+1})$ to the total flow through the current state s_t, denoted as $F(s_t)$:

$$P_F(s_{t+1}|s_t) := P(s_t \to s_{t+1}|s_t) = \frac{F(s_t \to s_{t+1})}{F(s_t)} \tag{3.11}$$

This ratio represents how much of the total flow at state s_t is forwarded towards the next state s_{t+1}. Similarly, the backward transition probability is as:

$$P_B(s_t|s_{t+1}) := P(s_t \to s_{t+1}|s_{t+1}) = \frac{F(s_t \to s_{t+1})}{F(s_{t+1})} \tag{3.12}$$

In terms of a continuous task $\mathcal{S}, \mathcal{A}$, for any sample trajectory $\tau = (s_1, s_2, ..., s_n)$ the forward probability will be the product of each individual forward transitions in the sequence:

$$\forall \tau = (s_1, s_2, ..., s_n), P_F(\tau) := \prod_{t=1}^{n-1} P_F(s_{t+1}|s_t) \tag{3.13}$$

Similarly, the backward trajectory probability $P_B(\tau)$:

$$\forall \tau = (s_1, s_2, ..., s_n), P_B(\tau) := \prod_{t=1}^{n-1} P_B(s_t|s_{t+1}) \tag{3.14}$$

3.2.3 Training Framework

In the case of a non-negative flow function $\hat{F}$ that accepts a state and action pair as input, $\hat{F}$ corresponds to a flow if the conditions for continuous flow matching are met:

$$\forall s' > s_0, \hat{F}(s') = \int_{s \in \mathbf{P}(s')} \hat{F}(s \to s') ds = \int_{s : T(s,a) = s'} \hat{F}(s, a : s \to s') ds \tag{3.15}$$

$$\forall s' > s_f, \hat{F}(s') = \int_{s" \in \mathbf{C}(s')} \hat{F}(s' \to s") ds" = \int_{a \in \mathcal{A}} \hat{F}(s', a) da \tag{3.16}$$

If reward environments are sparse, we can train a flow network that meets the flow-matching conditions to obtain the target flow. Using this as a basis, we can derive CFlowNets' continuous loss function as follows:

$$\mathcal{L}(\tau) = \sum_{s_t=s_1}^{s_f} \left(\int_{s_{t-1}\in\mathcal{P}(s_t)} F(s_{t-1}\to s_t)ds_{t-1} - R(s_t) - \int_{s_{t+1}\in\mathcal{C}(s_t)} F(s_t\to s_{t+1})ds_{t+1} \right)^2 \tag{3.17}$$

Nevertheless, CFlowNets implementations cannot directly apply this continuous loss function. The overall training framework of CFlowNets can be classified into three parts shown in 3.3. In the first part, an agent interacts with the environment and how actions are sampled in a continuous action space is discussed. In the second part, flow sampling is done for each state, and in the third part, the CFlowNets are trained using the continuous loss function.

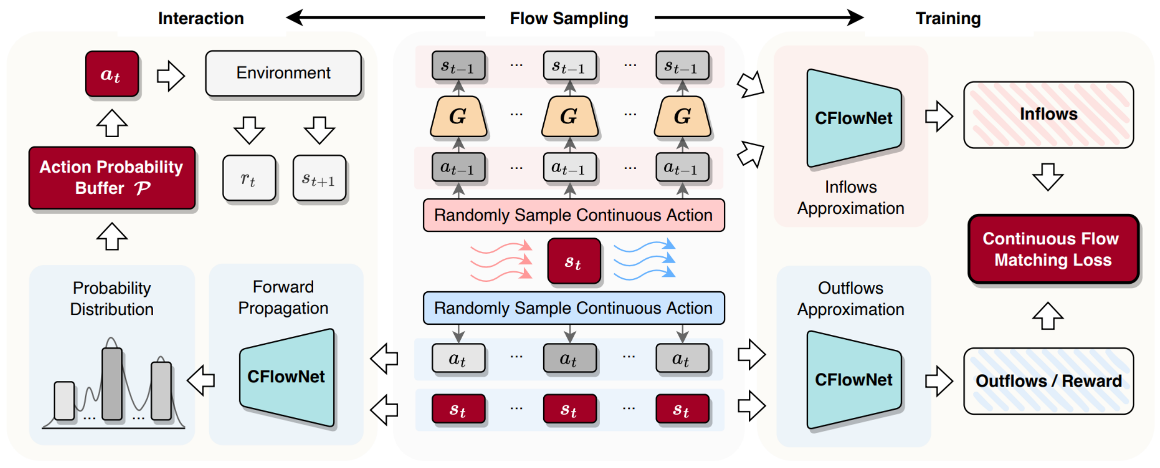


Figure 3.3: CFlowNets Training Framework [7]. The leftmost part represents the action selection procedure. The middle part is the flow-matching approximation visualization and the rightmost section shows the Continuous Flow-Matching Loss, which is utilized for training.

1. **Action Selection Procedure:** The first part of the training framework focuses on the stage where the agent interacts with the environment. This interaction phase involves selecting an action from the action probability buffer, and this process is called the action selection procedure. The goal is to obtain complete trajectories τ by iteratively sampling actions from the probability distribution $a_t \sim \pi(a_t|s_t)$, with the help of CFlowNets. However, the action space is con-

tinuous in this case. As a result, it is challenging to derive the precise action probability distribution function using the flow network. Therefore, in order to resolve this issue, the agent first uniformly samples M number of actions from the action space $\mathcal{A}$ at each state s_t. Then, it generates an action probability buffer $\mathcal{P} = [F(s_t, a_i)]_{i=1}^{M}$, which approximates action probability distributions. Finally, the agent samples an action from the buffer $\mathcal{P}$ based on the probabilities of all the actions. The actions with higher $F(s_t, a_i)$ will be sampled with a higher probability. As a result, the agent can approximate sample actions from a continuous distribution based on the probability of each action in the distribution. After an action is selected, the agent interacts with the environment to update its state. The process is repeated multiple times until the entire trajectory is sampled. Similar to the previous step, the whole process is repeated multiple times to generate trajectory sets that are stored in the buffer β.

2. **Flow Matching Approximation:** The second part of the training framework is known as the Flow Matching Approximation. After obtaining a collection of complete trajectories β after the environment interaction phase, to satisfy the flow matching condition, we need to ensure that the total inflow $\int_{a:T(s,a)=s_t} F(s, a)da$ of any node s_t, is equal to the total outflow $\int_{a\in\mathcal{A}} F(s_t, a)da$. However, calculating the inflows and outflows may pose a challenge since the inflows and outflows are continuous. As a result, directly calculating these integrals (inflows and outflows) is not feasible due to the infinite number of possible actions in the continuous action space. Therefore, an approximation method is utilized by randomly and uniformly sampling K actions from the continuous action space $\mathcal{A}$ and calculating the corresponding $F(s_t, a_k)$ values (for $k = 1$ to K) as the outflows. For outflows, the following approximation is used, where µ(A) indicates the measure of the continuous action space $\mathcal{A}$:

$$\int_{a\in\mathcal{A}} F(s_t, a)da \approx \frac{\mu(\mathcal{A})}{K} \sum_{k=1}^{K} F(s_t, a_k) \tag{3.18}$$

A deep neural network G (named "retrieval" neural network) is constructed to sample the inflows which is parametrized by ϕ. This NN G takes a state-action pair (s_{t+1}, a_t) as the input while it outputs the parent state s_t. It is trained based on the trajectory buffer β with the MSE loss. In other words, the sampled actions together with the current state are fed to G as the input to estimate the parent states. The network G can be pre-trained and occasionally updated with new sampled trajectories to maintain accuracy. Based on these, the inflows of each state are approximately determined by utilizing the following approximation:

$$\int_{a:T(g(s_t,a),a)=s_t} F(g(s_t, a), a)da \approx \frac{\mu(\mathcal{A})}{K} \sum_{k=1}^{K} F(G_\phi(s_t, a_k), a_k) \tag{3.19}$$

3. **Continuous Loss Function:** Finally, based on the approximate inflows and outflows in equation 3.18 and 3.19, we can train CFlowNets based on the continuous flow matching loss function:

$$\mathcal{L}_\theta(\tau) = \sum_{s_t=s_1}^{s_f} \left[\sum_{k=1}^{K} F_\theta(G_\phi(s_t, a_k), a_k) - \lambda R(s_t) - \sum_{k=1}^{K} F_\theta(s_t, a_k) \right]^2 \tag{3.20}$$

λ is the scaling factor used to scale the summation of the sampled flows appropriately, considering the size of the continuous action space $\mathcal{A}$. The loss function 3.20 can also be represented in log-scale inspired by GFlowNets log-scale loss function [21] to avoid numerical problems:

$$\mathcal{L}_\theta(\tau) = \sum_{s_t=s_1}^{s_f} \left[\log \left[\epsilon + \sum_{k=1}^{K} expF_\theta^{\log}(G_\phi(s_t, a_k), a_k) \right] \right.$$

$$-\log\left[\epsilon + \lambda R(s_t) + \sum_{k=1}^{K} expF_\theta^{log}(s_t, a_k)\right]\Bigg]^2 \quad (3.21)$$

As a whole, CFlowNets training pseudocode can be seen in the following manner:

Algorithm 1 Generative Continuous Flow Networks (CFlowNets) Algorithm [7]

```
1: Initialize: Flow network $\theta$; a pretrained retrieval network $G_\phi$; and empty buffer
   $D$ and $P$
2: repeat
3:     Set $t = 0$, $s = s_0$
4:     while $s \neq$ terminal and $t < T$ do
5:         Uniformly sample $M$ actions $\{a_i\}_{i=1}^{M}$ from action space $A$
6:         Compute edge flow $F_\theta(s_t, a_i)$ for each $a_i \in \{a_i\}_{i=1}^{M}$ to generate $P$
7:         Sample $a_t \sim P$ and execute $a_t$ in the environment to obtain $r_{t+1}$ and $s_{t+1}$
8:         $t = t + 1$
9:     end while
10:    Store episodes $\{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^{T}$ in replay buffer $D$
11:    [Optional] Fine-tuning retrieval network $G_\phi$ based on $D$
12:    Sample a random minibatch $B$ of episodes from $D$
13:    Uniformly sample $K$ actions $\{a_k\}_{k=1}^{K}$ from action space $A$ for each state in $B$
14:    Compute parent states according to $\{G_\phi(s, a_k)\}_{k=1}^{K}$ for each state in $B$
15:    Inflows:
16:        $\log\left[\epsilon + \sum_{k=1}^{K} \exp F_{\log\theta}(G_\phi(s_t, a_k), a_k)\right]$
17:    Outflows or reward:
18:        $\log\left[\epsilon + \lambda R(s_t) + \sum_{k=1}^{K} \exp F_{\log\theta}(s_t, a_k)\right]$
19:    Update flow network $F_\theta$ according to the Continuous Loss Function.
20: until convergence
```

To summarize, GFlowNets theoretical formulations only work in discrete tasks where the agent can only choose from a finite or countably infinite set of possible actions from a discrete state space. No experiments have verified that it can handle continuous actions and state space. On the contrary, CFlowNets have extended the theoretical formulation of previous GFlowNets so that it can adapt to Continuous control tasks and create policies that are proportional to rewards. It proposes a novel training framework that includes action selection procedure, flow approximation algorithm, and the continuous flow matching loss function. Since CFlowNets is more suitable for continuous control tasks, all the comparative analysis experiments for this thesis will be conducted based on the CFlowNets training framework instead of GFlowNets for robotic implementation and fault adaptation.

Chapter 4

Related Works

4.1 Fault diagnosis and Fault-tolerant Strategies

Gao et al. [9] conducted a comprehensive survey on various fault tolerance and fault diagnosis techniques and provided adequate explanations on how these techniques can be used in real-world fault diagnosis. The paper starts with a discussion of what it means for a system to have the capability of fault diagnosis and fault tolerance and proceeds to evaluate its importance, specifically in industrial applications. This survey then focuses on two approaches that are the most commonly used for fault diagnosis: the model-based approach and the signal-based approach. The model-based approach builds a mathematical model to represent or mimic the actual system. A comparison is made between the predicted behavior pattern of the mathematical model and the actual behavior pattern of the system for fault diagnosis. A fault is identified if there are any discrepancies between these two. Gao et al. [9] then provide a detailed explanation of different types of model-based approaches, such as analytical models, data-driven models, and physical models, all for the purpose of fault diagnosis.

The survey also discusses signal-based approaches for fault diagnosis. In signal-based approaches, the signals that are generated from the system are analyzed using statistical analysis, pattern recognition, etc, to detect any type of mechanical malfunction. Several signal processing methods are elaborately discussed, such as wavelet transform [22] (a method of decomposing signals into time-frequency components), Fourier transforms (decomposes a signal into a frequency component), empirical mode decomposition (decomposes a signal into an intrinsic model function), etc. Additionally, the author identifies three types of faults that are most common in machines, which are: 1. Sensor faults (e.g. constant/inaccurate value reading) 2. Actuator faults (e.g., broken/frozen actuator) 3. Plant faults (e.g., disconnection of components).

4.2 Trial-and-Error with select-test-update

Cully et al.[23] proposed a novel approach of trial and error to compensate and adapt for machine faults. This research was inspired by biomimicry, where an injured animal can adapt to its injury through a process of trial and error to figure out optimal movement. Similar to this concept, the authors sought to implement a trial-and-error algorithm in the field of robotics so that a robot could creatively adapt to malfunctions. For the purpose of guiding the trial-and-error algorithm, a pre-computed behavior-performance map was developed. The behavior-performance map consists of around 13,000 different robotic gaits. Using this behavior-performance map, a robot experiencing any physical malfunction will use a simulated behavior that is already predicted to be effective. The effectiveness of a behavior is determined by its predicted performance value that is associated with it. The motivation behind developing such a pre-computed, automatically generated behavior-performance map was based on the notion that animals possess good knowledge about their own search space of possible behaviors when they face an injury. This set of behaviors is developed from past experience. Similarly, a robot should have knowledge about its search space, in other words, a set of behaviors with a performance estimate at its disposal, which will guide its trial-and-error method when a fault occurs.

Whenever a robot faces a malfunction during the action selection procedure, it will be guided by the behavior-performance map. The robot will select and execute an action that has been assigned a high estimated performance value. After executing the action in the environment, the selected behavior will get a new performance rating for that particular task. Throughout a series of tests, the selected behavior's performance estimation values will be updated. This process will be reiterated until the robot identifies a behavior with a performance estimate that exceeds 90% of the best performance predicted for any behavior in the behavior-performance map [23].

The experimentation was done on a hexapod robot with six different conditions. Using the proposed Intelligent Trail-and-Error algorithm with a behavior-performance map, the robot was able to adapt to failure by learning compensatory behavior within a reasonably fast time.

The reason why the trial and error approach was not chosen to be a part of our comparative study is because it takes a fundamentally different approach than CFlowNets and RL methods. Whereas both CFlowNets and RL share a common algorithmic approach to learning policies by directly interacting in the environment, the trial and error approach updates a limited pre-computed behavioural map utility values via a process of iterative trial and error. We conducted our comparative analysis between CFlowNets and standard RL because both these approaches are methodologically consistent which allows for a direct comparison in terms of learning efficiency, adaptation speed, and average reward performance metric. Additionally, the trial-and-error approach utilizes a behavior-performance map of 13,000 robotic gaits that is computed and tailored to a specific hexapod robot and lacks generalization over different robotic simulations. As a result, it is not a good candidate for our comparative study.

4.3 Adaptation using Reinforcement Learning & Meta-RL

Yang et al. [24] in their research proposed a novel adversarial reinforcement learning framework in an effort to increase robot robustness through adaptation in joint malfunction robotic tasks. Instead of focusing on fault recovery on locomotion tasks as in prior studies [23], [25], the primary focus of this study was on joint damage crisis on manipulation tasks. Manipulation tasks lack redundancy and even a minor fault in a manipulation task may limit the action space. For instance, in an in-hand manipulation task, one impaired finger can make the other fingers stuck as well. In the proposed framework, the authors trained their policy on a predefined **q** (*policy for different joint states*) for a number of episodes. Next, they updated **q** for the next policy training iteration based on a search for challenging scenarios (*joint states with poor agent performance*) under the updated policy. Yang et al. experimented with and evaluated the proposed adversarial reinforcement learning framework on the D'Claw robot and the D'Kitty robot. Their experiment included the following joint malfunctions: 1. Inability to change joint angles and 2. Joints acting randomly. The experimental results validated that both real robots have demonstrated increased resiliency to joint damage.

Another research that introduced model-based meta-reinforcement learning for fault adaptation tasks was introduced by Clavera et al. [26]. In their meta RL framework, two adaptive learners were utilized for the algorithm. An initial set of parameters for a generalized dynamics model is learned using model-agnostic meta-learning (MAML) [27]. This algorithm uses a gradient-based adaptive learner (GrBAL) whose dynamics model is represented using a NN and updated using gradient descent. As for the second learner, it is a recurrence-based adaptive learner (ReBAL) that uses an RNN to represent the dynamics model and uses an update rule learned by the recurrent network to perform updates. In order to demonstrate the sample efficiency of both GrBAL and ReBAL, Clavera et al. conducted experiments and reported the average return in differing test environments based on the amount of data used in meta-training. The results were compared with two model-free methods TRPO and MAML-RL. Their proposed GrBAL and ReBAL were far superior in terms of sample efficiency even though the model-free methods (TRPO and MAML-RL) were trained with 1000 times more data. Additionally, they conducted comparative analysis on a number of continuous control tasks including simulated robots (OpenAI Gym's Ant and HalfCheetah) and real-life robots (millirobot), and the results showed faster adaptation to changing environments.

4.4 GFlowNets for Molecule Generation

Despite not yet being implemented in robotics, GFlowNets has been utilized in drug discovery, where it exhibits tremendous potential. Bengio et al. [21] in their GFlowNets empirical paper, conducted experiments on a large-scale domain of small drug molecule graph generation. The primary objective of this experiment was to train an agent that will be able to sequentially generate molecules with a high reward.

In this experiment, the environment consists of 10^{16} states. Given a state, the agent has to choose from a set of allowed actions ranging from 100 to 2000 actions. The molecules were generated by sequentially attaching predefined building blocks of molecules to the stems. The stem is where the new predefined building blocks are added sequentially, which results in the formation of a junction tree. In total, 72 building blocks are available for the agent to choose from. The blocks were chosen based on the framework developed by Jin et al. [28] over the ZINC database [29]. In a nutshell, the agent has an action space that has the resulting product of selecting which block to choose from the 72 predefined sets of blocks and which stem the chosen block will be attached to. The Reward is based on a proxy that predicts binding affinity with target proteins. The proxy is trained on a dataset of 300k randomly generated molecules.

The results of the experiment were compared in terms of the density of rewards, with Markov Molecular Sampling (MARS) [30] which is a widely used method of multi-objective drug discovery. Compared to MARS, GFlowNets found many more high-reward molecules. The average reward of GFlowNets was also compared with MARS, PPO, and JT-VAE with Bayesian optimization. Despite the fact that PPO, the state-of-the-art RL algorithm, plateaus after a while, GFlowNet's average rewards improve as more molecules are visited. As shown in this experiment, GFlowNets provide greater exploration ability compared to RL, where RL is satisfied with good enough trajectories, and GFlowNets was able to generate much more unique trajectories with high average reward. Although our approach to this research does not directly relate to this work, it provides motivation for it. Since GFlowNets have demonstrated superior performance compared to state-of-the-art RL algorithms in molecule generation, they may have the potential to excel in machine fault adaptation as well.

4.5 CFlowNets for Continous Control Tasks

Li et al.[7] in their paper conducted experiments with their proposed CFlowNets framework on three different simple continuous control tasks which are Point-Robot-Sparse, Reacher-Goal-Sparse, and Swimmer-Sparse. The rewards associated with these three tasks were sparse. To make a comparative analysis of the performance, they further compared the results with state-of-the-art reinforcement learning algorithms such as DDPG [14], TD3 [15], PPO [16], and SAC [17].

In the first task, there are two goals for Point-Robot-Sparse's agent i.e. the robot has two different target coordinates. Initially, the agent starts at (0, 0) and aims for the target coordinates (5, 10), (10, 5) by taking one step at a time. In this environment, episodes are limited to 12 for this experiment. The agent receives the reward only when the terminal state has been reached. Agents are rewarded based on the difference between their current position and the target coordinate. The rewards are high when the distance is less, and the reward is low when the distance is greater. The Reacher-Goal-Sparse task uses a robotic arm with two joints called "Reacher.". Agents move the end effector of the robot to reach randomized targets. Swimmer-Sparse involves suspending the "swimmer" in a two-dimensional pool. To move in the pool, the swimmer applies torque to the rotors and uses fluid friction. In this task, the goal is to move right or left as quickly as possible.

Based on the rewards distribution, average rewards, and the number of valid-distinctive trajectories generated, the results were compared between CFlowNets and RL algorithms. The findings of the first phase demonstrated that CFlowNets were highly effective at fitting the real reward distribution. However, other reinforcement learning algorithms, on the other hand, struggle to fit the actual reward distribution. Another study generated 10000 trajectories and reported the count of unique, valid-distinctive trajectories. The authors gathered valid-distinctive trajectories based on the fact that if two trajectories have high returns, but their MSE is small (defined by a threshold parameter δ_{MSE}), then instead of two trajectories, only one is counted. The results showed that DDPG, PPO, and TD3 have little to no exploration ability since they only generated one valid-distinctive trajectory. SAC demonstrated good exploration behavior at first but then the performance decreased as the training progressed. On the other hand, CFlowNets showed remarkable exploration capabilities, generating thousands of unique valid-distinctive trajectories that far exceeded any other RL algorithm. Furthermore, for Point-Robot-Sparse and Reacher-Goal-Sparse tasks, CFlowNets achieves a better average return in fewer timesteps than all other RL algorithms. However, it did not perform well in the Swimmer-Sparse task. The reason behind such a poor performance, as explained by the authors, was due to the fact that both the Point-Robot-Sparse and Reacher-Goal-Sparse have evenly distributed rewards which made them more inclined to explore. In contrast, Swimmer-Sparse has a steep reward distribution. Therefore, CFlowNets are better suited to exploration-based tasks, according to the authors.

Chapter 5

Experimental Setup

As we embark on Chapter 5, we transition from the theoretical and conceptual groundwork laid in the preceding chapters to the practical aspects of our research. In this chapter, the experimental setup is delineated in great detail, so that the hypotheses and research questions posed earlier can be validated and answered. It is in this pivotal phase that our hypothesis and theory converge with practice, allowing for an empirical exploration and examination of the proposed concepts and models.

In this chapter, a comprehensive description of the methodology applied is provided, including an in-depth description of the robotic environment used in our experiments. Furthermore, we discuss each of the four faults that were introduced in the simulated robotic environment and how these faults are applied to create four custom gym environments. Then we talk about the implementation of CFlowNets and all the RL algorithms that are used for this comparative experimentation including details on how our proposed models are developed and deployed, experimental designs, and evaluation metrics outlined to make sure our experiments are reproducible and justify our experimental results.

5.1 Hardware and Software

Hardware. The experiments are performed on Ubuntu 20.04.6 LTS server running Linux, equipped with NVIDIA RTX A6000 GPUs.

Software. Our hypothesis is tested using a simulated robot called Reacher-v2 adapted from OpenAI gym, simulated using the MuJoCo physics simulator. At the time of experimentation, to simulate the robotic environment, MuJoCo version 2.1.2.14 was used.

Virtual Environment. For our comparative analysis, we have conducted six segments of experimentation with a total of 5 algorithms including CFlowNets. To conduct these experiments, we created custom virtual environments (mujoco_env). All the virtual environments are created using Anaconda 3. The following Table 5.1 lists some of the necessary libraries of the virtual environment (mujoco_env):

Software	**Version**
Python	3.8.17
Gym	0.21.0
Glew	2.10
Glfw3	3.2.1
Stable-baselines3[extra]	1.8.0
Tensorboard	2.13.0
Numpy	1.24.3
Seaborn	0.12.2
Pandas	2.0.3

Table 5.1: Python version and added libraries of virtual environment

5.2 Environment

A detailed description of the robot's environment, its action and observation space, and its environment setup are provided in this section. All the information is gathered from the Gymnasium documentation.

5.2.1 Reacher-v2

The Reacher-v2 robotic environment is one of the most widely used performance benchmarks in the field of applied reinforcement learning. This robotic simulator is simulated using the MuJoCo (Multi-Joint Dynamics and Contact) physics simulator. Mujoco is a physics engine used for research and development in robotics, biomechanics, graphics and animation, machine learning, and other areas. It provides fast and accurate simulation of articulated structures interacting with their environment [31].

Description. The Reacher-v2 environment is simulated in a 2D plane and consists of a two-joint robotic arm with an end effector/fingertip at the end of the arm. The two joints are named joint0 and joint1 in the reacher XML file and these joints are capable of a wide range of motion. Joint0 fixes the first part of the arm (link0) with the point of fixture also known as the root and joint1 connects the second part of the arm (link1) with link0. The robot arm uses actuators/motors to control these joints, which help provide torque so it can maneuver around the 2D plane. The objective of the task is to maneuver the end effector or fingertip in a manner that enables it to reach a specified target location which is defined in the environment. With each episode, the coordinates of the target change to various locations within the environment. In Figure 5.1, a depiction of the Reacher-v2 environment is presented.

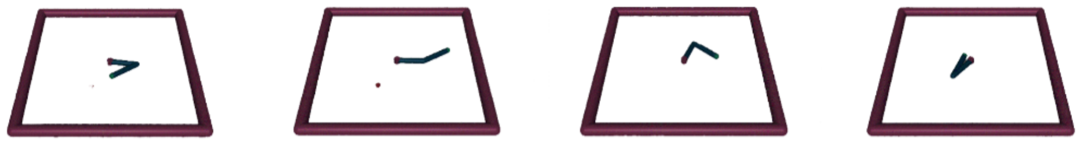


Figure 5.1: Reacher-v2 Environment [31]

Action Space. The action space is a vector of length 2 and the elements have type $float32$ values in the range of [-1.0,1.0]. Action is the application of torques among both hinge joints. Detailed information about each element of action space can be found in the table 5.2.

Table 5.2: Action Space of Reacher-v2 Environment [31]

Num	Action	Control Min	Control Max	Name	Joint	Unit
0	Torque applied at the first hinge (connecting the link to the point of fixture)	-1	1	joint0	hinge	Torque (N m)
1	Torque applied on the second hinge (connecting the two links)	-1	1	joint1	hinge	Torque (N m)

Observation Space. The observation space includes the angles of the joints, the position of the end effector, and the relative position to the target. The observation is a Box(-Inf, Inf, (11,), float64). Observational data includes the following:

- The cosine values corresponding to the angles of both links.
- The sine values associated with the angles of the two links.
- The coordinates where the target is located.
- The rates of change in the angles of the links, referred to as the angular velocities.
- A three-dimensional vector illustrating the spatial relationship between the target and the fingertip of the reacher, with the third dimension having a value of 0.

Details about the observation space have been attached to the table 5.3.

Rewards. Rewards are computed by adding two components:

- reward_distance: This is the first component of the reward which is the measure of distance between the end effector/fingertip of the reacher and the target location. After each episode, the further the distance of the fingertip from the target location, the more negative value is assigned. It is calculated as the negative vector norm of (position of the fingertip - position of target), or -norm("fingertip" - "target").

- reward_control: The second component of the reward is a negative reward which penalizes the reacher whenever the agent takes actions that are too large. It is measured as the negative squared Euclidean norm of the action, i.e., as - sum(action^2).

The total reward is computed as **reward** = reward_distance + reward_control

Episode End. The episodes of the Reacher-v2 environment terminate when it satisfies the following two conditions:

- **Truncation.** By default, each episode of the Reacher-v2 environment consists of 50 timesteps. If the Reacher fails to reach the target location within these 50 timesteps, the current episode terminates and a new episode begins. Otherwise, if the reacher's fingertips reach the target within 50 timesteps, a new target pops up in a random coordinate.

- **Termination.** A finite state space value no longer exists.

Table 5.3: Observation Space of Reacher-v2 Environment [31]

Num	Observation	Min	Max	Name	Joint	Unit
0	cosine of the angle of the first arm	$-\infty$	∞	cos(joint0)	hinge	unitless
1	cosine of the angle of the second arm	$-\infty$	∞	cos(joint1)	hinge	unitless
2	sine of the angle of the first arm	$-\infty$	∞	sin(joint0)	hinge	unitless
3	sine of the angle of the second arm	$-\infty$	∞	sin(joint1)	hinge	unitless
4	x-coordinate of the target	$-\infty$	∞	targetx	slide	position(m)
5	y-coordinate of the target	$-\infty$	∞	targety	slide	position(m)
6	angular velocity of the first arm	$-\infty$	∞	joint0	hinge	angular velocity (rad/s)
7	angular velocity of the second arm	$-\infty$	∞	joint1	hinge	angular velocity (rad/s)
8	x-value of position_fingertip-position_target	$-\infty$	∞	NA	slide	position(m)
9	y-value of position_fingertip-position_target	$-\infty$	∞	NA	slide	position(m)
10	z-value of position_fingertip-position_target	$-\infty$	∞	NA	slide	position(m)

5.3 Faults

The primary objective of our experimentation is to investigate whether or not CFlowNets can adapt when a fault occurs in the environment. To do so, the first step would be the modification process of the XML file for the Reacher-v2 environment. The XML file for the Reacher environment is a configuration file used by the MuJoCo (Multi-Joint Dynamics with Contact) physics engine to create a simulated environment. This file contains the physical parameters and structure of the Reacher robot and its environment, so users can simulate the robot's movement and interaction in a virtual environment.

At the very top of the hierarchy, the Worldbody component of the XML file contains all the physical elements and all the body attributes of the environment which includes the 2D arena, the robotic arm, and the target. The Arena component defines the boundaries/walls for the environment. The name attribute indicates each of the geometry of the arena, such as the ground and the four walls, which are sideS, sideN, sideW, and sideE. The Arm component in the environment represents the structure, bodies, joints, links, and geometry of the robotic arm of reacher-v2. The attributes of this component determine various dynamics of the robot such as the height/width of the links, the joint positions, the range of each joint etc. The last component of the worldbody is the Target which defines the target that the end effector of the robotic arm is supposed to reach. This component contains the sliding joints that enable the target to change its location in X, and Y directions within the arena. There is also another component outside the worldbody which is called the Actuator. The Actuator specifies the motors/actuators that are present in each joint of the robot. These actuators apply torques to the joints which in turn enables the robotic arm to move in the environment. The Actuator component consists of some attributes that determine the control range and output power of the gears of actuators. See Figure 5.2 for the XML configuration for the Reacher-v2 normal environment:

```
<mujoco model="reacher-v2">
  <compiler angle="radian" inertiafromgeom="true"/>
  <default>
    <joint armature="1" damping="1" limited="true"/>
    <geom contype="0" friction="1 0.1 0.1" rgba="0.7 0.7 0 1"/>
  </default>
  <option gravity="0 0 -9.81" integrator="RK4" timestep="0.01"/>
  <worldbody>
    <!-- Arena -->
    <geom conaffinity="0" contype="0" name="ground" pos="0 0 0" rgba="0.9 0.9 0.9 1" size="1 1 10" type="plane"/>
    <geom conaffinity="0" fromto="-.3 -.3 .01 .3 -.3 .01" name="sideS" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
    <geom conaffinity="0" fromto=" .3 -.3 .01 .3  .3 .01" name="sideE" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
    <geom conaffinity="0" fromto="-.3  .3 .01 .3  .3 .01" name="sideN" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
    <geom conaffinity="0" fromto="-.3 -.3 .01 -.3 .3 .01" name="sideW" rgba="0.9 0.4 0.6 1" size=".02" type="capsule"/>
    <!-- Arm -->
         <geom conaffinity="0" contype="0" fromto="0 0 0 0 0 0.02" name="root" rgba="0.9 0.4 0.6 1" size=".011" type="cylinder"/>
    <body name="body0" pos="0 0 .01">
      <geom fromto="0 0 0 0.1 0 0" name="link0" rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>
      <joint axis="0 0 1" limited="false" name="joint0" pos="0 0 0" type="hinge"/>
      <body name="body1" pos="0.1 0 0">
        <joint axis="0 0 1" limited="true" name="joint1" pos="0 0 0" range="-3.0 3.0" type="hinge"/>
        <geom fromto="0 0 0 0.1 0 0" name="link1" rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>
        <body name="fingertip" pos="0.11 0 0">
          <geom contype="0" name="fingertip" pos="0 0 0" rgba="0.0 0.8 0.6 1" size=".01" type="sphere"/>
        </body>
      </body>
    </body>
    <!-- Target -->
    <body name="target" pos=".1 -.1 .01">
       <joint armature="0" axis="1 0 0" damping="0" limited="true" name="target_x" pos="0 0 0" range="-.27 .27" ref=".1" stiffness="0" type="slide"/>
      <joint armature="0" axis="0 1 0" damping="0" limited="true" name="target_y" pos="0 0 0" range="-.27 .27" ref="-.1" stiffness="0" type="slide"/>
      <geom conaffinity="0" contype="0" name="target" pos="0 0 0" rgba="0.9 0.2 0.2 1" size=".009" type="sphere"/>
    </body>
  </worldbody>
  <actuator>
    <motor ctrllimited="true" ctrlrange="-1.0 1.0" gear="200.0" joint="joint0"/>
    <motor ctrllimited="true" ctrlrange="-1.0 1.0" gear="200.0" joint="joint1"/>
  </actuator>
</mujoco>
```

Figure 5.2: Reacher-v2 XML

As part of our efforts to introduce faults into the environment to demonstrate the adaptation performance of each algorithm, we have chosen four faults to focus on. Each of these faults represents and mimics some real-world malfunctions that are encountered by robot arms in a practical environment. We created four different custom gym environments representing these four faults. Each of these four custom gym environments contains a constant fault type. Constant fault type implies that the values of the modified attributes remain constant over time. An attribute search was conducted to determine the modified values of the attributes. This iterative testing approach is similar to a hyperparameter search where we test different settings of attributes and observe the learning trend for a limited number of timesteps. The severity of the faults was chosen in such a way that the fault introduces more complex dynamics in the new environment. However, the new dynamics of the environment were not changed to such an extent that completely derailed the learning process of the algorithms. In other words, the particular attribute values chosen for simulating the four fault environments were intended to significantly impact the Reacher-v2's performance but still allow for meaningful learning and adaptation. We started with initial values that were later modified based on preliminary runs to make sure that the fault produces a substantial amount of challenge for the robotic arm but does not render the task unsolvable. By performing this iterative attribute search and fine-tuning the attribute values we were able to create four custom gym environments to test each of the algorithms' adaptive performance. In the following parts, we will discuss each of these faults and how they were simulated in the environment.

5.3.1 Reduced Range of Motion

Robotic arms can experience a reduced range of motion or joint angular displacement due to a number of factors, including gear wear and tear, mechanical restrictions, or malfunctioning software. This type of fault prevents a robot from performing its intended task efficiently because it is unable to move through the full angular

range that it was designed for. As a result, this fault will impact the precision and flexibility of a robot in maneuvering its joints and will lead to decreased task efficiency. In the context of our environment Reacher-v2, a reduction in its range of motion will lead to poor performance as the arm will not be able to reach its distant targets which may require a wide range of motion within the given limited timestep. To simulate this real-world mechanical fault, we adjusted the $< joint >$ element's *range* attribute within the Reacher-v2's XML file. The *range* value of $joint1$ in the normal environment was "-3.0 3.0" radians, but we changed it to "-1.0 1.0" radians, which means the joint won't be able to rotate to its full extents.
Original configuration:

```
<joint ... range="-3.0 3.0" ... />
```

Modified configuration:

```
<joint ... range="-1.0 1.0" ... />
```

5.3.2 Increased Damping

The term damping refers to the decrease in oscillation amplitude. When there is an increase in damping in the robot's joint it indicates that there is an abnormal resistive force which is impairing the oscillation of the robot's links. In a practical environment, this type of malfunction may be caused by outside contaminants affecting the joints or a mechanical problem that increases friction within the joints. To simulate the effects of damping, we increase the attribute value of *damping* in the $< joint >$ element from "1" to "5" for joint1 in the XML configuration of Reacher-v2.
Original configuration:

```
<joint ... damping="1" ... />
```

Modified configuration:

```
    <joint ... damping="5" ... />
```

5.3.3 Actuator Damage

Robots have actuators or motors within their joints that provide torque for driving and maneuvering the joints. Actuator damage can be caused by incorrect software inputs to the actuators, gear damage, overheating or electrical faults, etc. Due to this type of fault, the reacher robotic arm may experience incorrect movements, reduced output force at the joints, and an inability to maintain position. Consequently, the Reacher's accuracy, speed, and strength are adversely affected, reducing its effectiveness and performance. We simulated the actuator damage fault by modifying the *gear* attribute of the $< motor >$ element within the XML file. The *gear* value was halved from 200.0 to 100.0. This reduction in the gear ratio resulted in a decrease in actuator power, mimicking a weakened motor.
Original configuration:

```
    <motor ... gear="200.0" ... />
```

Modified configuration:

```
    <motor ... gear="100.0" ... />
```

5.3.4 Structural Damage

One of the most common types of fault in real-world application of robotics is the structural damage of robotic manipulators. Structural damage can occur for various reasons such as external impacts, corrosive elements, manufacturing defects, or simply because of deterioration of the materials.

To analyze the Reacher-v2 environment performance onset of a structural damage fault, we decided to bend link1 of the robotic arm. However, in the XML configuration, there are no specific attributes that can be directly modified to bend the body link of the arm. In the XML configuration file, the start and end positional coordinates of geometric shapes in the 3D space are defined by a $< geom >$ tag. The $fromto$ attribute has six numbered coordinates where the first three numbers specify the x, y, z coordinates of the starting position and the last three numbers specify the x, y, z coordinates of the ending position of the geometric shape. In the original settings of the Reacher-v2 environment, the link1 element's $fromto$ attribute in the $< geom >$ tag was "0 0 0 0.1 0 0" which means the link starts at the origin (0, 0, 0) (here origin is relative to the parent $body1$ and not the global origin) and extends to (0.1, 0, 0) in the x direction. This represents a straight arm along the x-direction. To simulate the bend of $link1$, we first split the link into two equal segments and named them "link1a" and "link1b". "link1a" starts from "0 0 0" to "0.05 0 0" which is half the original link1 length in the positive x direction. "link1b" starts where link1a ends which is "0.05 0 0". To make a 45-degree bend by maintaining the length of link1b equal to link1a, we determined the end coordinates of link1b by using sine and cosine functions. It is worth mentioning that this manual manipulation will not create a new joint between "link1a" and "link1b", rather it will change the geometric dynamic of the arm in such a way so that it creates a bend. In other words, in the context of MuJoCo, if two rigid bodies are connected without a joint, they essentially become a single rigid body.

$$x = 0.05 + 0.05 \times \cos(45^\circ)$$
$$y = 0.05 \times \sin(45^\circ)$$

Using this trigonometric principle, we determined that "link1b" starts in "0.05 0 0" and extends to "0.08535, 0.03535, 0" which creates a $45-degree$ angle between the two links. Due to their orientation, it mimic a bend in link1. A visual representation has been attached in Figure 5.3 and 5.4.

Original configuration:

```
    <body name="body1" pos="0.1 0 0">
        <joint axis="0 0 1" limited="true" name="joint1" pos="0 0 0"
        range="-3.0 3.0" type="hinge"/>

        <geom fromto="0 0 0 0.1 0 0" name="link1" rgba="0.0 0.4 0.6 1"
        size=".01" type="capsule"/>

        <body name="fingertip" pos="0.05 0.05 0">
            <geom contype="0" name="fingertip" pos="0 0 0"
            rgba="0.0 0.8 0.6 1" size=".01" type="sphere"/>
        </body>
    </body>
```

Modified configuration:

```
    <body name="body1" pos="0.1 0 0">
        <joint axis="0 0 1" limited="true" name="joint1" pos="0 0 0"
        range="-3.0 3.0" type="hinge"/>

        <!-- First part of the original link1 (link1a)-->
        <geom fromto="0 0 0 0.05 0 0" name="link1a"
        rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>

        <!-- Second part connected at a 45-degree angle (link1b) -->
        <geom fromto="0.05 0 0 0.08535 0.03535 0" name="link1b"
        rgba="0.0 0.4 0.6 1" size=".01" type="capsule"/>
        <body name="fingertip" pos="0.05 0.05 0">
            <geom contype="0" name="fingertip" pos="0 0 0"
            rgba="0.0 0.8 0.6 1" size=".01" type="sphere"/>
        </body>
    </body>
```

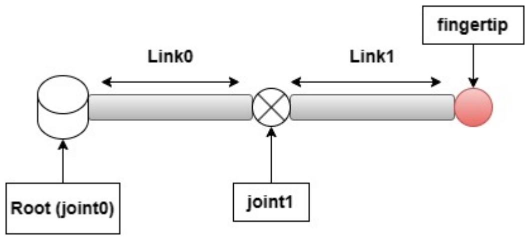


Figure 5.3: Original Robotic Arm

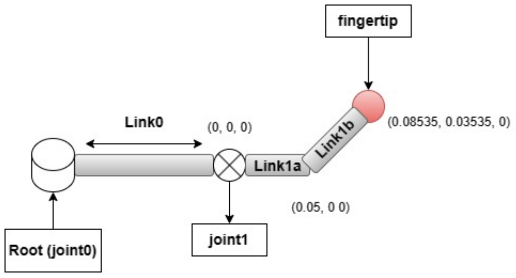


Figure 5.4: Bent Robotic Arm

5.4 Experimental Settings

Algorithms. For our comparative study, we have conducted six segments of experiments. Our results from each of these experiments helped us visualize the performance of CFlowNets and four different reinforcement learning algorithms on the same task (Reacher-v2). The implementation of the Continuous Flow Networks (CFlowNets) algorithm was adopted from the recently published paper [7] where the authors extended the theoretical formulation and flow-matching theorem of the GFlowNets foundation to make it appropriate for continuous scenarios. To evaluate the performance, adaptation efficiency, and adaptation speed of CFlowNets, we decided to compare it with four reinforcement learning algorithms which are: DDPG, TD3, PPO, and SAC. To implement these four algorithms in our Reacher-v2 task, we used the Stable Baslines3 library which contains the implementation of various popular reinforcement learning algorithms.

5.4.1 CFlowNets

In accordance with the theoretical formulation of CFlowNets discussed in Chapter 4, a retrieval network is constructed in order to make predictions about the parent nodes of each state in a sampled trajectory, which is elaborated in the equation 3.19. Since the state transition rules vary across different environments, we must pre-train the Retrieval network tailored to the Reacher-v2 environment. A custom RewardShapeWrapper class is utilized to modify the structure of the reward of the base environment to a sparse form. As a result, non-terminal steps are rewarded with zero, and terminal steps are rewarded with a reward derived from the distance to the target. The retrieval network is made of a feed-forward neural network. This NN consists of four layers. The input layer concatenates the state and action vectors in the first input layer. In each hidden layer, we have 256 neurons with ReLU activation functions applied. Tuples of experience are stored and managed in this experience buffer.

During training, at first, the agent randomly interacts with the environment to gather experiences and push them into the replay buffer. This approach enables the agent to randomly explore the environment to get an initial estimate. After a certain threshold of 500 timesteps, the agent stops this exploration and the policy update begins. The replay buffer facilitates efficient sampling of mini-batches, ensuring that the network is trained on a diverse range of experiences that are crucial for enhancing its generalization capability. The network training is done by sampling mini-batches from the replay_buffer and feeding the next state along with the corresponding action to the retrieval network. The retrieval network outputs the parent state for the given state-action pair. Then we iteratively update the model weights to minimize the MSE loss between the predicted parent state (predicted by the retrieval network) and actual subsequent states (collected from the mini-batch). Adam optimizer with a learning rate of 0.0003 is used for network optimization.

Subsequently, the Retrieval network is employed to enhance the precision of locating parent states during flow matching in the implementation that involves the training of the flow network. To stabilize the training of the flow network, agents executed actions are normalized and clipped to be within the valid action space range. The architecture of the flow network comprises a feedforward neural network with three layers. The concatenation vector of state and action is fed from the input layer through two hidden layers. There are 256 neurons in each hidden layer with the ReLU activation function applied. As the final layer, the output layer is composed of a single neuron that outputs the calculated edge flow. Using Softplus activation functions in the output layer ensures that edge flow values are non-negative since flows cannot be negative.

At first, the agent iteratively samples actions from the action probability buffer. This buffer is created by uniformly sampling a huge number of actions (10,000 actions in our implementation) which is defined by the size of the action replay buffer. These sampled actions along with the current state are then fed to the flow network which outputs a non-negative flow value for each sampled action. The agent selects an action from this buffer according to their flow value (high flow value = high probability of getting sampled). This sampling approach ensures the exploration and evaluation of a wide range of possible actions for a given state to determine which action the agent should take to produce high-return solutions. During training a batch of experiences are gathered from the replay buffer. At each state, a number of actions are randomly sampled (defined by the hyperparameter sample_flow_num). For the inflows, our implementation included using the retrieval network to predict the parent state, and then the predicted parent state-action pair is fed to the flow network. The outflows are calculated by using the same flow network which takes the randomly sampled actions along with the state as input and outputs the outflow. Adam optimizer with a learning rate of 0.0003 is used for network optimization.

The introduction of faults in the experiments is done by replacing the source path of the normal XML configuration with the modified XML configuration in the reacher.py file. The evaluation period of our experiments is 5000 timesteps which means that at this regular interval, we freeze the learned policy and evaluate it for 10 episodes. The rewards received in this evaluation period are averaged and recorded for further analysis.

The implementation involves several configurable hyperparameters, including action probability buffer size, number of sample flows, policy noise, noise clip, etc. We conducted a hyperparameter search for the CFlowNets algorithm to figure out the best combination of hyperparameters that leads to optimal performance. The search began with the published CFlowNets hyperparameter settings. Then we compared different hyperparameter settings by recording the average return in the last 200 evaluation for a single run. This process was conducted for a total of 10 runs to account for the performance variability. The rewards from these 200 evaluation points across 10 runs were then averaged again to get a single performance metric for each set of hyperparameter settings. This final average return is then used to compare the performance under different set of hyperparameters. A higher average return signifies a better hyperparameter setting. Tensorboard was also used for real-time training monitoring, which aided in fine-tuning hyperparameters. It is worth mentioning that all the hyperparameter tuning was done for the Reacher-v2 normal environment and not for the four modified fault environments. This decision was made based on the assumption that in practical scenarios we are optimizing the model only for the normal environment and not for the fault environments. The list of selected hyperparameters is shown in the Table 5.4.

One of our experiments involved investigating whether there is a benefit in terms of CFlowNet's performance if we save and transfer the learned knowledge from a normal environment to a faulty environment. In order to utilize this concept of transfer learning, we saved the model parameters/weights that the flow network has learned in the normal Reacher-v2 environment. Both the flow network and retrieval

Table 5.4: Best Performing Hyperparameters in the CFlowNets training.

CFlowNets	Reacher-v2
Total Timesteps	10,000,000
Max Episode Length	50
Eval Frequency	5,000
Learning Rate	0.003
Batchsize	256
Retrieval Network Hidden Layers	[256, 256, 256]
Flow Network Hidden Layers	[256, 256]
Number of Sample Flows	100
Action Probability Buffer Size	10,000
Replay Buffer Size	100,000
ϵ	1.0
Optimizer	Adam

network models were saved using PyTorch's torch.save() function, which stores the state dictionary of the model in a file. Using the state dictionary, it maps each layer of the model to its trainable parameters (weights and biases) which were later deployed using PyTorch's load_state_dict() function while experimenting with the four custom gym environments. Additionally, we have also transferred the replay buffer contents that were saved while training in the base normal environment to the fault environments which can potentially accelerate learning by leveraging prior experiences. The replay buffer contents which include the state, action, reward, and next states of each experience, are saved using the Python "Pickle" module which represents the agent's interaction in the pre-fault environment. The replay buffer is then deployed in the fault environment as a starting point. At this stage, the replay buffer contains experiences from the original pre-fault environment and also new experiences that the agent gathers in the new environment. This method of retaining both model and replay buffer was done to investigate if there are any benefits to transfer learning in CFlowNets models to mitigate cold start and make the learning process more stable in the initial episodes.

5.4.2 RL Algorithms (DDPG, TD3, PPO & SAC)

One of the key factors of our research was not only to investigate CFlowNets' performance in robotics and adaptation but also to make a comparison with current state-of-the-art reinforcement learning algorithms on the same task. To make this comparative analysis, we implemented four reinforcement learning algorithms with minor modifications to evaluate their performance in the Reacher-v2 environment. Unlike CFlowNets, for the reinforcement learning algorithms, we did not conduct an extensive hyperparameter search. We mostly used published hyperparameter settings for the reacher-v2 task with some minor tuning. Our focus was more on tuning the selective hyperparameters that promote exploration for these four RL algorithms such as policy noise, noise clip, target policy smoothing, gaussian exploration noise, temperature for entropy, etc. Additionally, hyperparameters that indirectly affect exploration (clipping_range_vf, clipping parameter for PPO) were also considered for tuning.

The implementation of both DDPG and TD3 in this study was derived from the introductory paper of these algorithms and the official codebase shared by Fujimoto et al. [15] in their repository. The TD3 algorithm follows an actor-critic architecture, and for our implementation, it is comprised of three fully connected layers. The first two hidden layers contain 256 neurons each and the ReLU activation function is applied for model non-linearity. The output layer uses the hyperbolic tangent activation function (tanh) which ensures that the output values are normalized and scaled to match the environment's action space. The critic network employs two critic networks Q1 and Q2 to stabilize function approximation by underestimation of Q-values. In order to estimate the Q-values, the action and state are used as inputs. Both Q1 and Q2 networks are made up of three fully connected layers. The hidden layers have 256 neurons each. To prevent the policy from overfitting to the noise in value estimation, and to ensure a more stable learning procedure, the actor updates are delayed. The hyperparameter "policy_freq" is set to 2 in the implementation which enables the actor-network policy to be updated after every two iterations. The table 5.5 reports the remaining hyperparameter settings.

DDPG's actor-network consists of three fully connected layers as well, but the first hidden dimension has 400 neurons and the second hidden dimension has 300 neurons. Both hidden layers utilize the ReLU activation function. The output layer uses a tanh activation function which works similarly to TD3's actor-network output layer. In order to fit the environment's action space constraints, the outputs are scaled by a max_action factor. As DDPG is the predecessor of TD3, instead of using two critic networks for Q value estimation, it uses one fully connected three-layered network. The first layer takes state dimensions as input. It has 400 neurons with ReLU activation function. The second hidden layer takes the output of the first layer and then concatenates it with the action input. This layer is made up of 300 neurons, followed by a ReLU activation function. Finally, the single neuron output layer produces the Q-value estimation. For our implementation, we used the published hyperparameters for DDPG and TD3 in [14] and [15] respectively, with minor modifications to the coefficient for soft update (τ) and policy noise. The hyperparameters are presented in the table 5.5.

Our implementation of Soft-actor-critic (SAC) was adopted from the [32] repo which contains the PyTorch implementation of the algorithm's introductory paper [17]. The policy and value network of this implementation utilized MLP architecture for function approximation. The actor's network is a three-layered network with 256 neurons in each of the hidden layers. The hidden layers of the actor-network get the state observations and it maps to action distributions. The output layer has two segments. One segment represents the mean, and the other represents the log standard deviation of the action distributions which are then used to parameterize a

Table 5.5: Hyperparameters used in the TD3 and DDPG training for the Reacher-v2 task.

Hyperparameters	DDPG	TD3
Total Timesteps	10,000,000	10,000,000
Max Episode Length	50	50
Eval Frequency	5,000	5,000
Learning Rate	0.003	0.003
Batchsize	256	128
Policy Network Hidden Layers	[400, 300]	[256, 256]
Value Network Hidden Layers	[400, 300]	[256, 256]
Discount Factor (γ)	0.99	0.99
Tau (τ)	0.005	0.005
Gaussian Exploration Noise		0.1
Policy Noise	0.2	0.2
Noise Clip	0.5	0.5
Replay Buffer Size	100,000	100,000
Optimizer	Adam	Adam

squashed Gaussian distribution for action selection. Similar to TD3 and DDPG, the double Q learning of the critic network is also a three-layered network that outputs a scalar Q-value estimate. The learning rate of both actor and critic networks is 0.0001. For the entropy term, the initial temperature parameter α is set to 0.1, and learnable temperature is enabled, allowing the model to optimize temperature during training. The set of parameters used for our experiments with the SAC algorithm on Reacher-v2 is outlined in the following Table 5.6.

Table 5.6: Hyperparameters used in the SAC training for the Reacher-v2 task.

Hyperparameters	SAC
Total Timesteps	10,000,000
Max Episode Length	50
Eval Frequency	5,000
Learning Rate	0.001
Batchsize	1024
Policy Network Hidden Layers	[256, 256]
Value Network Hidden Layers	[256, 256]
Discount Factor (γ)	0.99
Temperature (α)	0.1
Replay Buffer Size	100,000
Optimizer	Adam

For the sake of reproducibility and integrity in our comparative experiments, we used the Stable Baselines3 implementation for the target algorithm PPO with some added minor modifications. In the stable baselines3 version, no explicit evaluation frequency parameter was included. We added this feature in our implementation to allow us to stop the training after a specified number of iterations, evaluate the current policy for a specified number of steps, and record the performance of the current policy. The stable baselines3 implementation of PPO comes with multiple types of policy classes (ActorCriticPolicy, ActorCriticCnnPolicy, MultiInputActorCriticPolicy) for various types of observation spaces. Since we are dealing with continuous state-action spaces with low-dimensional vectors (position, velocities, etc.), for our implementation we utilized the ActorCriticPolicy class for our task. Based on the original PPO paper [16], the code implements the clipped surrogate objective. The "clip_range" parameter controls the extent to which the policy can be updated in a single step, providing a form of regularization and ensuring stability in training. The tuning of the "clip_range" parameter enabled restricting large policy updates and as a result, improved the overall training stability. We also decided to include the "clip_-range_vf" parameter which enables the clipping mechanism to the value function and further stabilizes the training by restricting large value function updates. For PPO, we added the feature of linear learning decay options and used the generalized advantage estimator (GAE) [33], resulting in more stable and effective policy updates by balancing bias and variance in the advantage estimates. The hyperparameters are presented in the Table 5.7.

Table 5.7: Hyperparameters used in the PPO training for the Reacher-v2 task.

Hyperparameters	PPO
Total Timesteps	10,000,000
Max Episode Length	50
Eval Frequency	5,000
Learning Rate	0.003
Batchsize	64
Policy Network Hidden Layers	[64, 64]
Value Network Hidden Layers	[64, 64]
Discount Factor (γ)	0.99
GAE lambda (λ)	0.95
clipping parameter	0.2
clip_range_vf	0.2
Optimizer	Adam

Timesteps and policy evaluation. All of the algorithms for this comparative experiment were run for 10 million timesteps to determine where each of them converges to their asymptotic performance. There were 50 timesteps in each episode of the task. The policy evaluation was performed for all the algorithms after an interval of 5,000 timesteps.

Chapter 6

Results and Discussion

In this chapter, we present our experimental results for our comparative analysis of CFlowNets and RL algorithms in the field of robotics and fault adaptation. Furthermore, our results are discussed and interpreted based on each experiment in detail. This chapter aims to bridge a gap between our theoretical hypothesis and empirical test and also to address the research questions that were posed in Chapter 1. In order to simplify the understanding, we segment our experiments into six distinct parts, each focussing on different aspects of our comparative analysis. Throughout this chapter, each section is devoted to a unique aspect of the experiments, providing a detailed analysis of the findings in relation to the hypotheses and objectives outlined earlier. All the experiments were run for 10 million timesteps, but for the sake of clarity and readability of performance during the early stages of learning, the plots are truncated to keep the first 1 million timesteps.

6.1 Initial Insights: Testing CFlowNets' Viability in Robotic Simulation Environment

For the first segment of our experiments, we have implemented CFlowNets and the four RL algorithms in the Reacher-v2 task simulated by the MuJoCo physics simulator. The purpose of this experiment is to answer our first research question, which was whether or not it is possible to implement CFlowNets in a robotic environment.

A recently published paper [7] about CFlowNets's introductory paper already addressed that question by applying CFlowNets to robotic tasks with sparse rewards. We re-constructed the retrieval network based on the theoretical formulation of the paper and also made some changes to the original flow network, such as modifying the hardcoded hyperparameter settings, making a configuration file to enhance hyperparameter tuning, and reducing complexity wherever possible. Afterward, we implemented the CFlowNets algorithm on Reacher-v2 and compared the average return against RL algorithms shown in Figure 6.1. The experiments were run ten times, and the shaded regions represent the performance variability across these ten runs. The dashed lines indicate the asymptotic performance to signify the point of convergence for each algorithm where the performance improvement or degradation becomes marginal.

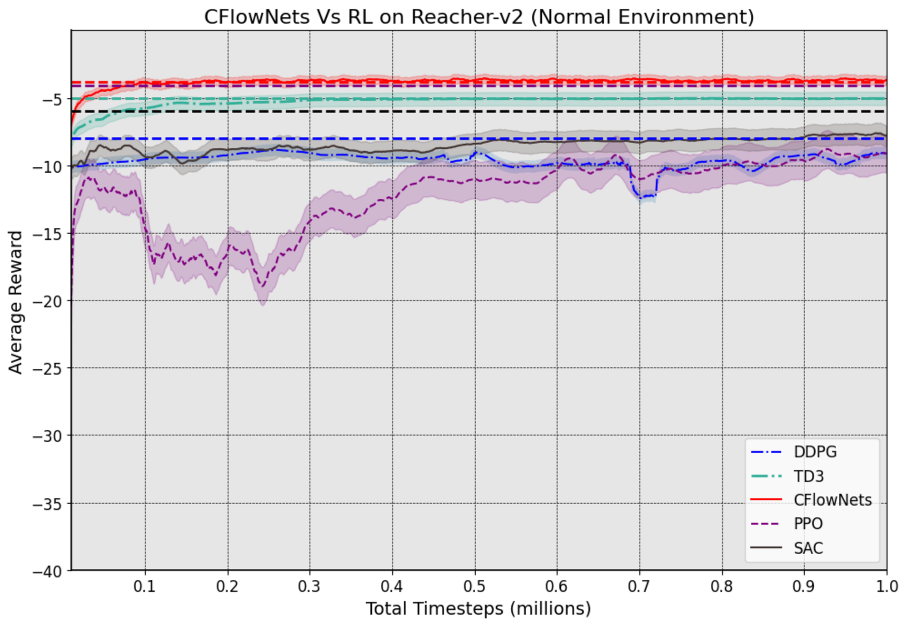


Figure 6.1: Average Return in Normal Reacher-v2 Environment. The shaded areas correspond to a 95% confidence interval. The dashed line represents the asymptotic performance.

Figure 6.1 illustrates the comparative performance of five algorithms: CFlowNets, TD3, DDPG, SAC, and PPO on the Reacher-v2 task. The empirical data collected in a span of 10 million timesteps for the Reacher-v2 task and a quick analysis of the graphical data reveal that CFlowNets and PPO stand out as the top-performing models.

Compared to other algorithms, CFlowNets exhibited a rapid rise in average reward accumulation over time. From the earlier training stages of learning, CFlowNets shows a commendable consistency in growth, eventually achieving a top-tier reward average. The quick improvement of CFlowNets is closely tied to its strong ability to explore and learn effectively with fewer samples. This feature becomes apparent when contrasted with other RL algorithms like PPO, which faces challenges during the initial learning phases. The smaller shaded areas on the CFlowNets, PPO and TD3 graphs show that these algorithms have more stable performance over ten runs. The learning curve of CFlowNets stabilizes around an average reward of 2.75, making it the top-tier performer for the Reacher-v2 task in the normal environment because a high point of convergence suggests that CFlowNets is able to find a more high-return solution through exploration. The comparison of CFlowNets with the other RL algorithms will be discussed in more detail in the subsequent segments of experiments, where we evaluate its performance with faulty Reacher-v2 environments. This initial first segment of our experiments was just to determine if CFlowNets could function in a robotic setting, rather than being a comprehensive performance comparison with all the RL algorithms.

6.2 Initial Insights: Execution time of 1 Million Timesteps

Figure 6.2 demonstrates a key factor of algorithmic efficiency which is the execution time. All the algorithms were run for 1 million timesteps and the time it takes for each algorithm to finish executing in Reacher-v2 was reported. This is also part of our preliminary experiment to get initial insights into the performance of CFlowNets. It helps to establish an initial benchmark and is not indicative of the optimal operation for each algorithm. It should be noted that this comparison might not be considered a fair one since some algorithm reaches convergence significantly earlier before executing 1 million timesteps. To provide a foundational understanding of CFlowNets, we chose to collect 1 million timesteps as a starting point to gather initial data and performance indicators. These will be refined and nuanced in subsequent, more tailored experiments. Later in the study, we intend to conduct a more granulated analysis that takes into account the distinct time of each algorithm when they reach near-asymptotic performance which will ensure a more fair comparison.

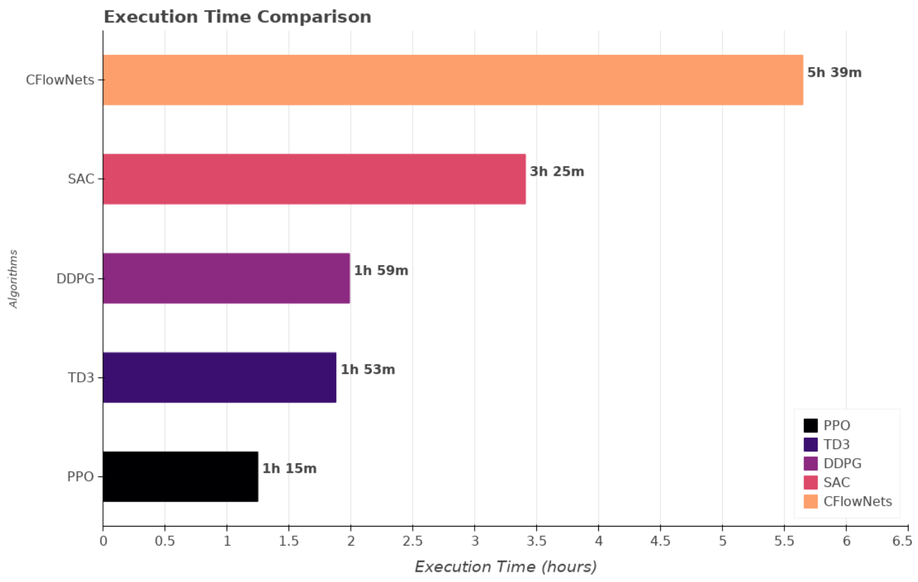


Figure 6.2: Execution Time in Normal Reacher-v2 Environment (1 million timestep).

From Figure 6.2, we can see that CFlowNets takes the longest to execute 1 million timesteps of the Reacher-v2 environment, with an execution time of 5 hours and 39 minutes. On the contrary, other RL methods necessitate approximately between 1 to 3.5 hours which is much less compared to CFlowNets. This longer execution time can be due to the fact that CFlowNets generate a distribution over all possible paths, and samples from the most rewarding paths with a higher probability. Although this feature enables CFlowNets to showcase good performance in terms of accumulating rewards, this comes at a computational cost. The continuous normalizing flows, which empower CFlowNets in modeling complex dynamics, may inherently be more computationally intensive. It is also worth noting that the reported execution time only accounts for the training of flow networks which already have a pre-trained Retrieval network. If we take into account the retrieval network execution time (which is tailored to each specific environment), then that will result in an execution time that will be much higher than the other RL algorithms.

6.3 Adaptation Performance

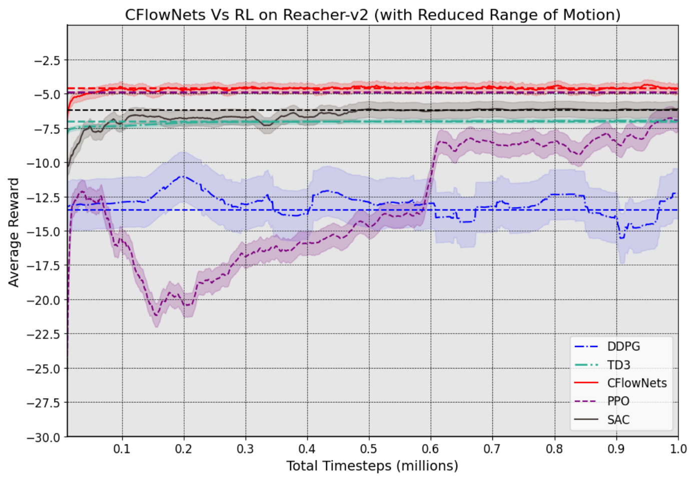
In the third segment of our comparative analysis, we evaluate the adaptation performance and sample efficiency of CFlowNets compared to RL algorithms in four distinct fault environments. To simplify the explanation, we have categorized the faults into pairs: reduced range of motion and increased damping are labeled as Fault 1 and Fault 2 (Motion Impairment Faults), respectively, while actuator damage and structural damage are designated as Fault 3 and Fault 4 (Structural and Mechanical Faults). For each of these four experiments, initially the agent randomly explores the environment for the first 500 timesteps to gather experience in the buffer. After this initial random exploration, at 500 timesteps the training begins and the policy is updated. The first policy evaluation is done during this period to set up an initial performance benchmark for each of the algorithms which reflects each of the policy's exploration capabilities after having some random interaction with the environment at the start of the training phase. After that, the policy was evaluated at the regular 5000 timestep. All the algorithms start with the same performance at timestep = 0 however, as the first evaluation point (at 500 timesteps) is so close to the origin in a large-scale plot of 1 million timesteps, visually it may seem like it's starting from non-zero. Additionally, each of the algorithms begins with a random initial policy which leads to variations in initial rewards at the first evaluation point.

Motion Impairment Faults. Figures 6.3 depicts the average return with respect to the number of real experiences collected (i.e., timesteps) for the environment with fault 1 and fault 2 (reduced range of motion and increased damping respectively).

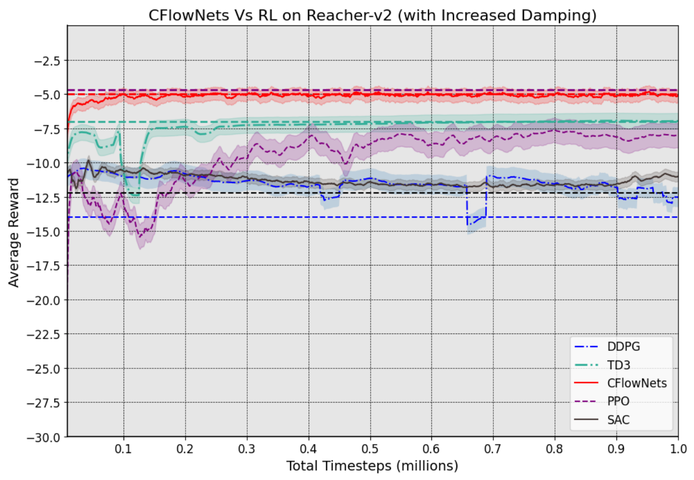
Both of these faults impact the precision and flexibility of a robot in maneuvering its joints and lead to decreased task efficiency. By analyzing the graph, we can see that CFlowNets maintains a relatively consistent performance through the experiment for both the fault environment. The narrow shaded area corroborates the stability and minimal fluctuation of the algorithm over the ten trials. Additionally, it surpassed all the other RL algorithms in terms of average reward, accumulating the highest average reward compared to all the RL algorithms (TD3, DDPG, PPO, and SAC) for fault 1, which indicates CFlowNets architecture has the inherent capability to understand and adapt in adversarial conditions while dealing with constrained action space. For fault 2, Figure 6.7b indicates that while CFlowNets did not top the charts, its performance remained robust, closely mirroring the PPO's asymptotic performance. Based on sample efficiency, CFlowNets provided excellent results because, for both fault environments, CFlowNets required the least amount of real experience to gain a near-asymptotic performance.

In both the custom environment, PPO encounters an initial dip in performance and undergoes a steeper learning curve. However, as the timesteps progressed, the PPO algorithm was able to stabilize its learning curve after collecting a considerable number of real experiences. It maintained a high average reward at the end of the learning period of 10 million timesteps, gained convergence closer to CFlowNets's learning curve for fault 1, and outperformed every other algorithm for fault 2. Across the 10 experimental runs, PPO has a wider shaded region, indicating a variance in its performance.

TD3 and SAC exhibited similar performance in the environment with the reduced range of motion. Although SAC outperformed TD3 in terms of average return with a higher learning curve, TD3 displayed better sample efficiency by achieving asymptotic performance with fewer than half the number of experiences compared to SAC. However, the narrative changed when both TD3 and SAC algorithm was run in the custom gym environment with increased damping (fault 2). TD3 outperformed SAC



(a) Reacher-v2: Reduced Range of Motion (Fault 1)



(b) Reacher-v2: Increased Damping (Fault 2)

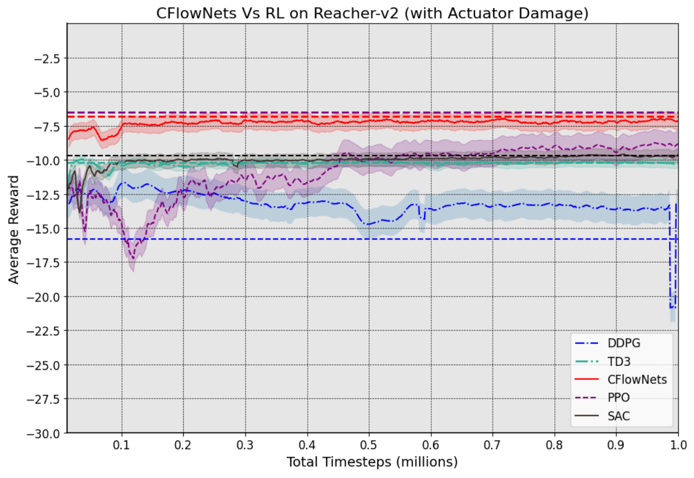
Figure 6.3: The early performance in Motion Impairment Fault environments is depicted through learning curves for all five algorithms. The dashed line represents the asymptotic performance,

both in terms of a higher learning curve and better sample efficiency. Although TD3 faced some performance setbacks in its initial learning phase, it quickly recovered and stabilized within 0.5 million timesteps. Conversely, SAC experienced a decline in its learning curve and necessitated a significantly greater number of timesteps to stabilize and converge.

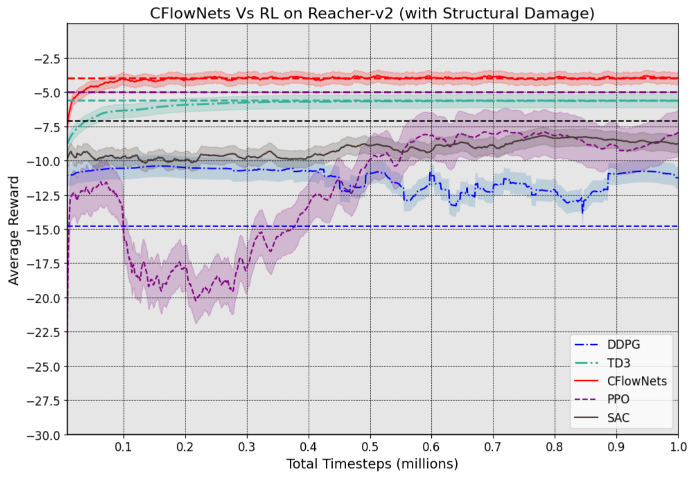
DDPG, the predecessor of TD3, exhibited the poorest performance out of all the algorithms depicted in Figures 6.7a and 6.7b. DDPG showed volatile and fluctuating performance since the beginning of its learning phase and showed numerous dips in its performance throughout the 10 million timesteps. This downward and unstable trend suggests that DDPG found it challenging to modify its policy to overcome the constraints of reduced range of motion and increased damping, indicating its unsuitability for adaptive tasks in these particular environments.

Structural and Mechanical Faults. The performance of the CFlowNets and RL algorithms are showcased in Figure 6.4, in which the algorithms are tested in environments with mechanical (actuator damage) and structural (bend damage) faults. Compared to other fault environments, CFlowNets's performance for fault 3 environment was relatively unstable. During the initial timesteps, noticeable yet minor fluctuations marked the learning curve before it plateaued post 1.2 million timesteps. Unlike other fault environments where CFlowNets's high sample efficiency was able to achieve near-asymptotic performance within a few hundred thousand timesteps, the environment with actuator damage is the only environment where CFlowNets's learned policy struggled to adapt quickly and required more than a million real experiences before it could attain convergence. Nevertheless, the overall performance of CFlowNets was still one of the top only falling behind PPO. CFlowNets also showed minimal performance deviations in its ten runs. In the context of fault 4 environment with structural damage, CFlowNets surpassed its RL counterparts both in terms of upward accumulated average reward trajectory and sample efficiency captured in Figure 6.4b. It required the least amount of experience to reach near-asymptotic performance, and compared to RL algorithms, CFlowNets showed a relatively consistent and stable learning trend.

PPO, on the other hand, was susceptible to a major performance dip initially and was subject to a fluctuating learning trajectory throughout the predominant phase of its learning cycle in both fault 3 and 4 settings. Despite this, PPO showed remarkable



(a) Reacher-v2: Actuator Damage (Fault 3)



(b) Reacher-v2: Structural Damage (Fault 4)

Figure 6.4: The early performance in Structural and Mechanical Faults fault environments is depicted through learning curves for all five algorithms. The dashed line represents the asymptotic performance,

adaptability to both adversarial conditions by outperforming every algorithm in the comparative analysis for the fault 3 environment and placing second in the fault 4 custom environment. The broad shaded area in the graphical representation indicates a higher degree of variance in PPO's performance relative to its counterparts in these specific tasks.

TD3 and SAC demonstrated almost similar adaptive performance, both exhibiting initial fluctuation at the early stages of their learning period for the custom fault 3 environment. Both algorithms quickly stabilized, showing almost identical reward trends. However, when benchmarked against PPO and CFlowNets, the asymptotic performance of both TD3 and SAC was observed to be inferior. In a subsequent analysis conducted within an environment with structural damage, TD3 displayed an improved performance, peaking around an average reward of -5.6, and exhibited consistent performance with a significantly reduced sample experience. Conversely, the SAC's performance trajectory demonstrated quite the fluctuation, indicating a lack of stability in the learning process. In the end, convergence was achieved, but it required a lengthy learning process.

As observed in fault 1 and fault 2, DDPG exhibited suboptimal adaptive performance both in terms of average reward trend and algorithmic stability. From Figures 6.4a and b, we can see that the initial learning curve was relatively stable, but as the timesteps advanced, a stark deviation was noted. For the entirety of the learning period, DDPG experienced multiple abrupt declines in performance. The algorithm's policy proved to be inadequately robust to recover from it, resulting in a poor average reward gain.

Discussion. In our third segment of comparative analysis, overall, both CFlowNets and PPO emerge as commendable top-tier performing algorithms for our fault adaption task. Out of the four custom gym environments with four distinct faults, CFlowNets was able to surpass its reinforcement learning counterparts in two environments (fault 1 and fault 4), which justifies its robust design for diverse exploration and adaptation capability. On the other hand, for fault 2 and fault 3, PPO outperformed CFlowNets in terms of adaptation performance over 10 million timesteps.

To gain a more comprehensive understanding of the performance difference between these top two algorithms we decided to conduct a statistical analysis for the four fault environments. In order to achieve this, we used the average reward data across the ten independent runs and calculated the mean of the average reward values at regular intervals of every 5000 timesteps for both algorithms. We conducted our statistical analysis by performing independent t-tests at each of the 200 intervals over the course of 1 million timesteps for both CFlowNets and PPO. We chose to conduct this analysis for the first million timesteps because during this early training phase, the learning strategies and efficiencies of these two algorithms become more apparent whereas later the algorithms converge to approximately the same level of asymptotic performance. Additionally, since our goal is a fast adaptation we are more invested in the first initial learning steps of the fault environments. This statistical approach helps to compare the means of two independent groups. The p-values were adjusted by using the Bonferroni correction method for multiple comparisons to substantially reduce the chances of receiving Type I errors (false positives). This method involves multiplying each calculated p-values by the number of tests to compute the adjusted p-values. For the first 1 million timesteps, all the adjusted p-values were below the traditional alpha threshold level of 0.05 which implies that the performance difference between CFlowNets and PPO is statistically significant. By analyzing the calculated p-values a common trend is observed across all four fault environments; a lower adjusted p-value (<0.0039) is reported for the initial timesteps (typically 100k - 500k) of the experiments. These lower p-values correspond to a high statistically significant performance disparity in average reward at the early phase of the training between CFlowNets and PPO. Conversely, as the timestep progressed (usually after 500k timesteps), the least significant differences were recorded (higher p-values > 0.008) which indicates that the performance gap between these two algorithms became less pronounced. The t-test statistical analysis highlights key periods during training where the sample efficiency of CFlowNets allowed a superior accumulation of average rewards compared to PPO and helped us to effectively contrast these two algorithms' initial strengths and weaknesses at the early training phase.

Performance retention is another criterion we decided to include to further compare CFlowNets and PPO. In our context of adaptation, we are trying to determine which algorithm has the intrinsic ability to adjust its policy in response to adversarial conditions in the environment and still maintain high performance rather than outputting an equivalent level of performance as in the normal environment. For instance, if we face an elbow injury, and we change how we move our shoulder to compensate for this injury. However, that does not guarantee that our performance will match as pre-injury level. Rather we are trying to cope with these changing circumstances to maintain our daily activities. Therefore, it is important to determine what percentage of normal performance CFlowNets and PPO can retain in the fault environments. To compute this we utilized the point of asymptotic performance across 10 million timesteps for CFlowNets and PPO and compared them with the asymptotic performance in the four fault environments. Overall, CFlowNets is able to retain approximately 68.43% to 94.74% of its normal asymptotic performance in the environment with faults 1, 2, and 4. However, in the fault 3 environment with the actuator damage, a substantial reduction of 78.95% is observed because this custom gym environment introduces a more complex non-linearity in the environment. Conversely, for fault 3, PPO also experienced a 59.54% asymptotic performance reduction. While this decrease is less severe than CFlowNets, it is a considerable reduction. For the rest of the fault environments, PPO was able to retain 78.05% to 85.37% of its original performance. This evaluation provides insight into the varied impact of each fault condition on the two algorithm's asymptotic performance. CFlowNets demonstrated a higher degree of resilience under specific fault environments such as fault 4, whereas PPO maintains a more stable performance retention in others.

A potential reason for CFlowNets' good performance is that it generates a distribution over all possible paths and samples from the most rewarding paths with a higher probability. CFlowNets's ability to generate a distribution over trajectory means that it can explore multiple potential paths concurrently. As a result, the possibility of the algorithm getting stuck in a local optimum reduces and makes it more suitable to find more globally optimal solutions, which makes them more sample efficient than standard reinforcement learning algorithms. By prioritizing the most rewarding paths, CFlowNets' sampling mechanism continuously fine-tunes its parameters based on the highest returns. As a result of this prioritization, convergence occurs faster and results in more optimal performance. Additionally, because of its off-policy nature, it can utilize stored experiences to learn effectively from a limited number of samples by reusing experiences multiple times. In continuous state and action space, CFlowNets' architecture may facilitate more accurate state generalization. In this way, it is capable of recognizing and acting optimally in unseen but similar states, which is invaluable in dynamic environments.

On the other hand, for every fault environment, PPO faces a performance dip initially in its learning curve, and it requires a lot of real experiences (timesteps) to reach asymptotic performance. The algorithm's lower sample efficiency can be attributed to a number of factors inherent to the architecture of PPO. First and foremost, PPO is an on-policy algorithm. Due to its on-policy nature, PPO's learning phase depends on the most recent experiences. However, in the initial stages of learning, these experiences are based on random or suboptimal policies. This on-policy feature may be beneficial in the long run, but it requires more timesteps to collect valid experiences that contribute to optimal policy improvements [34]. Other than that, the clipping mechanism of PPO can also play a role in the algorithm's poor sample efficiency. PPO utilizes a clipping mechanism in its objective function, which hinders large policy updates. This strategy prevents PPOs from overshooting policy updates and ensures training stability. Despite this, this clipping mechanism might be the reason why PPO's policy updates have a slower rate of improvement during the early phases of training. Therefore, the algorithm takes a long time to reach asymptotic performance.

In the context of sample efficiency, TD3 demonstrated similar performance to CFlowNets in all the fault environments illustrated in Figure 6.3 and Figure 6.4.

However, despite its good sample efficiency, the overall average reward trajectory was lower compared to CFlowNets and PPO. In the preceding chapters, we discussed how TD3 utilizes a twin-value network to estimate the Q-values and delayed policy updates to reduce overestimation bias and ensure a more stable training phase respectively. Additionally, TD3 uses target policy smoothing, which adds noise to the target policy, making the policy more robust and the estimation of Q-values more conservative. The delayed policy updates and target policy smoothing may result in efficient learning, but it also creates an obstacle to exploration. As a result, TD3 may produce lower rewards than other methods, such as CFlowNets and PPO. In addition to this, TD3 is an off-policy algorithm that learns from a replay buffer. Learning from stored experiences can lead to better sample efficiency. Nevertheless, it might also cause the model to be stuck in outdated and suboptimal policies, which may cause a lower average reward trajectory.

SAC (Soft Actor-Critic) exhibited a stable average reward trajectory for most of the experiments in the four custom gym environments. However, if we compare the performance with CFlowNets, PPO, and TD3, its average reward trajectory was lower. Despite this poor adaptive performance, one notable feature that was evident was its good exploration ability. In the context of sample efficiency, SAC was able to quickly achieve convergence for most of the fault environments. SAC is entropy-regularized, which means it uses an entropy term to encourage more exploration. As a result, SAC maintains a more stochastic policy and explores more in the environment which explains its good sample efficiency. Furthermore, due to the off-policy nature, SAC is more sample-efficient and requires fewer timesteps to achieve near-asymptotic performance. This may, however, lead to the algorithm being less responsive to recent changes than on-policy algorithms such as PPO, which might be an underlying reason behind SAC's lower accumulation of reward.

Lastly, DDPG (Deep Deterministic Policy Gradients) showed the lowest adaptive performance over 10 million timesteps, lagging in both sample efficiency and reward accumulation. There was fluctuation in average reward across all four fault environments, indicating a lack of adaptability. Apart from demonstrating instability, the results depicted a major abrupt dip in its performance. There could be various potential reasons why DDPG is not suitable for fault adaptation tasks. As discussed in Chapter 2, the DDPG algorithm has a tendency to cause function approximation errors and overestimation bias of the Q-values of the critic network. This can cause the agent to get stuck into a local optimum because of suboptimal policy updates, which can be a potential reason for DDPG's performance issues. Apart from this, its off-policy nature can also negatively affect learning, being less responsive to recent changes in the environment. Additionally, it has been reported in several studies [15] [35] that DDPG is highly sensitive to hyperparameters. The fact that we used the published hyperparameters [14] and did not perform extensive hyperparameter tuning may have contributed to the performance degradation.

6.4 Adaptation Speed and Sample Efficiency

As we continue our comparative analysis, we reach the fourth segment, which delves deeper into the adaptation speed and sample efficiency of each algorithm. Utilizing a grouped bar chart in the Figure 6.5 we gain more insights about each algorithm's efficacy in the four custom gym environments. In the graph, we have organized our collected data from third-segment experiments into four groups, each corresponding to one of the fault environments (Reduced ROM, Increased damping, Actuator Damage, and Structural Damage). The Y-axis quantifies the timesteps required for each algorithm to reach asymptotic performance. It is important to note that each of these five algorithms has a different execution time due to their distinct model architectures. To provide a clearer perspective on adaptation speed, we have incorporated the real-time duration for each algorithm to achieve convergence in an hour-minute format.

From Figure 6.5, we can see that CFlowNets demonstrated remarkable perfor-

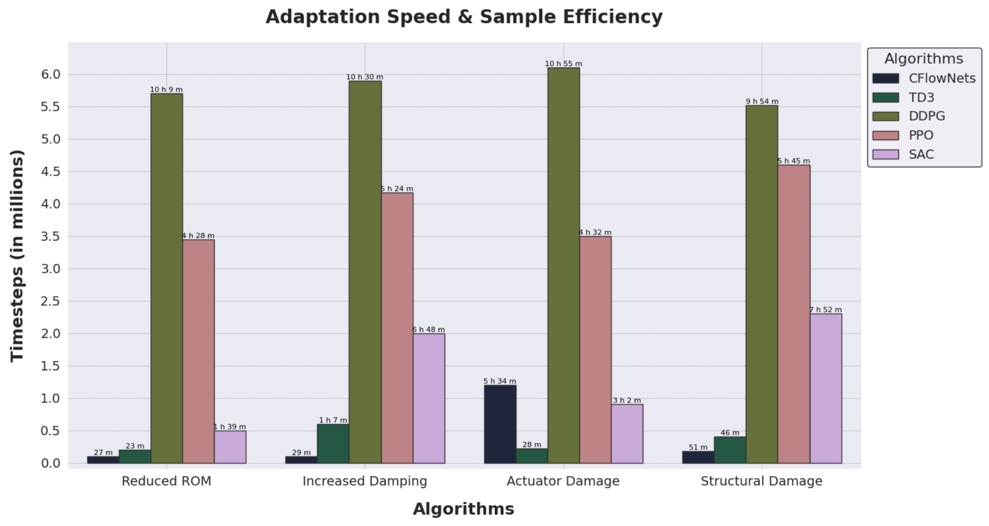


Figure 6.5: Adaptation Speed and Sample Efficiency in the four fault Reacher-v2 Environments (10 million timesteps). Execution time to achieve asymptotic performance is indicated on top of each algorithm's bar in an Hour-Minute format.

mance both in terms of fast adaptation and sample efficiency for most of the fault environments. The model took between 27 to 51 minutes and approximately only 100k – 200k timesteps to gain asymptotic performance for faults 1, 2 and 4. However, a noticeable anomaly is exhibited in the fault environment with actuator damage where CFlowNets policy struggled to adapt, requiring more than a million timesteps (approx. 1.2 million) with an execution time of 5 hours 34 minutes to converge. Further research and optimization are needed to enhance its universal applicability and performance in light of this anomaly. Nevertheless, the superior sample efficiency of CFlowNets is evident from this graphical representation.

TD3, on the other hand, had one of the best performances in the context of requiring many fewer real experiences (300k – 700k timesteps) and lower execution time to adapt. Its performance, although not surpassing CFlowNets in the majority of the environments, outperformed all the rest of the three reinforcement learning models with an execution time ranging from 23 minutes to an hour. This result demonstrates that among the RL algorithms, TD3 is a good candidate algorithm for robotic tasks that require exploration with a limited number of samples.

SAC showed mixed results across the four environments, positioning it as the third best in this aspect of our comparative study. It excels in the Reduced Range of Motion and Actuator Damage environments but encounters challenges in Increased Damping and Structural Damage, taking a relatively greater number of samples before it converges. Due to the different dynamics and nature of the fault environments, Soft Actor-critics policies might be adequate to handle certain complexities while being sensitive to others.

PPO, due to its on-policy nature, experiences significant challenges in terms of sample efficiency. From the grouped bar chart, it is evident that PPO necessitates much more real experiences ranging from 3.45 to 4.6 million timesteps and as a consequence requires a longer execution time before it can adapt to the malfunctions. This low sample efficiency may undermine its applicability in tasks where sample efficiency and faster adaptation are paramount considerations.

Finally, in this segment of our comparative analysis, DDPG ranked last, requiring a large number of timesteps (approximately 5.5 to 6.1 million timesteps) and execution time running between 9 to 11 hours to plateau in its performance curve. In real-time applications where rapid adaptation is crucial, its slower convergence speed can be a significant bottleneck.

6.5 CFlowNets' Compute Efficiency Analysis: GPU Memory Usage

The fifth segment of our experiments is dedicated to investigating the compute resource usage of CFlownets compared to the four RL algorithms. These algorithms' operational efficiency, particularly GPU memory consumption during the training phase, is crucial to their feasibility and applicability in real-world machine fault adaptation scenarios. The data collected for this comparison is the GPU memory usage of each algorithm in a normal Reacher-v2 algorithm. To collect the data we have used the GPUtil library which is a Python library dedicated to monitoring NVIDIA GPU utilization and memory usage. The GPUtil monitoring setup continuously retrieves the current GPU memory usage. The current GPU memory usage data was collected at regular intervals. An average of the collected data for each evaluation timestep was then calculated and plotted in the bar graph 6.6.

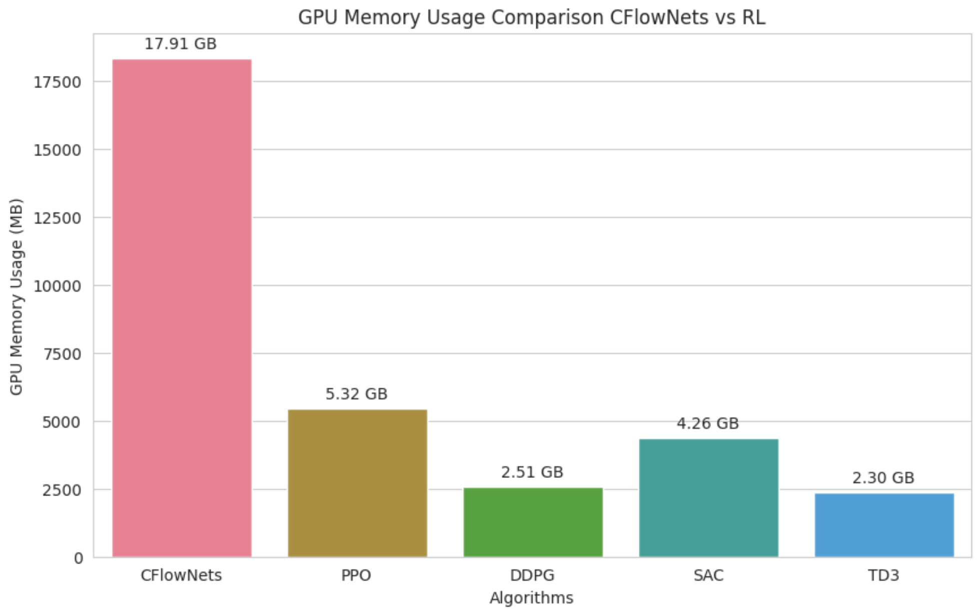


Figure 6.6: Bar chart illustrating the average GPU memory usage of various algorithms: CFlowNets, PPO, DDPG, SAC, and TD3 for Reacher-v2 (Normal Environment)

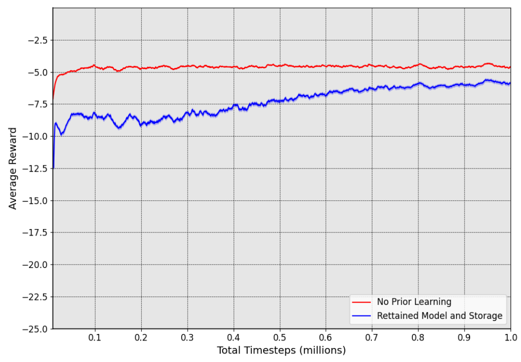
Although CFlowNets showed promising results in our previous experimental segments, when it comes to computing resource utilization, CFlowNets consumed on average a substantial 17.91 GB of GPU memory during its execution across the episodes. In contrast, the RL algorithms necessitated less than one-third of the GPU memory to perform a simple robotic task, Reacher-v2, which ranged from 2.3 to 5.32 GB. Reinforcement learning model-free algorithms such as PPO, SAC, DDPG, and TD3 learn a policy that directly maps states to action while sticking to the most rewarding paths. On the other hand, CFlowNets takes a more sample-efficient and sample-intensive approach. Instead of focusing on the most rewarding path, it generates a distribution over all possible paths and assigns higher probabilities on the more rewarding paths. Therefore, it explores the solution space more exhaustively. To execute this exhaustive and comprehensive approach, CFlowNets is bound to store and process a significant amount of data corresponding to all the possible paths and their associated states and actions. As a result, this algorithmic approach requires much more computation resources than standard reinforcement learning methods which in turn leads to a high GPU memory usage.

6.6 CFlowNets: A Transfer Learning Analysis

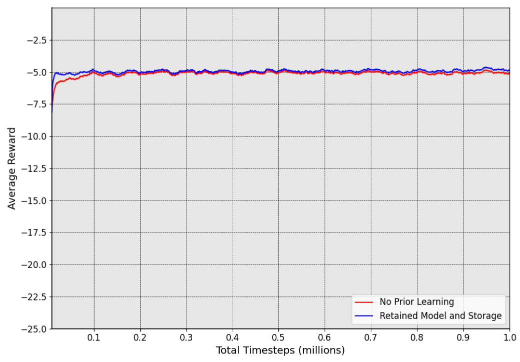
The utilization of transferring pre-trained models and replay buffer contents to a new training task has been proven to be beneficial in many deep-learning applications. In our final segment of the comparative evaluation, we implemented the concept of transfer learning to investigate if CFlowNets's performance in fault environments improves or not.

As this study is primarily focused on figuring out the efficacy of CFlowNets in machine fault adaptation tasks, we decided not to include other RL algorithms for this experiment. In this experiment, we trained CFlowNets in the original Reacher-v2 normal environment. The model parameters learned in the normal environment and the storage contents of the replay buffer were saved and then transferred to the four fault environments. We evaluated the performance of these models with retained model and storage contents with the performance of models with no prior learning. By no prior learning, we mean no knowledge transfer is involved, and the policy is trained online solely with the data collected in the fault environments.

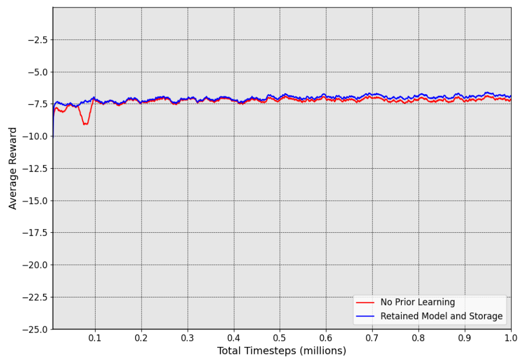
Figure 6.7a illustrates that transferring the model parameters and the reply buffer contents of the normal environment did not lead to any massive performance boost; rather, for the environment with the reduced range of motion, the performance was worse than the original model. Several factors could play the underlying reason for the performance degradation in the fault environment with a reduced range of motion. One of the potential reasons could be the mismatch of experience relevance. The experiences that were collected in the normal environment may have little relevance



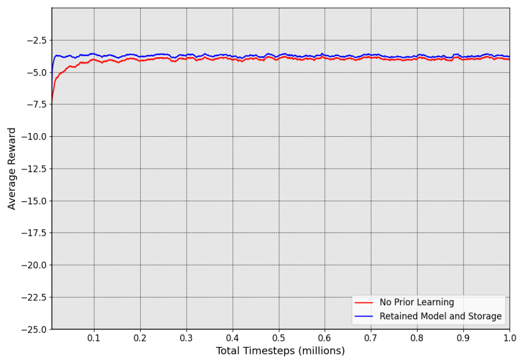
(a) Reduced ROM



(b) Increased Damping



(c) Actuator Damage



(d) Structural Damage

Figure 6.7: Early Performance of CFlowNets in Reacher-v2 fault environments. Comparison of performance is done between the CFlownets model with no prior learning and the CFlowNets model with retaining pre-trained model and replay buffer (data collected from the normal Reacher-v2 environment).

to the altered environment with a reduced range of motion where the dynamics of the environment hugely differ from the original environment. As a result, the learned policies and the past experience, including the action and state transition, may no longer be valid and counterproductive for the new task, leading to a sub-optimal performance for fault 1. The decline in performance could also be linked to distribution shift where the distribution of data that the model encounters when learning online in the fault environments, differs from the distribution of data on which the original model was trained on offline. This shift can lead to a significant performance drop because the patterns and parameters that the model learned previously may not be well suited to the new test data.

Conversely, when examining Figures 6.7b, 6.7c, 6.7d, it shows a more stable performance boost at the initial 100k timesteps across the rest of the three fault environments. In the fault environment with actuator fault damage illustrated in 6.7c, when no prior learning was done, there was a significant performance decline at the outset of CFlowNets's learning period. Yet, when we retained the model and replay buffer from the normal environment, it demonstrated a more stable and robust performance, circumventing any initial downturn. In our experimental setup, the process of learning models and collecting the experiences in the storage is akin to offline learning in the normal environment. In other words, when learning a policy in a normal environment, it is essentially learned offline. Such models, when deployed in fault environments, are exposed to online and out-of-distribution experiences. These new dynamics of the online and out-of-distribution experiences can degrade the performance of the models when no models and storage contents are retained. However, when we retain the pre-trained models and the storage contents in the post-fault tasks, the impact of encountering online and out-of-distribution experiences is substantially reduced in the beginning. That is why, when we transfer knowledge from the source task to these modified target tasks it initiates a jumpstart in the performance for fault environments 2, 3, and 4. The term "Jumpstart" [36] refers to the initial performance improvement that an agent experiences when it starts learning a new task using knowledge learned from a previous related task. When we transfer the pre-trained models along with experiences from the pre-fault normal environment to the fault environment, it mitigates the cold start because the agent already has a good estimate for the new environment by utilizing past experiences. Instead of learning from scratch, the replay buffer enables the agent to exploit good policies from the start without random exploration; hence, during the initial phase of the training, there is an improvement in performance compared to the models with no prior learning. In later timesteps, the model converges to a similar learning curve as the model with no prior knowledge. The advantage of the replay buffer diminishes because of buffer saturation. As the finite replay buffer is constantly being populated with new experiences from the current environment, it overwrites the old ones. As a result, after a certain period of timesteps, as the agent gathers more samples from faulty environments, the old experiences may become less useful.

Summary. To summarize, each segment of our experiments offers key insights that help us address the initial five research questions. Throughout the series of experiments, we determine that CFlowNets applicability in continuous control tasks and its algorithmic structure is well suited for effective adaptation in simple robotic simulations. Although CFlowNets was outperformed by PPO in two out of four fault environments in terms of average return, however, due to a limited time budget, if the task demands fast adaptation requiring the least amount of experience with good performance, CFlowNets excels. Nevertheless, the significant resource consumption of the algorithm may pose certain challenges to its application in real-world settings.

It is crucial to mention that in all of the experiments, the reinforcement learning algorithms did not have any pre-trained components at the beginning of the learning phase. However, CFlowNets had an added advantage because it had access to a pre-trained retrieval network which resulted in the agent having a better estimate of the environment's state and action space. This pretraining aspect might be a reason why CFlowNets was able to quickly adapt and gain convergence compared to other RL algorithms which had to learn from scratch.

However, it is also imperative to acknowledge the fact that CFlowNets operated in a sparse reward-structured environment whereas the RL algorithms received intermediate rewards. As a result, all the RL algorithms had an advantage because these intermediate rewards at each timestep guided the learning process resulting in a more efficient policy update. On the other hand, CFlowNets experiments were done in a sparse reward setting where the agent received a reward only at the terminal state at the end of an episode. This infrequent feedback from the environment is a huge disadvantage that can hinder the overall adaptation performance of CFlowNets. Therefore, while CFlowNets' pre-trained retrieval network offered an initial advantage, the sparse reward setting presented a unique challenge, balancing the overall comparison with traditional RL algorithms. The next step in this research will be to train both the retrieval and flow networks simultaneously to make the comparison more fair.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

The purpose of this comparative study was to determine the feasibility of flow networks (specifically CFlowNets) in the field of robotic applications. Flow network implementations have already shown promising results in molecular generation in drug discovery compared to many reinforcement learning algorithms. However, its potential in robotics still remains unexplored. This thesis is dedicated to answering some fundamental questions about CFlowNets:

- *Can Generative Flow Networks be utilized in Robotic Applications?*
- *Is it possible to develop a Continual Generative Flow Network Learning approach for machines well suited for continual interaction with environments? In other words, Can GFlowNets or CFlowNets theoretical formulations work with continuous action and state space?*
- *Can Generative Flow Networks adapt when a fault is introduced in the robotic environment? In other words, can it be utilized for fault adaptation tasks?*
  - *If yes, then does it perform better than state-of-the-art reinforcement learning methods (PPO, SAC, TD3, and DDPG) in terms of adaptation speed and sample efficiency?*

- *How does the computational resource consumption of flow networks compare to traditional RL algorithms?*
- *Does incorporating the transfer of task knowledge enhance performance efficiency for Generative Flow Networks?*

Although GFlowNets serves as the foundation of the flow network concepts, its theoretical formulations are only able to deal with discrete tasks where there are a limited number of state and action pairs and each edge in the space represents one discrete action. Real-world robotic application involves environments with continuous state and action space where multimodal reward distribution may be present. In our research, we shifted our focus from GFlowNets to a continuous variant of GFlowNets called CFlowNets, which is capable of utilizing its novel training framework to produce policies in a continuous state-action space.

We have conducted six segments of experiments in our comparative study. Each set of experiments offered a key insight into CFlowNets' performance in a robotic environment. Overall, from the experimental results, we can conclude that CFlowNets can not only be employed in a robotic simulated environment, but it is one of the top-tier algorithms when it comes to incorporating hardware fault adaption to machines for the Reacher-v2 environment. Across our four custom gym environments with faults, CFlowNets have exhibited excellent sample efficiency and adaptation required the least number of timesteps. As a whole, the asymptotic performance of CFlowNets was also pretty high compared to most standard reinforcement learning algorithms. Among the RL algorithms, PPO also demonstrated commendable adaptation performance. PPO's average reward trajectory converged higher than CFlowNets in some fault environments, nevertheless, it required a large number of experiences before it could stabilize. As a result, the algorithm took longer to adapt in all the fault environments compared to CFlowNets.

Based on our observations of the experimental results, CFlowNets, especially with retaining model and replay buffer, can be considered a promising option for rapid adaptation in robots. In practical terms, when a robot encounters malfunctions in the real world and immediate maintenance is either unavailable or prolonged, CFlowNets can be considered a good candidate due to its rapid adaptation and exceptional sample efficiency. PPO also exhibited good adaptation abilities but as it takes longer to adapt, it might not be suitable for tasks that require immediate adaptation with a short time delay. From the results, we can also conclude that TD3 has a consistent adaptation performance, although the overall reward trend may not be as high compared to CFlowNets and PPO. It is worth mentioning that the performance of CFlowNets may not be superior to that of RL in every scenario, as the nature of the task may affect the performance. The use of CFlowNets for sampling distributions over complex spaces may indeed be advantageous for tasks that require a diverse set of solutions through comprehensive exploration. However, for tasks that are strictly about maximizing cumulative rewards, traditional RL algorithms might be more suitable. Additionally, one of the limitations of this study was that it was conducted on a simple simulated robotic task. It would be interesting to investigate if the performance of CFlowNets remains good as it scales up to more complex and larger problems.

Algorithms like CFlowNets require substantial GPU memory which can be resource-intensive, limiting their applicability in real-world scenarios where resources are constrained. In real-world robotic applications, some hardware designs may require minimal resources which may not be able to support CFlowNets's memory-intensive computations. Additionally, high GPU memory requirements with expensive hardware will also add to computation costs. We have experimented with CFlowNets on a simple simulated robotic environment but in real-world tasks with more complex environments, because of its resource-intensive behaviours the algorithm may not be as scalable and may face deployment issues in embedded systems with limited capacity.

Although CFlowNets' adaptation performance did show initial improvement for most tasks when we incorporated saved models and replay buffer contents into models dealing with fault environments, the improvement was not significant. Because of the changing environment dynamics in fault environments, the retained model and replay buffer helped mitigate "cold-start" in the early stages of training. When the fault occurs in the environment and the models face online and out-of-distribution experiences, the offline learned model along with the past experiences stored in the replay buffer, facilitates early adaptation to the fault tasks. According to our results, it is best to retain the pre-fault task model parameters and replay buffer contents before deployment.

To conclude, in this thesis, we have investigated the potential and limitations of an alternative to traditional reinforcement learning algorithms in robotics applications. We have successfully implemented CFlowNets to add fault tolerance to a simulated robot. Based on certain performance metrics, we have conducted a comprehensive evaluation of some key aspects of CFlowNets and compared them with four different state-of-the-art reinforcement learning widely popular in their control system and robotics applications. We believe this study is the first step in utilizing a new supplementary approach to exploration-biased robotic tasks with machine fault adaptation. We hope that further research in CFlowNets will lead to more reliable and efficient real-world robots in various fields, making them a common choice for tasks where exploration and adaptability are key.

7.2 Future Work

We identified some of the future works that can extend our comprehensive analysis and experiments to enhance and optimize the performance and applicability of CFlowNets along with the RL algorithms in continuous robotic environments, particularly those involving machine faults.

Hyperparameter Tuning: One of the key limitations of our research was that an intensive hyperparameter search was not conducted for the implementation of the RL algorithms. In most cases, published hyperparameters for the same Reacher-v2 task were used with minor tuning. We did a selective hyperparameter tuning and focused more on hyperparameters that facilitate exploration. For a more fair comparison, we should do an intensive hyperparameter search to optimize the RL algorithms in the normal environment.

Environmental Dynamics: All of the evaluations in this study were done on a single type of robotic environment. We created four custom gym environments, but at its core, all of them were different variations of the same Reacher-v2 environment. Future experiments of CFlowNets should be done in various robotic environments with differing environment dynamics and complexities to gather a deeper understanding of the adaptability of the algorithm. This will help us determine if CFlowNets policies are suitable for certain types of tasks and potentially unsuitable for others.

Meta RL: One of the most prominent candidates for fault adaptation tasks is the Meta RL method. Meta RL's "learning to learn" approach enables a domain adaptation strategy where the agent learns learning strategies that are invariant to a range of domains. As a result, these algorithms are more suitable to quickly adapt to new tasks where the environments are non-stationary and the dynamics can change over time. We plan to conduct future research to compare the adaptation of CFlowNets with Meta RL algorithms to determine the generalization capability of both these frameworks across multiple fault scenarios.

Combining CFlowNets with RL: From our findings in this thesis, we conclude that CFlowNets has excellent exploratory ability and sample efficiency. However, RL is still the way to go for large-scale tasks where reward maximization is vital. One of the future extensions of this study could be to harness CFlowNets's exploratory ability and combine it with RL methods to maximize rewards.

Gradual Hardware Faults: In our study, we have simulated fault environments by changing attribute variables in the XML file of our robotic environment. Each of these attributes defines certain aspects of the robot. However, all of the four-fault environments had constant fault types where the values of the modified attributes remained the same over time. In real-world scenarios, a robot typically faces gradual malfunction over time where the severity of faults generally increases; for instance, the joint becomes more and more restrictive because of gradual wear and tear. Instead of sudden and abrupt malfunction to the environment, for future experiments, the fault could be simulated in such a way that the values of the attributes gradually change as the timestep progresses. As the dynamics of the environment change bit by bit, it would be interesting to investigate if CFlowNets can demonstrate better adaptability in this sort of progressive learning setting.

Real-world Robotic Implementation: All of our experiments were done in a simulated robotic environment (Reacher-v2). We intend to deploy our work into real-world robots to validate and optimize the performance.

Bibliography

[1] R. Teti, K. Jemielniak, G. O'Donnell, and D. Dornfeld, "Advanced monitoring of machining operations," *CIRP annals*, vol. 59, no. 2, pp. 717–739, 2010.

[2] M. Riazi, O. Zaiane, T. Takeuchi, A. Maltais, J. Günther, and M. Lipsett, "Detecting the onset of machine failure using anomaly detection methods," in *Big Data Analytics and Knowledge Discovery: 21st International Conference, DaWaK 2019, Linz, Austria, August 26–29, 2019, Proceedings 21*, Springer, 2019, pp. 3–12.

[3] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey," *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, 2017.

[4] California Wildlife Center, *Helping birds get their "wings" back*, https://cawildlife.org/helping-birds-get-their-wings-back/, Accessed: 2024-01-15.

[5] The Engineer, "Forces of nature: Biomimicry in robotics," *The Engineer*, Accessed: 2024-01-15. [Online]. Available: https://www.theengineer.co.uk/content/features/forces-of-nature-biomimicry-in-robotics.

[6] Y. Bengio, S. Lahlou, T. Deleu, E. J. Hu, M. Tiwari, and E. Bengio, "Gflownet foundations," *Journal of Machine Learning Research*, vol. 24, no. 210, pp. 1–55, 2023.

[7] Y. Li, S. Luo, H. Wang, and J. Hao, "Cflownets: Continuous control with generative flow networks," *International Conference on Learning Representations*, 2023.

[8] R. Isermann, "Supervision, fault-detection and fault-diagnosis methods—an introduction," *Control engineering practice*, vol. 5, no. 5, pp. 639–652, 1997.

[9] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches," *IEEE transactions on industrial electronics*, vol. 62, no. 6, pp. 3757–3767, 2015.

[10] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance.* Springer Science & Business Media, 2005.

[11] W. Zhang, G. Peng, C. Li, Y. Chen, and Z. Zhang, "A new deep learning model for fault diagnosis with good anti-noise and domain adaptation ability on raw vibration signals," *Sensors*, vol. 17, no. 2, p. 425, 2017.

[12] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2012, pp. 5026–5033.

[13] G. Brockman *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[14] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR, San Juan, Puerto Rico, May 2-4, Conference Track Proceedings*, 2016.

[15] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*, PMLR, 2018, pp. 1587–1596.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, 2017.

[17] T. Haarnoja *et al.*, "Soft actor-critic algorithms and applications," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1812.05905.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[19] N. Malkin, M. Jain, E. Bengio, C. Sun, and Y. Bengio, "Trajectory balance: Improved credit assignment in gflownets," *CoRR*, vol. abs/2201.13259, 2022. arXiv: 2201.13259. [Online]. Available: https://arxiv.org/abs/2201.13259.

[20] Y. Bengio, K. Malkin, and M. Jain, *The gflownet tutorial*, 2022. [Online]. Available: https://tinyurl.com/gflownet-tutorial.

[21] E. Bengio, M. Jain, M. Korablyov, D. Precup, and Y. Bengio, "Flow network based generative models for non-iterative diverse candidate generation," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 381–27 394, 2021.

[22] C.-S. Huang, S.-L. Hung, C. Lin, and W. Su, "A wavelet-based approach to identifying structural modal parameters from seismic response and free vibration data," *Computer-Aided Civil and Infrastructure Engineering*, vol. 20, no. 6, pp. 408–423, 2005.

[23] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[24] F. Yang, C. Yang, D. Guo, H. Liu, and F. Sun, "Fault-aware robust control via adversarial reinforcement learning," in *IEEE 11th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems*, 2021, pp. 109–115.

[25] K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret, "Reset-free trial-and-error learning for robot damage recovery," *Robotics and Autonomous Systems*, vol. 100, pp. 236–250, 2018.

[26] I. Clavera, A. Nagabandi, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn, "Learning to adapt: Meta-learning for model-based control," *arXiv preprint arXiv:1803.11347*, vol. 3, p. 3, 2018.

[27] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 1126–1135.

[28] W. Jin, R. Barzilay, and T. Jaakkola, "Junction tree variational autoencoder for molecular graph generation," in *International conference on machine learning*, PMLR, 2018, pp. 2323–2332.

[29] T. Sterling and J. J. Irwin, "Zinc 15–ligand discovery for everyone," *Journal of chemical information and modeling*, vol. 55, no. 11, pp. 2324–2337, 2015.

[30] Y. Xie *et al.*, "Mars: Markov molecular sampling for multi-objective drug discovery," *9th International Conference on Learning Representations, ICLR, Virtual Event, Austria, May 3-7*, 2021.

[31] G. Brockman *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[32] D. Yarats and I. Kostrikov, *Soft actor-critic (sac) implementation in pytorch*, https://github.com/denisyarats/pytorch_sac, 2020.

[33] V. Mnih *et al.*, *Asynchronous methods for deep reinforcement learning*, 2016. arXiv: 1602.01783 `[cs.LG]`.

[34] J. Queeney, I. C. Paschalidis, and C. G. Cassandras, "Generalized Proximal Policy Optimization with Sample Reuse," *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS, December 6-14, virtual*, 2021.

[35] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7*, pp. 3207–3214, 2018.

[36] M. E. Taylor and P. Stone, "Transfer Learning for Reinforcement Learning Domains: A Survey," *Journal of Machine Learning Research*, vol. 10, no. 1, pp. 1633–1685, 2009.