Verification of Neural Networks for Safe Reinforcement Learning

Polaka Surendra Kumar Reddy

Department of Computer Science, Bioengineering, Robotics and System Engineering (DIBRIS)

University of Genova

Supervisor

Armando Tacchella , Stefano Demarchi

In partial fulfillment of the requirements for the degree of

Master of Science in Robotics Engineering

March, 2024

Acknowledgements

First and foremost, I would like to take this opportunity to express my sincere gratitude to my Professor Armando.tacchella and supervisor stefano.demarchi for his valuable advice, guidance and willingness during my thesis. Both expertise and feedback have been invaluable for the success of the work I would like to thanks to all the professors during my course for sharing the knowledge. My close friends have shown all their support and encouragement for the work I am doing. Last and before all, I would like to express my sincere gratitude to my parents who gave me love and hope to succeed

" This Thesis dedicated to my Parents "

Abstract

This thesis provides a comprehensive exploration of Reinforcement Learning (RL) from fundamental principles to advanced tools and experimental applications. Beginning with motivations, Objectives. The transition from tabular methods to neural networks, specifically Convolutional Neural Networks and the Actor-Critic architecture, is highlighted, with a focus on the architectural elegance and the introduction of Soft Actor Critic (SAC).

The thesis then delves into practical implementations through tools and frameworks like Open AI Gym, PyTorch, TensorFlow, and the Never2 Tool . The Never2 Tool's architectural design, installation process, and procedures for building models, defining properties, and handling models through a command-line interface are outlined. The tool's functionalities extend to training networks, verification strategies, and output visualization.

Experimental results in the Classic control environment are detailed, evaluating different methods and neural network approaches. The network verification process is emphasized, ensuring the robustness of the tool. The thesis concludes by contributing a holistic perspective on RL, bridging theoretical foundations with practical applications, and paving the way for future advancements in RL research and real-world implementations.

Contents

1 Introduction **1**

1.1 Motivations 2

1.2 Research Objectives 4

1.3 Contribution of the Thesis 5

2 Reinforcement Learning **7**

2.1 Fundamentals of Reinforcement Learning 8

2.1.1 Core Components of Reinforcement Learning: 9

2.1.2 The Markov Decision Problem 10

2.1.3 Bellman Equations 11

2.2 Evolution from Tabular Methods to Neural Networks in Reinforcement Learning 12

2.2.1 Architectural Elegance in Convolutional Neural Networks . 14

2.2.2 Actor Critic Architecture 15

2.3 Soft Actor Critic(SAC) 17

3 Tools and Frameworks **19**

3.1 Environments 21

3.2 Pytorch 24

3.3 Tensorflow 26

3.4 Comparision between Tensorflow and Pytorch 28

4 Learning and Verification with NeVer2 30

4.1 Installation 32

4.2 Software Architecture 34

4.3 Procedure 35

4.3.1 Building the model 36

4.3.2 Defining the Property 38

4.3.3 Load and Save Models 40

4.3.4 Command-line interface 41

4.4 Training the Network 42

4.5 Verification Strategy 45

4.6 Output Visualization 45

5 Case Study 47

5.1 Software and Hardware Details 49

5.2 Classic control Environment 50

5.2.1 Experiments 51

5.2.2 Basic Method 52

5.2.3 Q learning Method 56

5.2.4 Q learning Neural network 60

5.2.5 Pytorch 62

5.3 Network Verfication 66

6 Conclusions 70

References 74

List of Figures

1.1 Machine Learning is subset of Artificial Intelligence 2

2.1 Overview of the different components in the Reinforcement Learning 9

2.2 Interaction loop between Agent and Environment. 10

2.3 Scientific Diagram of Neural Network 13

2.4 Schematic Diagram of Convolutional Neural Networks 15

2.5 Sample Diagram of Actor-Critic Architecture 16

2.6 Schematic Diagram of Soft Actor-Critic extensive Reward Function 18

3.1 Open AI gym Environments 22

3.2 Pytorch workflow 25

3.3 Tensor flow Key features 27

4.1 UML Diagram of Never2 Tool 36

4.2 Interface of Never2 Tool after adding defining Properties 37

4.3 ReLU Neural network architecture 38

4.4 sample Diagram of Defining Polyhedral Property 39

4.5 Sample Diagram of Defining smt property 40

4.6 sample diagram of parameters how to train the Network 43

4.7 Output Visualization to verify the Network 46

5.1 Basic Method output Results 53

5.2 The Graph Represents Total Reward and Average Reward Per Per step to Per Episode 55
5.3 The Graph Represents Exploration and Exploitation Per Episode 55
5.4 Q learning Output Results 57
5.5 Graph Represents Epsilon per Episode 59
5.6 Graph Represents Q value Magnitude Per Episode 59
5.7 Q learning Neural Network Output specifications 63
5.8 Total reward per Episode 63
5.9 total reward per episode using pytorch 65

List of Tables

5.1 Development Machine specifications 50
5.2 Network output and Input Properties 68
5.3 Never-2 output Results 69

Chapter 1

Introduction

Machine Learning (ML), a pivotal branch of artificial intelligence, is dedicated to developing models and algorithms capable of learning from data, continuously refining their performance. It finds widespread application in finance, healthcare, marketing, and manufacturing, reshaping problem-solving and decision-making in various industries.

At its core, ML's intrinsic ability to learn from experience allows it to iteratively enhance understanding as it encounters more data. This introduction explores fundamental ML concepts, including supervised and unsupervised learning, delving into the intricate workflow and key terminologies like features and labels. The aim is to shed light on ML's transformative potential across industries while acknowledging real-world challenges.

Reinforcement Learning (RL) is a distinct facet of ML, introducing a different paradigm where an agent learns through interaction with an environment. Unlike supervised and unsupervised learning, RL involves an agent making sequential decisions to maximize cumulative rewards. This approach is particularly powerful in dynamic and complex scenarios.In finance, RL can optimize algorithmic trading strategies by learning from market feedback. Healthcare applications involve

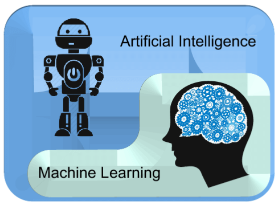


Figure 1.1: Machine Learning is subset of Artificial Intelligence

personalized treatment plans, where RL tailors interventions based on patient responses. Marketing strategies benefit from RL's ability to adapt to changing consumer behaviors, optimizing advertisement campaigns for maximum impact.

The transformative impact of ML and RL is evident in diverse sectors. In finance, ML algorithms assess risks and detect fraud, while RL optimizes trading strategies. Healthcare embraces ML for diagnostics and drug discovery, with RL personalizing treatment plans. Marketing leverages ML for personalized recommendations, and RL adapts strategies based on evolving consumer preferences. While ML showcases its power to revolutionize industries, RL introduces a dynamic element by enabling agents to learn optimal behaviors through trial and error. This journey into ML and RL illuminates their combined potential, emphasizing responsible deployment to navigate challenges and fully harness the benefits of these transformative technologies.

1.1 Motivations

Reinforcement Learning (RL) based on neural networks has demonstrated remarkable success in various applications, from playing games to controlling complex systems. However, a significant challenge arises when these RL models produce policies that may exhibit unsafe behaviors in real-world scenarios. Ensuring the safety and reliability of RL-based systems becomes paramount, prompting the need for verification methods to identify and rectify potential risks.

Neural networks, the backbone of many RL models, are highly flexible and can learn complex relationships from data. Yet, this flexibility also poses a risk of learning unsafe policies that might lead to unintended consequences. In safety-critical domains like autonomous vehicles, robotics, or healthcare, the consequences of unsafe policies could be severe. Verification methods offer a systematic approach to addressing these safety concerns. These methods involve assessing the RL model's behavior against predefined safety specifications or constraints. By formally analyzing the model's decision-making processes, verification techniques aim to identify situations where the learned policy may deviate from safe behavior.

One approach to verification involves leveraging formal methods such as model checking, where the RL model's behavior is systematically verified against a set of safety properties. This process helps identify potential vulnerabilities or areas where the model may fail to adhere to safety specifications. Additionally, incorporating human feedback and domain knowledge into the training process can act as a form of verification. By combining RL with techniques like imitation learning, where the model learns from human demonstrations, the training process can be guided to prioritize safe and desirable behaviors.

In the pursuit of safe RL, researchers are actively exploring the integration of verification methods into the training pipeline. This includes developing techniques to certify safety properties during the learning process, providing assurances that the resulting policies align with predefined safety criteria. Ultimately, addressing safety concerns in RL-based neural networks requires a multidisciplinary approach that combines advances in machine learning, formal methods, and domain-specific knowledge. By integrating verification methods into the development cycle, researchers aim to enhance the reliability and safety of RL models, fostering their responsible deployment in critical real-world applications.

1.2 Research Objectives

The intersection of reinforcement learning (RL), PyTorch, and specialized control tools such as Never2 in a new era of transformative development in artificial intelligence (AI). It provides a comprehensive framework for understanding development and challenges. and ensuring the security of neural networks. This approach leverages the dynamic computer graphics capabilities of PyTorch, formal verification techniques built into Never2, and interactive interfaces for property specification, providing a comprehensive approach with deep implications across fields.

Reinforcement learning, a central part of this paradigm, has gained significant traction due to its ability to train agents with the environment through trial and error, resulting in decision-making skills spanning applications from games to robotics. RL algorithms such as deep Q networks (DQN) and political gradient methods have shown significant achievements in forming the backbone of intelligent systems that can adapt and learn from their experiences.

Neural networks, the basis of many modern AI applications, introduce a layer of complexity and nonlinearity, especially in the context of deep learning, which involves several hidden layers. These networks, inspired by the human brain, have proven to be very effective in tasks such as image recognition, natural language processing and learning. However, the transparency of deep neural networks presents challenges to understand their decision-making processes, which requires the development of tools and methods for verification and transparency.

PyTorch, an open source deep learning framework, emerged in this landscape. Known for its dynamic computation graph and user-friendly interface, PyTorch facilitates the implementation of complex neural network architectures. Its flexibility has made it the best choice for researchers and professionals involved in developing cutting-edge AI applications. PyTorch's seamless integration with reinforcement learning algorithms has simplified the process of building and testing RL models, enabling rapid prototyping and iterative development.

Never2, a dedicated neural network tool, plays a critical role in addressing growing concerns about the reliability and security of neural networks. In this context, verification requires that neural network models conform to certain properties, constraints, or security guidelines. Never2 uses formal verification methods to mathematically prove the correctness of these models and provides a robust platform for verifying user-defined properties. Its interactive interface allows users to visualize the verification process, explore counterexamples, and gain insight into the decision limits of the neural network. Importantly, Never2 is designed to seamlessly integrate with PyTorch, simplifying the verification process and making it more widely accessible to researchers and practitioners.

The integration of these components has far-reaching effects in various fields. In safety-critical applications such as autonomous vehicles and healthcare, the ability to formally verify neural network models becomes paramount to ensure the reliability and ethical operation of AI systems. The convergence of reinforcement learning, PyTorch, and neural network Tools, exemplified by Never2, represents an important step forward in the field of artificial intelligence. This integrated approach not only meets the challenges of developing and understanding complex neural networks but also lays the foundation for building reliable and transparent AI systems.

1.3 Contribution of the Thesis

The Primary objective of the thesis consists of three steps :

1. Create RL-based UAV control policies: We intend to construct Neural Networks (NNs) that can autonomously execute control tasks required for UAVs

using RL algorithms such as Proximal Policy Optimization (PPO) and Deep Deterministic Policy Gradients (DDPG). These responsibilities include navigation, obstacle avoidance, and trajectory optimization. The emphasis is on using RL's learning capabilities to adapt to diverse and dynamic settings, giving UAVs the agility required for real-world applications.

2. Evaluating Robustness in Simulation and Real-World Scenarios: The trained NNs will be empirically tested in both simulated and real-world settings. Simulations provide a controlled environment for preliminary evaluations, allowing us to examine the performance of learned control strategies under a variety of scenarios. Following that, real UAVs will be used to test the robustness of these strategies in complicated and dynamic circumstances. The purpose of this empirical evaluation is to identify potential obstacles, strengths, and flaws of learnt control strategies when applied to actual UAVs

3. Integrate Formal Verification approaches: To evaluate and verify the trained NNs, we will use cutting-edge NN verification approaches, including methodologies pioneered in our lab. Formal verification is a methodical way to finding potential flaws, vulnerabilities, or violations of control policy safety restrictions. The goal is to improve the reliability of UAV control systems by proactively addressing these concerns. Formal verification will provide an additional degree of confidence by confirming that control rules correspond to stated specifications and safety standards

This work has important implications for the development and deployment of UAVs in real-world situations. The goal is to contribute to the creation of a reliable and robust UAV control system by integrating RL methods, empirical tests, and formal verification. This work mainly focuses on the growing need for special and safe technologies, especially in applications where mistakes have high consequences

Chapter 2

Reinforcement Learning

Reinforcement Learning (RL) has emerged as a vibrant field in machine learning, propelled by recent strides in deep learning (DL), which paved the way for the evolution of deep reinforcement learning. Positioned as the third paradigm in machine learning, alongside supervised and unsupervised learning, RL introduces a novel approach to decision-making problems. The core concept revolves around the agent, the central figure in RL, engaging in a continuous cycle of trial and error. Within this dynamic, the agent discerns valuable decisions from penalizing ones by leveraging information derived from a reward signal, mirroring the trial-and-error process inherent in human and animal behavior.

To comprehend the current state-of-the-art in RL, this chapter embarks on a journey through the theoretical underpinnings of traditional RL, establishing the notation used. It then seamlessly transitions towards the realm of deep RL, providing an introductory exploration into deep learning fundamentals. The discussion delves into essential algorithms, with a keen focus on those integral to the thesis project. The ultimate section aims to paint a vivid picture of the contemporary landscape of deep RL as applied to autonomous systems and real-world robotic tasks, setting the stage for the subsequent exploration.

Before delving into the thesis results, it is paramount to grasp the intricacies of this paradigm. The fusion of RL with deep learning not only enhances function approximation but also reshapes the landscape of decision-making processes. This synthesis captures the essence of learning through sequential experimentation, enriching the agent's ability to navigate complex environments. As we unravel the chapters that follow, the convergence of RL and deep learning unfolds as a potent force, driving innovation in autonomous systems and real-world robotic applications.

2.1 Fundamentals of Reinforcement Learning

Reinforcement Learning (RL) is a paradigm in machine learning where an agent learns to make decisions by interacting with an environment. The agent is the central entity in this framework, responsible for decision-making. This could manifest as a physical robot, a software algorithm, or any system with the capability to take actions within a given environment. Understanding the components of RL and the relationship between the agent, environment, actions, and rewards is essential to grasp the dynamics of this learning approach. The agent's [1]primary objective is to maximize its cumulative reward over time by learning a strategy or policy that maps environmental states to actions. States represent the current situation or configuration of the environment, actions are the decisions made by the agent, and rewards provide feedback on the desirability of those actions. The environment is the external system with which the agent interacts, and it responds to the actions taken by the agent by transitioning to new states and providing corresponding rewards.

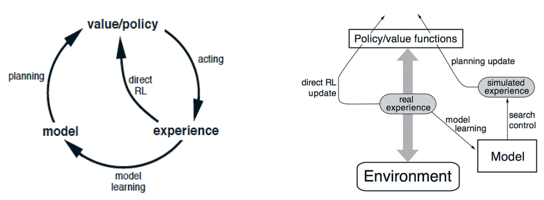


Figure 2.1: Overview of the different components in the Reinforcement Learning

2.1.1 Core Components of Reinforcement Learning:

Agent : The agent encapsulates the decision-making entity within the RL system. It can take various forms, from a physical robot navigating a real-world environment to a software algorithm playing a game. The agent's decision-making process is guided by a policy, a mapping from states to actions. The goal is to learn the optimal policy to improve and maximize the cumulative reward.

Environment: The environment is the external context in which the agent operates. It could be a physical space, a simulated world, or any system that the agent interacts with. The environment has a state, which represents the current situation. The agent's actions influence the environment, causing it to transition to new states

Actions:Actions are the decisions or moves made by the agent within the environment. The set of possible actions is defined by the task at hand and the capabilities of the agent. The agent's goal is to learn a policy that selects actions leading to favorable outcomes, i.e., actions that result in high cumulative rewards.

Rewards: Rewards are numerical values that provide feedback to the agent about the desirability of its actions. The agent's main goal is to improve and maximize the cumulative reward over time. Positive rewards encourage the agent to repeat actions that lead to favorable outcomes, while negative rewards or punishments discourage undesirable actions.

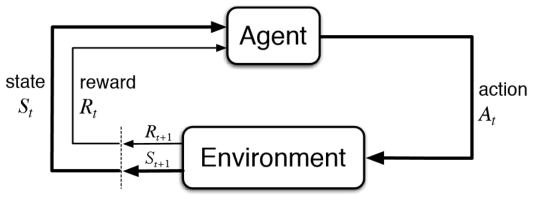


Figure 2.2: Interaction loop between Agent and Environment.

2.1.2 The Markov Decision Problem

A Markov Decision Problem (MDP) is a formal mathematical framework for modeling decision-making problems involving uncertainty over time. It is characterized by a set of states, actions, transition probabilities, rewards, and a discount factor. In addition to these components, MDPs involve the concepts of policies, models, and value functions, which play crucial roles in finding optimal strategies for decision-making.

1. Policies: A policy in the context of an MDP is a strategy or a rule that defines the agent's behavior. It maps states to actions, specifying the action the agent should take in each possible state. Policies can be deterministic, prescribing a single action for each state, or stochastic, providing a probability distribution over actions. The goal is to find an optimal policy that maximizes the expected cumulative reward over time.

2. Models: The transition model, often denoted by P P, describes the dynamics of the MDP. It defines the probabilities of transitioning from one state to another based on the agent's actions. In a Markovian setting, the future state depends only on the current state and action, not on the history of events leading to that state. The transition model is a fundamental component for predicting the evolution of the system and is crucial for planning under uncertainty.

3. Value Functions: Value functions are central to solving MDPs and evaluating the desirability of different states and actions. There are two main types

of value functions:The state value function represents the expected cumulative reward starting from a given state and following a particular policy. It quantifies the desirability of being in a particular state under a specific policy. The action value function, also known as the Q-function, represents the expected cumulative reward starting from a given state, taking a specific action, and then following a particular policy. It evaluates the desirability of taking a particular action in a specific state under a given policy.

2.1.3 Bellman Equations

The recursive relationships between state values play a crucial role in understanding the dynamics of Markov Decision Processes (MDPs). These equations capture the dependence of a state's value on the values of its successor states. The Bellman equations, depicted in (2.7) further emphasize this by indicating that the next state is sampled from the environment.

$$V(s) = maxa(R(s,a) + V(s')) \tag{2.1}$$

State(s): current state where the agent is in the environment. Next State(s'): After taking action(a) at state(s) the agent reaches s'. Value(V): Numeric representation of a state which helps the agent to find its path.V(s) here means the value of the state s. Reward(R): treat which the agent gets after performing an action(a). R(s): reward for being in the state s.R(s,a): reward for being in the state and performing an action a .R(s,a,s'): reward for being in a state s, taking an action a and ending up in s' textbfExample Good reward can be +1, Bad reward can be -1, No reward can be 0.

The essence of value functions lies in their ability to establish a partial ordering among policies. Specifically, for policies and ', if V is greater than or equal to V' for every state in the state space S, then is considered better than or equal to ’, denoted as   ’. This ordering sets the stage for the sanity theorem, which posits that for any MDP, there exists an optimal policy, denoted as , surpassing or equal to all other policies, i.e.,   for any policy . Moreover, all optimal policies share the achievement of the optimal state-value function and the optimal action-value function.

Despite the significance of the Bellman optimality equation in characterizing optimal policies, its solution is non-linear, lacking a closed-form expression. . These iterative approaches provide computational tools to converge towards the optimal policy and value functions, offering a practical means to navigate the challenges posed by the non-linearity of the Bellman optimality equation

Policy Iteration is a key DP strategy aimed at discovering the optimal policy by directly manipulating the starting policy. However, before embarking on this process, a thorough evaluation of the current policy is essential. This evaluation follows an iterative procedure, as outlined in algorithm A.1, where the parameter defines the accuracy of the evaluation. A lower value indicates a more precise evaluation.

2.2 Evolution from Tabular Methods to Neural Networks in Reinforcement Learning

Reinforcement learning strategies designed for systems with well-defined states and actions often rely on lookup tables. In this paradigm, the state-value function V V and action-value function Q Q have entries for each state and state-action pair, respectively. However, this approach faces significant challenges when scaling up to large Markov Decision Processes (MDPs). Issues related to memory constraints, slow individual state learning, and the impracticality of linear lookup

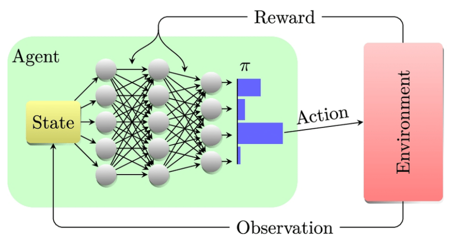


Figure 2.3: Scientific Diagram of Neural Network

in continuous action and state spaces become apparent.

.

In the contemporary landscape of research, neural networks have emerged as the most intuitive option for function approximation. Their widespread use is driven by their ability to reduce training time for high-dimensional systems and their efficient memory utilization. This integration serves as a crucial bridge between traditional reinforcement learning methods and recent advancements in deep learning theory. The enthusiasm surrounding deep learning over the last decade has established neural networks as fundamental tools for developing deep reinforcement learning (Deep RL), yielding remarkable results.

DeepMind's seminal papers [44] and [45] mark a pivotal step toward Deep RL and general artificial intelligence. These contributions demonstrate the broad applicability of AI across various environments. Given the focus of this work on model-free algorithms, the ensuing section explores the state-of-the-art theories underpinning the Deep RL framework. Additionally, it provides an overview of deep learning, culminating in the presentation of two deep actor-critic algorithms employed in the thesis experiments: Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC). This comprehensive exploration encapsulates the evolution from tabular methods to the pivotal role of neural networks in contemporary reinforcement learning landscapes.

2.2.1 Architectural Elegance in Convolutional Neural Networks

Sensory reception, a fundamental aspect of how humans and animals react to changes, involves the use of sensors that process input data and respond to specific stimuli. This concept serves as inspiration for the architecture of Convolutional Neural Networks (CNNs), designed to efficiently handle significant input data, particularly finding applications in computer vision.

A notable representation of CNN [2]architecture is LeNet-5, showcased in Figure 2.6 on the following page. LeNet-5 is adept at recognizing digits in images, making it a quintessential example of a standard convolutional neural network. This architecture typically comprises a sequence of convolutional layers followed by a subsampling pooling layer. In the convolutional stack's culmination, the values map into the final hidden layers of the network, ultimately computing the low-dimensional output. These final layers often consist of fully-connected layers.

The key strength of CNNs lies in their ability to hierarchically learn features from input data. It is conceivable that the initial layers focus on learning low-level features, such as edges and textures, from the input data. As one progresses through the network, the subsequent layers then combine these low-level features to form more abstract and high-level representations. This hierarchical feature learning makes CNNs particularly effective in tasks like image recognition.

The convolutional layers in a CNN perform the vital task of convolving filters or kernels over the input data. This operation helps detect local patterns, allowing the network to recognize features in various spatial locations. The subsampling pooling layer contributes to spatial invariance, enhancing the network's robustness to translations and distortions in the input data.

In summary, Convolutional Neural Networks draw inspiration from sensory reception systems, leveraging a hierarchical architecture to efficiently process sub-

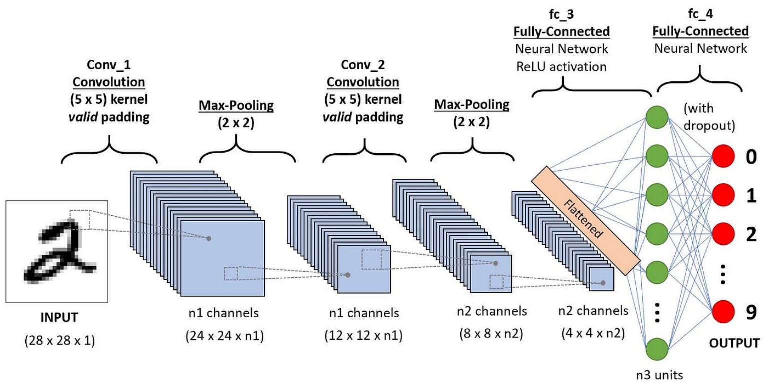


Figure 2.4: Schematic Diagram of Convolutional Neural Networks

stantial input data. The example of LeNet-5 demonstrates the standard structure of a CNN, emphasizing the role of convolutional and pooling layers in learning hierarchical features. This understanding of CNNs underscores their significance, especially in the realm of computer vision applications, where they excel in tasks such as image recognition

2.2.2 Actor Critic Architecture

The Actor-Critic architecture, illustrated in Figure 2.5 on the subsequent page, stands as the juncture where value-based approaches intersect with policy gradient methods. Essentially policy gradient methods and actor-critic methods utilize the value function to learn the parameters

of the policy. This approach involves two distinct components, the actor, and the critic, forming a symbiotic relationship in the pursuit of effective reinforcement learning.

In this tandem framework, the actor pertains to the policy, dictating the agent's actions in the environment. Conversely, the critic is responsible for estimating a value function, often in the form of a Q-value function. Deep reinforcement learning integrates neural networks as function approximators [43] to

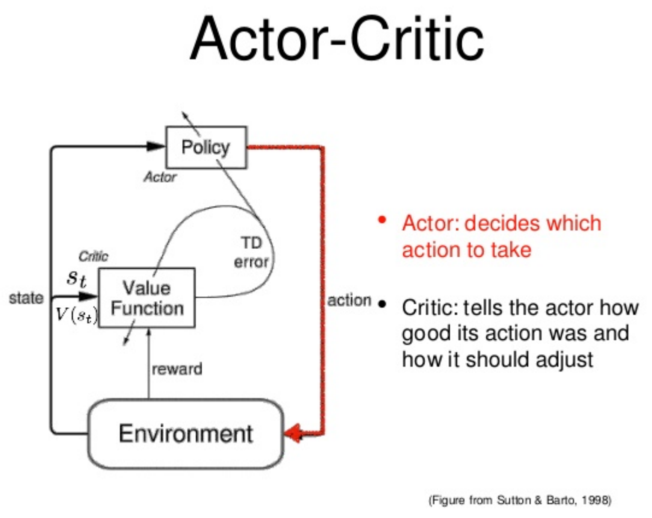

Actor-Critic
Policy
Actor
Critic
TD error
s_t
state
$V(s_t)$
Value Function
reward
action
Environment
Actor: decides which action to take
Critic: tells the actor how good its action was and how it should adjust
(Figure from Sutton & Barto, 1998)

Figure 2.5: Sample Diagram of Actor-Critic Architecture

represent both the actor and the critic. The actor leverages gradients derived from the policy gradient theorem to adjust policy parameters, while the critic estimates the approximate value function corresponding to the current policy .

A common strategy in Actor-Critic architectures is to update both networks using the Temporal Difference (TD) Error, as discussed in Section 2.1.5 on page 15. The critic's estimation plays a pivotal role in determining the contribution that expected values of the current and next state provide to the TD-error. Essentially, the output of the critic becomes instrumental in the update of the actor's parameters, fostering a dynamic interplay between policy improvement and value estimation.

The synergy between the actor and critic in the Actor-Critic architecture enhances the efficiency of reinforcement learning algorithms. The actor learns to improve decision-making policies by incorporating feedback from the critic, which, in turn, refines its estimates based on the observed performance of the actor. This dual-learning process contributes to the stability and effectiveness of the overall system, making Actor-Critic architectures a popular choice in the realm of deep reinforcement learning

2.3 Soft Actor Critic(SAC)

Soft Actor-Critic (SAC) innovatively combines the off-policy actor-critic setup with a stochastic policy, creating a link between stochastic policy optimization and DDPG-style approaches. This proves especially valuable in scenarios with continuous action spaces[6], showcasing SAC's model-free capabilities. Unlike DDPG, SAC addresses the challenges associated with stabilizing and tuning hyperparameters, providing a robust alternative.

DDPG's Achilles' heel lies in the interplay between the deterministic actor network[3] and the Q-function, resulting in instability and sensitivity to tuning. The learned Q-function tends to overestimate Q-values, leading to policy breakdown by exploiting errors in the Q-function. SAC mitigates this by adopting Clipped Double-Q Learning, a technique also employed by Twin Delayed DDPG (TD3). SAC employs two Q-functions, using the smaller Q-value to formulate targets in the Bellman error loss functions, enhancing stability.

Entropy regularization is another standout feature of SAC. The policy is trained to optimize a trade-off between expected return and entropy, a measure of policy randomness. This characteristic directly addresses the exploration-exploitation trade-off, where increased entropy facilitates more exploration, accelerating learning while preventing premature convergence to local optima.

In SAC, five neural networks come into play: the local stochastic policy network with parameter , two local Q-Networks with parameters 1 and 2, and two target Q-Networks with parameters 1 and 2. The behavior mirrors that of DDPG target networks, updating through the algorithm's specified equations. This en-

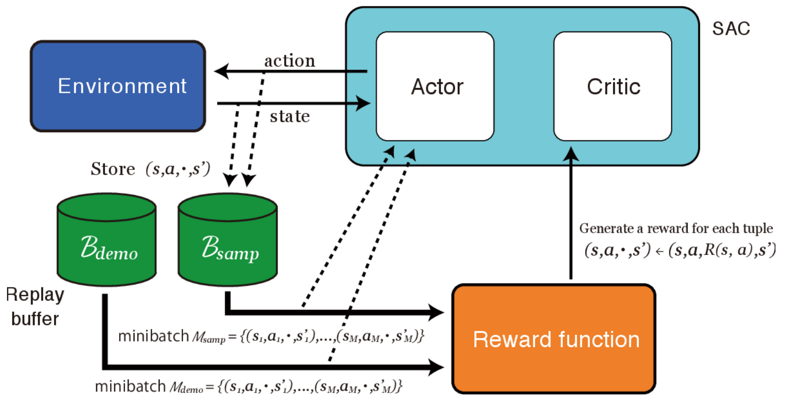

SAC
Environment
action
state
Actor
Critic
Store $(s,a,\cdot,s')$
$\mathcal{B}_{demo}$
$\mathcal{B}_{samp}$
Replay buffer
Generate a reward for each tuple
$(s,a,\cdot,s') \leftarrow (s,a,R(s,a),s')$
minibatch $M_{samp}=\{(s_1,a_1,\cdot,s'_1),\ldots,(s_M,a_M,\cdot,s'_M)\}$
Reward function
minibatch $M_{demo}=\{(s_1,a_1,\cdot,s'_1),\ldots,(s_M,a_M,\cdot,s'_M)\}$

Figure 2.6: Schematic Diagram of Soft Actor-Critic extensive Reward Function

semble of networks contributes to SAC's effectiveness in handling complex reinforcement learning tasks.

Entropy-regularized reinforcement learning introduces the concept of entropy, which quantifies the average rate at which a stochastic data source produces information. In simpler terms, entropy measures the randomness of a random variable. The formula for calculating the entropy (H) of a random variable x with probability mass or density function P is given by eq. (2.8):

$$H(P) = ExP[logP(x)] \tag{2.2}$$

This equation captures the essence of entropy as a measure of information content, emphasizing that low-probability events convey more information than high-probability ones.In the context of reinforcement learning (RL), entropy regularization modifies the standard RL objective by incorporating entropy[7]. The agent receives a bonus reward at each time step proportional to the entropy of the policy at that timestep.Entropy regularization aligns with the overarching goal of reinforcement learning to discover optimal policies in uncertain environments.

.

Chapter 3

Tools and Frameworks

This chapter delves into the pivotal tools and frameworks employed in the development of the thesis project, aimed at advancing the field of reinforcement learning. The journey begins with theOpen AI Gym toolkit[8], a fundamental component for developing and comparing reinforcement learning algorithms.Open AI Gym provides a standardized environment for testing and benchmarking various algorithms, allowing for a systematic evaluation of performance across different scenarios. Its role in the thesis project is critical, as it sets the stage for experimentation and analysis, providing a consistent platform to measure the efficacy of the developed algorithms.

Moving forward, the exploration delves into the PyTorch framework, a powerful deep learning library that has gained immense popularity in both research and industry. PyTorch's dynamic computational graph and intuitive interface make it an ideal choice for implementing complex neural network architectures, a necessity when dealing with reinforcement learning tasks. The chapter highlights the significance of PyTorch in enabling the seamless integration of neural networks into the project, fostering the development of sophisticated models that can learn and adapt in dynamic environments.

The narrative then extends to TensorFlow, another prominent deep learning framework that has played a pivotal role in shaping the landscape of artificial intelligence. TensorFlow's strengths lie in its scalability and flexibility, making it suitable for a wide range of applications. In the context of the thesis project, TensorFlow complements PyTorch by providing an alternative framework for experimentation and comparison. The chapter sheds light on the unique features of TensorFlow that contribute to the project's overarching goals.

As the exploration reaches its zenith, attention is directed towards PyTorch Networks, a specialized extension of PyTorch designed for reinforcement learning tasks. This framework goes beyond the standard capabilities of PyTorch, offering tailored functionalities that cater specifically to the nuances of reinforcement learning algorithms. The chapter underscores the importance of PyTorch Networks in fine-tuning models and optimizing their performance within the context of reinforcement learning challenges.

To gauge the effectiveness and efficiency of the developed algorithms, the Never2 Tool emerges as a key element in the concluding sections of the chapter. This measurement tool provides quantitative insights into the performance metrics of the reinforcement learning models, allowing for a meticulous evaluation of their learning capabilities and adaptability. The chapter culminates with a comprehensive discussion on the insights derived from the Never2 Tool, providing a holistic view of the impact and contributions made by the implemented tools and frameworks in advancing the field of reinforcement learning.

In essence, this chapter serves as a roadmap through the intricacies of the selected tools and frameworks, highlighting their individual significance and collective synergy in shaping the trajectory of the thesis project. Each component contributes to the overarching goal of advancing reinforcement learning, laying the groundwork for innovative solutions and pushing the boundaries of what is achievable in this dynamic and evolving field

3.1 Environments

Open AI Gym, introduced in 2016 during its public beta, has evolved into one of the most influential toolkits and frameworks in the realm of reinforcement learning. This discussion explores the motivations behind the creation ofOpen AI Gym, delving into the challenges within the reinforcement learning landscape and how the framework addresses them.

Reinforcement Learning Landscape and Challenges:

Reinforcement learning, a subset of machine learning, is dedicated to the study of decision-making and motor control. Researchers aim to understand how an agent can learn and improve to achieve specific goals in complex, often unknown environments. Its broad applicability, ranging from robotics to business decisions and financial trading, has made reinforcement learning an attractive area for both academia and industry.

However, the progress in reinforcement learning faced hurdles, primarily due to the absence of robust benchmarks. Unlike supervised learning, which flourished with datasets like ImageNet, reinforcement learning lacked equivalent standardized benchmarks. Additionally, the lack of standardization in the design of environments presented a challenge. Minor differences in problem definitions, reward functions, or action spaces could significantly impact the difficulty of tasks, impeding reproducibility and hindering the comparison of results across different studies.

Motivations forOpen AI Gym: The need to address these challenges was the driving force behind the development ofOpen AI Gym. The framework aimed to provide a solution to the dearth of benchmarks and the lack of standardization

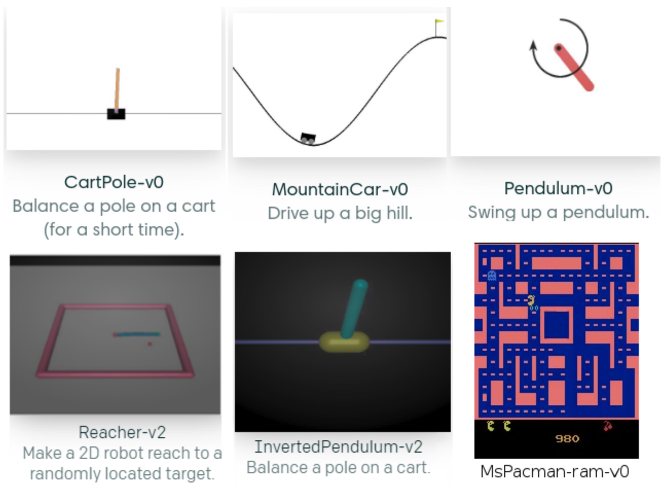


Figure 3.1: Open AI gym Environments

in reinforcement learning experiments. It envisioned a platform that would serve as a standardized interface for environments, allowing researchers and developers to focus on the core of reinforcement learning—agent design—without being constrained by predefined interfaces.

In reinforcement learning, the key components are the agent and the environment.Open AI Gym made a strategic choice to emphasize the abstraction of environments rather than agents. Instead of imposing pre-defined agent interfaces, the framework provides a standard environment interface. This decision empowers developers to design agents independently, fostering creativity and innovation in the core aspects of reinforcement learning.

The significance of this approach lies in the versatility it affords. Agents implemented withOpen AI Gym can seamlessly interact with any environment within the framework, promoting adaptability and ease of experimentation. Developers can tailor environments to suit specific experiments, enabling personalized testing scenarios that cater to the unique requirements of diverse research endeavors.

Open AI Gym encompasses various environments categorized into distinct types:

Algorithms: This category focuses on tasks involving the imitation of computations, such as copying or reversing symbols from an input tape. Task difficulty can be adjusted by varying the sequence length.

Atari: Emulating the Atari 2600 video games, this section of environments relies on the Arcade Learning Environment (ALE). It provides over 100 environments offering observations in the form of raw pixel images or RAM.

Box2D: Tasks in this group involve continuous control in a simple 2D simulator, featuring challenges like BipedalWalker, CarRacing, and LunarLander.

Classic Control: Derived from control theory, this class includes problems widely used in classic reinforcement learning literature. Examples include balancing a pole on a cart or swinging up a pendulum.

MuJoCo: This collection introduces continuous control tasks within a fast physics 3D simulator, known as MuJoCo. It serves as a valuable resource for research and development in robotics, biomechanics, graphics, and animation.

Robotics: Open AI Gym's Robotics category presents eight environments with more complex manipulation tasks than MuJoCo. Notable examples include Fetch, a robotic arm for object manipulation, and ShadowHand, a robotic hand for intricate object manipulation.

These environments collectively offer a rich and diverse set of challenges for reinforcement learning algorithms, spanning various domains and complexities

Interface functions

ExploringOpen AI Gym, it is essential to focus on the most crucial interface func- tions that the agent will exploit to interact with the environment. The functions which constitute the skeleton of anOpen AI Gym environment are the following:

• **def step(self, action)**: through this function, the agent can communi- cate the action it wants to take. The input data depends on the type and number of variables in the actions space (e.g. discrete or continuous). As will be discussed in section 3.1 on the following page, the values returned by this function represent the environment state after the manipulation caused by the agent action. Thanks to these data, the agent will be able to select the next action following the reinforcement learning loop.

• **def reset(self)**: during the episode, internal variables of the environment changes, influenced by the action taken previously. This function allows the agent to restart the initial situation of the environment. This procedure is particularly helpful when an episode finishes and the agent has to restart the next learning episode in a brand new copy of the environment.

• **def render**(self, mode='human', close=False): this function is mainly used in simulated environments. It enables the visual render (if available) of the environment.

• **def close(self)**: the final function to close the environment after the end of all experiments and episodes.

3.2 Pytorch

PyTorch, an open-source machine learning and deep learning library developed by Facebook's AI Research Lab, was released to the public in October 2016. It aims to provide an intuitive and straightforward framework for artificial intelligence projects, with a particular focus on computer vision and natural language processing.

Developed using Python, C++, and CUDA, PyTorch leverages CUDA-enabled GPUs for general-purpose processing. While the primary interface is in Python, there is also a C++ interface, showcasing the versatility of the library. The components of PyTorch include:

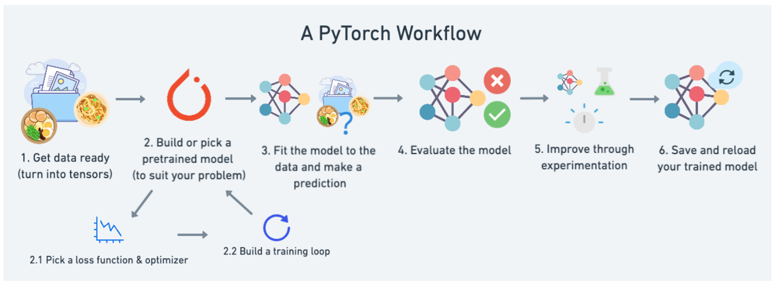

A PyTorch Workflow
1. Get data ready (turn into tensors)
2. Build or pick a pretrained model (to suit your problem)
2.1 Pick a loss function & optimizer
2.2 Build a training loop
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentation
6. Save and reload your trained model

Figure 3.2: Pytorch workflow

torch: A tensor library with robust GPU support, implementing interfaces similar to NumPy. It includes data structures for multi-dimensional tensors and mathematical operations, offering utilities for efficient serialization of tensors and arbitrary types.

torch.autograd: A tape-based automatic differentiation library supporting every differentiable operation on tensors available in torch.

torch.jit: A compilation stack that uses TorchScript to create serializable and optimizable models from PyTorch code. This allows training in PyTorch using Python and exporting the model to production environments where Python might be less advantageous for performance reasons.

torch.nn: A neural networks library compatible with autograd and designed for flexibility. torch.multiprocessing: Based on the Python multiprocessing library, it implements memory sharing of torch tensors across processes.

torch.utils: Contains utility functions to better exploit the features of PyTorch.

PyTorch provides a NumPy-like experience for interacting and manipulating data structures suitable for GPU computation, known as Tensors. Tensors can be used on both CPU and GPU, accelerating computations with functions explicitly designed for scientific computation needs.

Unlike frameworks that are primarily complex C++ bindings, PyTorch prioritizes Python, providing a natural user experience. The design emphasizes intuitiveness and linearity, making PyTorch synchronous for improved debugging experiences. Developers aimed to create a product that is easy to use, and this intention is reflected in the library's design.

One distinctive feature of PyTorch is its tape-based automatic differentiation, offering a single way to build neural networks. While other frameworks like TensorFlow or Theano utilize a static approach in graph creation, PyTorch employs Tape-Based Autograd. This approach, based on reverse-mode automatic differentiation, allows users to change the network structure dynamically without lag or overhead. It relies on the properties of the chain rule, making it possible to calculate derivatives efficiently.

PyTorch's commitment to providing a user-friendly, flexible, and efficient deep learning platform has made it a popular choice in the machine learning community. Its seamless integration with Python and emphasis on dynamic computation graph creation set it apart from other frameworks, contributing to its widespread adoption in both research and industrial applications

3.3 Tensorflow

TensorFlow is a powerful open-source machine learning library extensively used in reinforcement learning (RL), a paradigm where an agent learns by interacting with an environment to maximize cumulative rewards. TensorFlow's flexibility and efficiency make it well-suited for building and training neural networks, a key component in many RL algorithms.

In RL, TensorFlow is employed to define and train agents, which are typically neural network models. Using TensorFlow's high-level API, Keras, users can construct and optimize complex neural networks that represent an agent's policy or

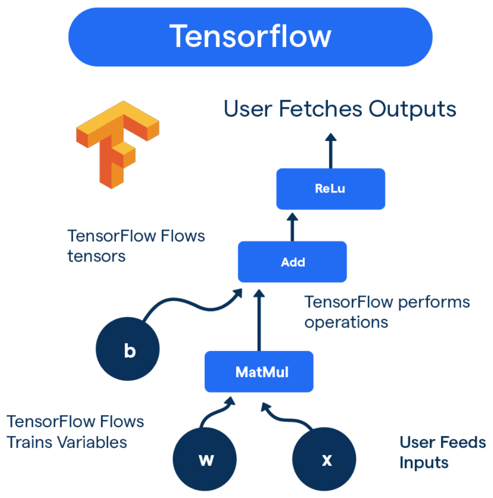


Figure 3.3: Tensor flow Key features

value function. The policy is a strategy guiding the agent's decisions based on observed states, and the value function estimates the expected cumulative reward for a given state-action pair. Training RL agents involves iterative interactions with the environment. TensorFlow facilitates this process by providing optimization algorithms and tools for efficient neural network training. Experience replay, a technique enhancing training stability, is also supported by TensorFlow.

Integration withOpen AI Gym, a popular RL toolkit, is seamless. TensorFlow users can leverageOpen AI Gym environments to simulate and test various RL algorithms. TensorBoard, a visualization tool integrated with TensorFlow, assists in monitoring training progress, evaluating performance, and diagnosing issues in real time.

A common RL example using TensorFlow is the implementation of a deep Q-network (DQN). TensorFlow's model-building capabilities, optimization algorithms, and gradient computation simplify the DQN training process. In a typical DQN implementation, a neural network approximates the Q-values, representing the expected cumulative rewards for state-action pairs. Training involves adjusting the network's parameters to minimize the difference between predicted and target Q-values.

TensorFlow's model-saving functionality enables users to store trained models, facilitating deployment for decision-making in real or simulated environments. This feature is crucial for practical applications where RL models transition from training to serving stages.

3.4 Comparision between Tensorflow and Pytorch

n the post-deep learning era, the development of neural network architectures and frameworks has become a focal point for many companies. Two prominent frameworks, TensorFlow by Google and PyTorch by Facebook , have emerged as leaders in this field. Despite serving the same purpose, implementing a neural network in these frameworks can yield different results due to the inherent distinctions in their training processes and underlying technologies.

One significant difference lies in the construction of computational graphs, an abstraction representing the computation process. TensorFlow adopts a static approach, defining computational graphs before code execution, allowing for parallelism and driving scheduling. This method communicates with the external world via tensors, which are later substituted by input data during runtime. In contrast, PyTorch employs a dynamic computational graph, constructed incrementally at runtime without placeholders. This flexibility supports on-the-fly changes to the computational graph, making PyTorch more adaptable and Pythonic.

Distributed training and data parallelism are crucial features, with PyTorch offering native support for asynchronous execution from Python, potentially improving performance. TensorFlow, while also supporting these capabilities, requires more developer effort to fine-tune computations for specific devices. Although both frameworks provide opportunities for distributed training, PyTorch requires less effort for seamless integration.

Visualization tools play a vital role in machine learning research, and here TensorFlow leverages TensorBoard, offering extensive features for visualizing and tracking the training process. PyTorch, on the other hand, relies on Visdom, developed by Facebook researchers, which provides minimalistic features compared to TensorBoard. However, TensorBoard can be used with PyTorch through the TensorBoardX library, bridging the gap in visualization capabilities.

In terms of production deployment, TensorFlow shines with TensorFlow Serving, a framework that facilitates REST Client API usage for deploying trained models. PyTorch has made strides in deployment, but it currently lacks a dedicated framework for web deployment. Developers must resort to Flask or Django as backend servers to create the necessary environment for model exploitation .

TensorFlow and PyTorch are popular open-source deep learning frameworks. TensorFlow is developed by Google and is known for its scalability, deployment capabilities, and extensive community support. PyTorch, developed by Facebook, is praised for its dynamic computation graph, simplicity, and intuitive debugging. TensorFlow's static graph facilitates optimization, while PyTorch's dynamic graph offers flexibility during model development. TensorFlow has strong integration with TensorFlow Lite for mobile and TensorFlow.js for web applications. PyTorch's eager execution simplifies debugging and experimentation. Both frameworks provide a rich ecosystem, but the choice often depends on personal preferences, project requirements, and the development style.

Chapter 4

Learning and Verification with NeVer2

NeVer2 stands as a versatile tool, combining a graphical user interface (GUI) with a command-line interface (CLI) to facilitate the seamless creation, training, and validation of neural networks. This innovative platform streamlines the complex process of managing neural networks within a unified environment, providing users with an integrated solution for their machine-learning endeavors. Moreover,[10] NeVer2 extends its functionality to encompass the verification of Very Deep Neural Network Library (VNN-LIB) properties on Open Neural Network Exchange (ONNX) models, offering a command-line interface for users who prefer a more streamlined and efficient approach.

One of NeVer2's distinguishing features lies in its commitment to user-friendly implementation. The tool is designed to be accessible to users at varying levels of expertise, ensuring that the complexities of deep learning are made more manageable. This commitment is exemplified through the incorporation of the pyNeVer API, providing a solid and extensible framework that underpins the tool's seamless functionality.

NeVer2's proficiency extends across the entire lifecycle of neural network development, emphasizing a holistic approach to model refinement. Its support for dynamic computational graphs, reminiscent of the PyTorch philosophy, is a key feature that sets it apart. This dynamic nature empowers users to make real-time adjustments to the computational graph, offering unparalleled flexibility during runtime. For projects requiring on-the-fly modifications to the neural network architecture, NeVer2 emerges as a flexible and adaptive solution.

The tool's prowess is further showcased in its adept handling of distributed training, a critical component of large-scale deep learning endeavors. NeVer2 leverages PyTorch's capabilities to streamline the complexities associated with asynchronous execution from Python. This capability not only enhances performance but also ensures scalability in scenarios where significant computational resources are required. The seamless integration of distributed training capabilities underscores NeVer2's commitment to providing users with a comprehensive and efficient neural network development environment.

In the realm of neural network verification, NeVer2 excels by providing a command-line interface (CLI) for validating Very Deep Neural Network Library (VNN-LIB) properties on Open Neural Network Exchange (ONNX) models. This versatile CLI allows users to execute verification procedures outlined in Satisfiability Modulo Theories (SMT) files on specified neural networks in the ONNX format. This dual-interface approach caters to diverse user preferences, allowing for both graphical and command-line interactions, depending on the user's workflow and requirements.

The heart of NeVer2's verification capabilities lies in its diverse range of strategies. Users can opt for the 'complete' strategy, leveraging the exact algorithm suitable for small-sized networks. Alternatively, the 'approximate' strategy employs an over-approximate algorithm, balancing accuracy and computational efficiency. The 'mixed1' and 'mixed2' strategies introduce a nuanced approach, refining one or two neurons per layer, respectively. This adaptability ensures that users can tailor the verification process to the specific characteristics and requirements of their neural network, striking an optimal balance between precision and computational resources.

NeVer2 stands as a comprehensive and adaptable tool that seamlessly integrates into the Python ecosystem. Its user-friendly design, support for dynamic computational graphs, and adept handling of distributed training make it a valuable asset for both researchers and practitioners in the field of deep learning. Whether through its graphical user interface or command-line capabilities, NeVer2 caters to diverse user preferences, ensuring a flexible and efficient neural network development environment. As the landscape of machine learning continues to evolve, NeVer2 remains at the forefront, empowering users with the tools they need to navigate the complexities of neural network development with confidence and ease.

NeVer2 emerges as a robust and flexible tool that addresses various facets of neural network development and verification. Its integration with the pyNeVer API, dynamic computational graph support, and emphasis on verification techniques position it as a valuable asset for researchers and practitioners. The tool's commitment to user-friendly implementation, distributed training support, and visualization capabilities contributes to its versatility

4.1 Installation

The installation process for NeVer2 involves setting up the required packages and dependencies to enable seamless operation of the neural network tool. NeVer2 relies on two main components: the pyNeVer API and the PyQt6 framework.

These can be easily installed using the Python package manager, PIP.

To initiate the installation, execute the following command in your terminal or command prompt:

```
pip install pynever PyQt6
```

This command fetches and installs the pyNeVer API and PyQt6 framework, ensuring that NeVer2 has the essential components to function properly.Once the packages are successfully installed, NeVer2 can be launched from the root directory using the following command:

```
python NeVer2/never2.py
```

This command activates NeVer2, allowing users to access the graphical user interface (GUI) and leverage its capabilities for neural network development, training, and verification.

ARM BASED MAC OS

For users on ARM-based Mac OS, additional considerations come into play due to compatibility issues with the default Python distribution for ARM platforms. To address this, it is recommended to install miniforge for arm64 (Apple Silicon), a distribution that is compatible with the architecture.Creating a Python virtual environment is the next step in the ARM-based Mac OS installation process. This involves using Conda, a package manager, to set up a specific environment named 'myenv' with Python version 3.9.5:

```
conda create -n myenv python=3.9.5
conda activate myenv
```

This ensures that the subsequent installations are specific to the newly created environment. The next step involves installing additional dependencies for TensorFlow using Conda:

```
conda install -c apple tensorflow-deps
pip install tensorflow-macos tensorflow-metal
pip install pynever PyQt6
```

Now, NeVer2 is ready to be executed on an ARM-based Mac OS with the following command

```
python NeVer2/never2.py
```

It's important to note that each time NeVer2 is to be run, the Conda environment must be activated using:

```
conda activate myenv
```

This ensures that NeVer2 operates within the specific environment with the correct dependencies, providing a smooth and consistent experience for users on ARM-based Mac OS. In summary, the installation process for NeVer2 involves acquiring the necessary packages, addressing compatibility concerns on specific platforms, and setting up an isolated environment to ensure a reliable and efficient operation of the neural network tool

4.2 Software Architecture

The Never2 Tool design is depicted in Figure 1 using a UML class diagram, focusing on its fundamental structure rather than a comprehensive Software Architecture overview. The internal representation of neural networks is managed by pyNeVer, a [11]Python API offering learning and verification capabilities. The core component is the Project class, encapsulating functionalities for network design, interaction with pyNeVer, and procedures for opening and saving neural network files.

For the graphical interface, PyQt's Graphics View framework is employed. This framework facilitates the creation and interaction with 2D graphical items, supporting zooming and rotation. The event propagation architecture and Binary Space Partitioning (BSP) tree enable real-time visualization of large scenes. The GraphicsScene class is used for creating or destroying objects and setting global parameters, while the Scene class serves as a container for all application objects.

Concrete instances of the abstract class Block are displayed in the scene. The LayerBlock represents network layers, the FunctionalBlock defines input and output, and the PropertyBlock represents VNN-LIB properties. The Block classes are designed to support multiple inputs and outputs, allowing future extensions to accommodate architectures beyond feed-forward neural networks, such as ResNets and recurrent neural networks.

Leveraging pyNeVer, the tool enables direct import of neural network models in ONNX or PyTorch formats for visualization, property addition, or conversion. The generic interface of pyNeVer facilitates the creation of custom model conversion for various file formats, expanding the tool's capabilities and supporting additional benchmarks. The flexibility and extensibility of Never2 are highlighted, emphasizing its adaptability to evolving neural network architectures and file formats

4.3 Procedure

In NeVer2, building a neural network for the ACC application involves creating layers and defining input/output using the LayerBlock and Functional Block classes. The network is exported to PyTorch for training. Upon completion, it's re-imported into Never2, demonstrating the tool's bidirectional capability. This allows users to seamlessly transition between NeVer2 and PyTorch for design and

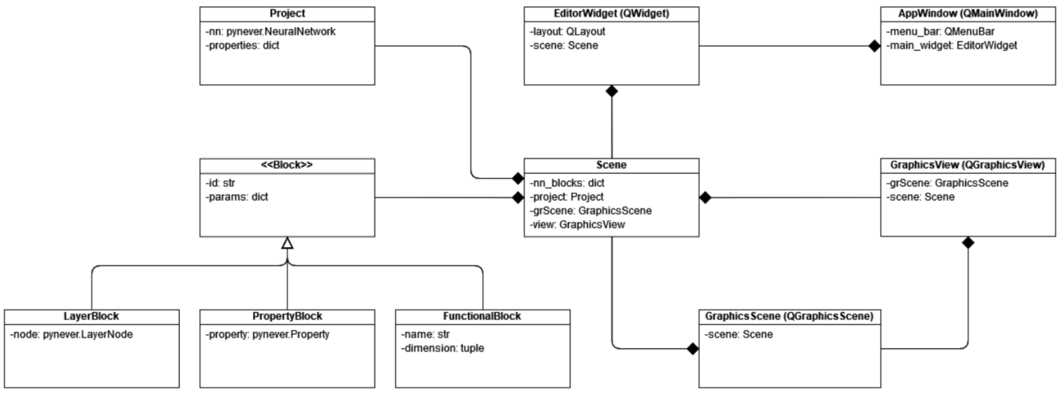
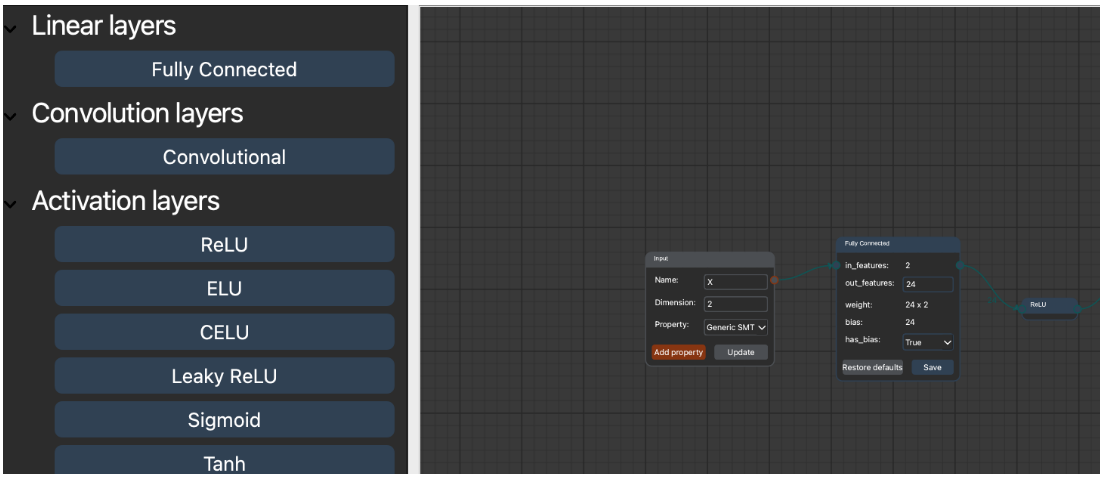


Figure 4.1: UML Diagram of Never2 Tool

training, reinforcing the tool's versatility in supporting diverse workflows.

4.3.1 Building the model

The initial screen of Never2 showcases two Functional Blocks, enabling users to define network input and corresponding labels. Sequentially, the first fully connected layer with 24 neurons, followed by a ReLU activation, is defined within the tool. To maintain order, each layer is added and updated individually, incorporating the input block to specify dimensions. The Save button on each block facilitates parameter updates, ensuring configuration accuracy. For user convenience, the Restore defaults option resets values to default settings without overwriting, enhancing flexibility in experimentation. This side-by-side interface design streamlines the process of constructing neural networks within NeVer2, providing a user-friendly experience. The automatic sequential addition of layers aligns with default settings, allowing users to progressively build and refine their network architecture. This step-by-step approach, coupled with intuitive controls, exemplifies NeVer2 commitment to simplicity and precision in network design for applications like the ACC scenario.



Input
Name: X
Dimension: 2
Property: Generic SMT

Fully Connected
in_features: 2
out_features: 24
weight: 24 x 2
bias: 24
has_bias: True

Figure 4.2: Interface of Never2 Tool after adding defining Properties

Activation layers(ReLU)

Rectified Linear Unit, or ReLU, is a widely used activation function in artificial neural networks, playing a crucial role in introducing non-linearity to the model's decision-making process. It operates on an input x, producing an output f(x) defined as the maximum of zero and the input itself, expressed as f(x) = max(0, x).

One of the primary reasons for ReLU's[12] popularity is its simplicity and efficiency. By allowing positive inputs to pass through unchanged and setting negative inputs to zero, ReLU aids in the model's ability to learn complex patterns in the data. This simplicity also contributes to faster convergence during the training process, as the linearity of the function eases gradient computation.

ReLU effectively addresses the vanishing gradient problem encountered in traditional activation functions like sigmoid or hyperbolic tangent. During backpropagation, gradients can diminish as they are propagated through layers, hindering the learning process. ReLU's derivative is either zero or one, preventing the vanishing gradient issue and facilitating more effective learning in deep networks.

However, ReLU is not without challenges. One notable issue is the "dying

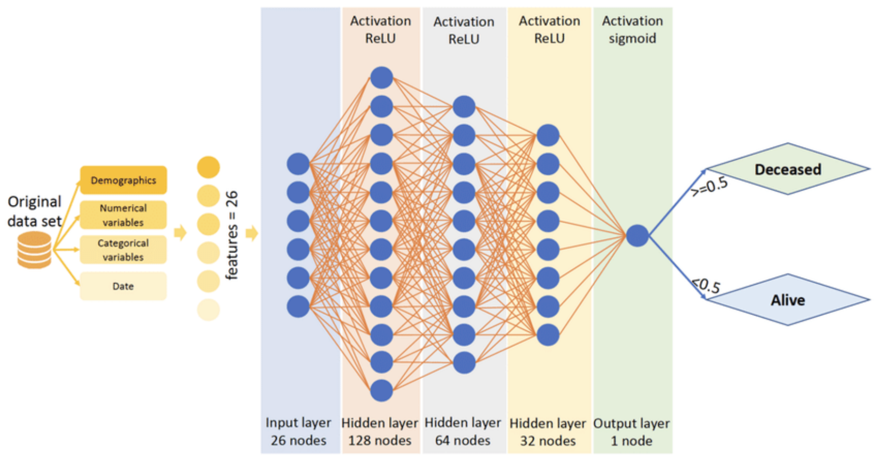


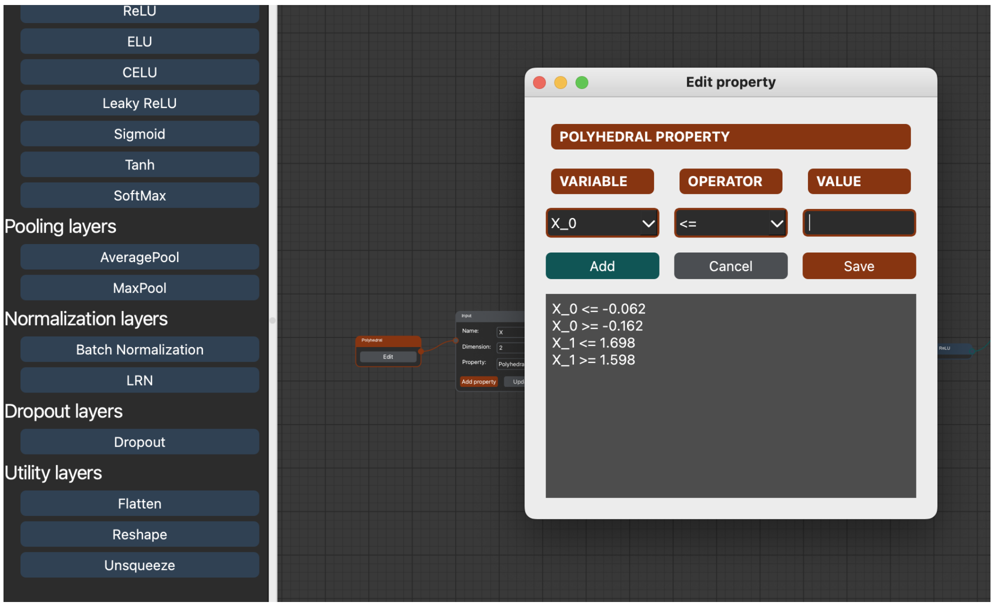
Figure 4.3: ReLU Neural network architecture

ReLU" problem, where neurons can become inactive during training and cease to update their weights. If a large gradient flows through a ReLU unit, it can cause the weights to update in such a way that the unit will always output zero. This can limit the model's capacity to learn and adapt.

To address the dying ReLU problem, variations like Leaky ReLU have been introduced. Leaky ReLU allows a small, non-zero gradient when the input is negative, ensuring that even neurons with negative inputs can contribute to the learning process.ReLU is a foundational activation function in deep learning, offering simplicity, efficient computation, and mitigation of the vanishing gradient problem. While it has challenges like the dying ReLU problem, researchers have introduced variations to enhance its performance and adaptability in training deep neural networks.

4.3.2 Defining the Property

In the finalization phase of configuring a neural network, a pivotal step involves defining VNN-LIB properties related to both input and output. Drawing guidance from the description provided in [8], the emphasis is on bounding input



VARIABLE: X_0
OPERATOR: <=
VALUE:

```
X_0 <= -0.062
X_0 >= -0.162
X_1 <= 1.698
X_1 >= 1.598
```

Figure 4.4: sample Diagram of Defining Polyhedral Property

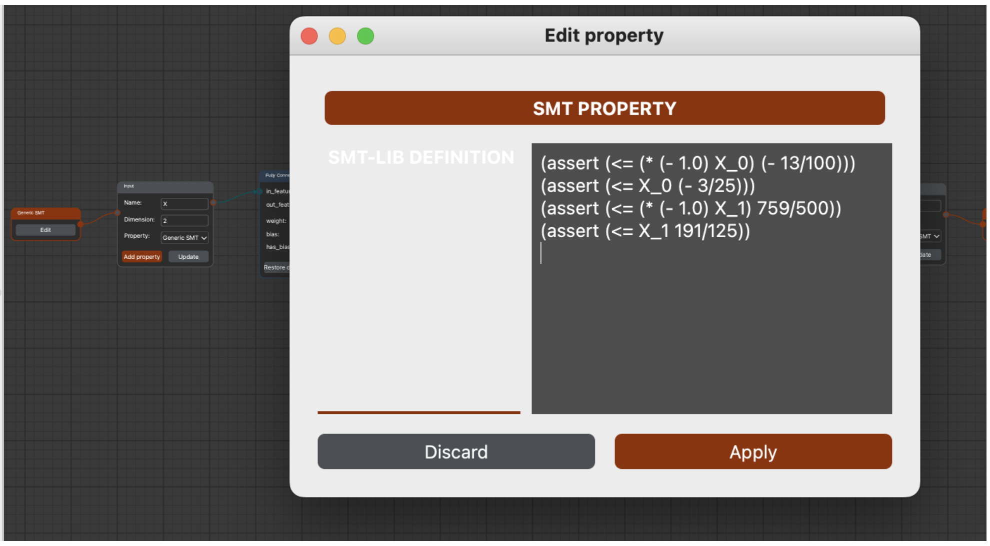
variables. Within this context, the OutBounds description recommends two distinctive approaches: Polyhedral Properties and Generic SMT Properties.

Polyhedral Property:

provides an insightful look into the interface where a Polyhedral Property is defined. This approach leverages the property selector in the input block, offering a controlled and efficient environment for bounding variables without the need for manually crafting SMT (Satisfiability Modulo Theory) statements. The Polyhedral Property method streamlines the process, allowing users to set constraints on input variables within a well-defined and controlled framework. As depicted in the figure, this property configuration aligns with the values outlined in Section 3.2, ensuring a harmonious integration of input bounding mechanisms.

SMT Property:

Conversely, the Generic SMT Properties approach involves the direct crafting of SMT expressions. While providing more flexibility, this method demands a deeper understanding of SMT syntax. Unlike the automated nature of Polyhe-



```
(assert (<= (* (- 1.0) X_0) (- 13/100)))
(assert (<= X_0 (- 3/25)))
(assert (<= (* (- 1.0) X_1) 759/500))
(assert (<= X_1 191/125))
```

Figure 4.5: Sample Diagram of Defining smt property

dral Properties, SMT Properties require explicit SMT statements for bounding variables. This approach allows for a more intricate and customized specification of constraints, affording users finer control over the bounding configurations.

In the Polyhedral Property method, the emphasis is on a user-friendly interface, minimizing the need for users to delve into the complexities of SMT expressions directly. Instead, the property selector in the input block becomes the conduit for bounding variable constraints. This approach simplifies the user experience while maintaining robust bounding mechanisms for input variables.

4.3.3 Load and Save Models

NeVer2 simplifies the integration of neural networks into its framework by supporting direct loading of models in ONNX and PyTorch formats. It also streamlines the linking of properties to networks when they share identical input and output identifiers, enhancing its utility for VNNCOMP benchmark creation.To generate a VNNCOMP-compatible benchmark using NeVer2, users import a neural network model in ONNX or PyTorch format, ensuring compatibility. Subsequently, users link a property to the imported network, verifying alignment in input and output identifiers.

Once the neural network and its corresponding property are integrated in NeVer2, users create benchmark files easily. In the menu, selecting "Save as..." with the VNN-LIB entry results in two distinct files: a .onnx file representing the neural network model and a .smt2 file encapsulating the defined property. This structured process streamlines benchmark creation, ensuring compatibility with VNNCOMP standards. NeVer2, with its capacity to seamlessly link properties to networks and export in standardized file formats, serves as a versatile tool for researchers participating in VNNCOMP or working within the VNN-LIB framework. Its commitment to simplifying the bench-marking process underscores NeVer2's significance in the realm of neural network verification.

4.3.4 Command-line interface

NeVer2 Tool offers command-line functionality through the options check model and -convert model, extending its capabilities beyond the graphical interface. These command-line features empower users to efficiently incorporate NeVer2's functionalities into automated workflows or scripts. The -check option facilitates the quick validation of an ONNX model's compliance with the VNN-LIB standard. Users can easily determine whether a given ONNX model adheres to the specifications outlined by VNN-LIB, streamlining the validation process through the command line.

Similarly, the -convert option enables the transformation of PyTorch models to the ONNX[13] format using NeVer2, contingent upon the compatibility of operators with the VNN-LIB standard. This command-line functionality ensures a seamless conversion process, provided the necessary operators are supported. These command-line options enhance NeVer2 Tool's versatility, allowing users to integrate its features into automated processes. Whether validating compliance or converting between model formats, the command-line interface provides a flexible means of leveraging NeVer2 Tool's capabilities for neural network analysis and verification in a programmatic fashion.

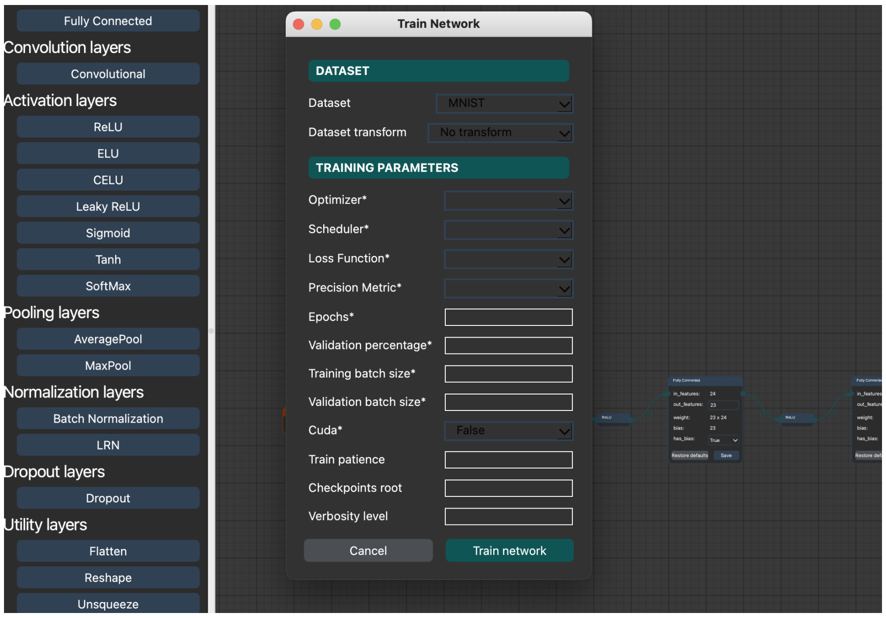
```
python pynever.py complete single -s /Users/surendrakumarreddypolaka
/Desktop/Thesis/NeVer2-main/dqn_network.onnx > /Users
/surendrakumarreddypolaka/Desktop/Thesis/NeVer2-main/output1.smt2
```

4.4 Training the Network

Upon completing the network construction, the next step involves training. Navigate to the menu bar and select "Learn..." -¿ "Train" to access the training window. In this interface, users can choose the dataset and configure various learning parameters.

NeVer2 provides default access to both MNIST and fMNIST datasets due to their widespread popularity. If these datasets are not already present, they will be downloaded and stored in the NeVer2 working directory upon the first selection. The inclusion of such widely used datasets ensures user convenience and accessibility.

For the training process, NeVer2 [14]offers a pre-defined dataset transform tailored for both convolutional and linear MNIST and fMNIST networks. This transformation consists of 2 or 3 steps: pilToTensor and Normalize(1, 0.5) are common to both cases, and an additional Flatten transform is included exclusively for the linear MNIST network. These transformations are crucial for preparing the data in a format suitable for training the neural network.



Dataset: MNIST
Dataset transform: No transform
Cuda*: False

Figure 4.6: sample diagram of parameters how to train the Network

In summary, NeVer2 simplifies the training process by providing a user-friendly interface for dataset selection and learning parameter configuration. The availability of default datasets, along with pre-defined transformations optimized for various network architectures, enhances the efficiency and accessibility of the training workflow within the NeVer2 tool.For efficient training in NeVer2, users can fine-tune learning parameters through a user-friendly interface:

Optimizer: Select the "Adam" optimizer, an acclaimed gradient-based optimization algorithm known for its effectiveness. Users can further customize related parameters once the optimizer is chosen.

Learning Rate Scheduler: Currently, NeVer2 supports the "ReduceLROnPlateau" learning rate scheduler. This scheduler adjusts the learning rate when a plateau in model performance is detected, providing adaptability during training.

Loss Function: Choose between "Cross Entropy" and "MSE Loss" based on the neural network's structure. This selection influences the calculation of the error during training.

Precision Metric: Select either "Inaccuracy" or "MSE Loss" to define the precision metric, guiding the evaluation of the model's performance during training.

Epochs: Define the number of training epochs, representing the iterations through the entire dataset during training.

Validation Percentage: Set a value between 0 and 1 to indicate the percentage of the dataset allocated for validation purposes.

Training and Validation Batch Size: Define the dimensions of training and validation batches, influencing the granularity of data processed during each iteration.

CUDA: Enable this option to leverage NVidia GPU architecture for accelerated computation, optimizing training speed.

Train Patience (Optional): Specify the number of epochs with no loss decrease before triggering early stopping in the training process.

Checkpoints Root (Optional): Designate the directory for storing training strategy checkpoints. The default location is the NeVer2 working directory.

Verbosity Level (Optional): Set the frequency of log prints during training, controlling the level of detail displayed. The default is set to print logs after each training batch.

These configurable parameters empower users to tailor the training process to their specific needs, balancing customization and user-friendly design within the NeVer2 environment.But in this Thesis we mainly focus on Verification tool rather than training the Network.

4.5 Verification Strategy

NeVer2 is a versatile tool designed to simplify the neural network development process. It features a user-friendly GUI that streamlines building, training, and verifying neural networks within a unified environment. The graphical interface offers an intuitive platform for users to navigate the complexities of machine learning seamlessly.

In addition to the GUI, NeVer2 caters to advanced users by providing a Command-Line Interface (CLI). This allows for efficient verification of VNN-LIB properties on ONNX models without the need for the graphical interface. Users can execute commands like "python NeVer2/never2.py" to verify specific properties specified in SMT files on ONNX networks. This CLI functionality enhances flexibility, enabling users to interact with NeVer2 in a more streamlined manner, especially when dealing with specific verification tasks.

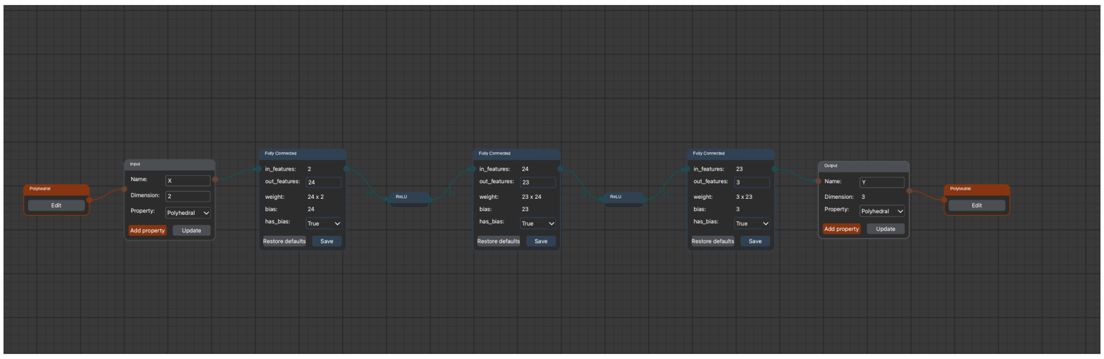
```
python NeVer2/never2.py -verify <property>.smt2
<network>.onnx [complete | approximate | mixed1 | mixed2]
```

verification procedure for the property specified in the SMT file on the network specified in the ONNX file. The verification strategy is one among the following:

- complete: uses the exact algorithm (for small-sized networks)
- approximate: uses the over-approximate algorithm
- mixed1: uses the mixed algorithm refining 1 neuron per layer
- mixed2: uses the mixed algorithm refining 2 neurons per layer

4.6 Output Visualization

In NeVer2, extending the neural network involves selecting corresponding blocks from the left toolbar. Adding ReLU activation functions is straightforward, re-



Name: X
Dimension: 2
Property: Polyhedral

in_features: 2
out_features: 24
weight: 24 x 2
bias: 24
has_bias: True

in_features: 24
out_features: 23
weight: 23 x 24
bias: 23
has_bias: True

in_features: 23
out_features: 3
weight: 3 x 23
bias: 3
has_bias: True

Name: Y
Dimension: 3
Property: Polyhedral

Figure 4.7: Output Visualization to verify the Network

quiring no additional parameters. Fully Connected layers can be seamlessly incorporated directly. The presented example illustrates the construction of a complete network, showcasing the simplicity of layer addition within NeVer2.

Seamless Layer Addition and Output Visualization in Neural Network Construction. Explore the simplicity of integrating ReLU activations and Fully Connected layers. Witness the clear computation of the final output, enhancing user understanding. NeVer2 provides an intuitive interface for efficient and visual network design

In this example, particular emphasis is given to the final layer, which performs the computation yielding the network's output. The output is visually presented in the output block, offering users a comprehensive view of the entire network's architecture and its functional progression.[15] NeVer2's intuitive interface facilitates a user-friendly experience, allowing for a clear visualization of the neural network's structure. This streamlined process enhances users' ability to comprehend the intricacies of each layer, fostering a deeper understanding of how the network computes the final output. As a result, NeVer2 provides a powerful tool for both novice and experienced users to efficiently design and analyze neural networks within a visually accessible environment

Chapter 5

Case Study

The experimental results section delves into the application of classic control reinforcement learning algorithms on two widely recognized problems, Mountain Car and Pendulum. These problems serve as benchmarks to assess the performance of three distinct types of algorithms: Basic Q-learning, Q-learning with a neural network, and Q-learning with a neural network implemented using the PyTorch framework. The emphasis is on measuring the output of the system through various metrics and employing the Never2 Tool to verify the network's satisfaction of specified conditions.

The first algorithm, Basic Q-learning[16], operates on a tabular Q-value approach. In this traditional method, the agent maintains a table to store Q-values for different state-action pairs. The algorithm iteratively updates these values based on rewards received, aiming to learn an optimal policy. The primary advantage of Basic Q-learning lies in its simplicity and ease of implementation, making it an essential baseline for comparison.

The second approach involves enhancing Q-learning with a neural network. Here, the Q-values are represented by the output of a neural network, allowing for a more flexible and scalable approximation of the optimal policy. By utilizing neural networks, the algorithm can handle high-dimensional state spaces more efficiently, overcoming limitations associated with tabular methods. The implementation of this enhanced Q-learning involves training a neural network to approximate the Q-values, with the network learning to generalize across different states.

PyTorch, a popular deep learning framework, is chosen for the neural network implementation. PyTorch's dynamic computation graph and extensive library of operations make it well-suited for reinforcement learning tasks. The integration of PyTorch into the Q-learning algorithm adds a layer of sophistication, facilitating the use of neural networks for more complex control problems.

To evaluate the performance of these algorithms, experiments are conducted on two classic control problems: Mountain Car and Pendulum. These environments provide challenging scenarios that test the adaptability and learning capabilities of the implemented algorithms. Metrics such as convergence speed, exploration efficiency, and overall task completion are measured to quantify the success of each algorithm in solving the control problems.

In addition to these metrics, the section introduces the Never2 Tool, a verification tool designed to assess whether the neural network satisfies specified conditions. Verification is crucial in reinforcement learning applications, especially in safety-critical environments. The Never2 Tool provides an additional layer of analysis, allowing researchers to validate the correctness and robustness of the learned policies.

The integration of Never2 into the experimental framework offers a comprehensive evaluation process. This tool verifies whether the learned policies adhere to predefined constraints, ensuring that the neural network's outputs align with safety and performance specifications. This step is particularly important when deploying reinforcement learning models in real-world applications where safety and reliability are paramount.

The experimental results section provides a thorough investigation into the performance of classic control reinforcement learning algorithms on Mountain Car and Pendulum problems. By comparing Basic Q-learning, Q-learning with a neural network, and PyTorch-based Q-learning, researchers gain insights into the strengths and limitations of each approach. The inclusion of the Never2 Tool further enhances the experimental framework, offering a robust means of verifying the network's adherence to specified conditions. This comprehensive analysis contributes to the broader understanding of reinforcement learning techniques in classic control scenarios and reinforces the importance of verification in real-world applications.

5.1 Software and Hardware Details

To conduct experiments leveraging the aforementioned specifications, a MacBook Air serves as the primary computing platform. The MacOS operating system provides a stable and user-friendly environment, fostering a seamless integration with various software tools. The development environment is established through Anaconda, offering a comprehensive suite of Python libraries and facilitating streamlined code development, analysis, and visualization.

For machine learning endeavors, the MacBook Air is equipped with Python 3.9, PyTorch 1.10.0, and TensorFlow 2.10.0. These frameworks empower researchers and developers to implement and train sophisticated machine learning models, while Open AIAI Gym version 0.21.0 serves as a fundamental toolkit for the development and evaluation of reinforcement learning algorithms

Component	Version/Details
Laptop	Macbook Air
Operating system	MacOS
Development Environment	Ananconda
Open AI AI Gym	v.0.21.0
Python	v.3.9
Pytorch	v.2.10.0
Tensorflow	v.2.10.0

Table 5.1: Development Machine specifications

5.2 Classic control Environment

The "MountainCar-v0" environment is a classic reinforcement learning problem included in the Open AIAI Gym toolkit. It represents a simplified two-dimensional physics simulation in which an underpowered car is tasked with reaching a flag located at the top of a hill.

Key Features of the Environment:

State Space: The state of the environment is defined by a two-dimensional vector representing the car's position and velocity. The position ranges from -1.2 to 0.6, indicating the car's location along the x-axis. The velocity ranges from -0.07 to 0.07, representing the car's speed.

Action Space: The car has three possible discrete actions: accelerate to the left, decelerate, or accelerate to the right. Actions are discrete and not continuous, providing a limited set of choices for the agent.

Rewards: The agent receives a reward of -1 for each time step until it reaches the flag at the top of the hill. The goal is to reach the flag with the minimal number of time steps.

Termination Condition: The environment terminates when the car reaches the flag at the top of the hill or when a predefined maximum number of time steps is reached. Constraints: The car is underpowered, making it unable to reach the flag directly. Thus, the agent must learn a strategy to build enough momentum by moving back and forth.

Difficulty: The challenge lies in mastering the timing and coordination of actions to propel the car up the hill efficiently.

This environment is commonly used to test and develop reinforcement learning algorithms, particularly those based on value iteration or policy gradients. Agents learn to navigate the trade-off between short-term negative rewards and the long-term goal of reaching the flag, showcasing the ability to solve problems with sparse and delayed rewards. The "MountainCar-v0" environment is a useful benchmark for understanding exploration-exploitation trade-offs and learning to solve complex tasks in reinforcement learning.

5.2.1 Experiments

This section encompasses a comprehensive evaluation of various reinforcement learning approaches applied to the task, including Verified Basic, the Q-learning Method[17], the Q-learning Neural Network Method, and the PyTorch Method. Each method undergoes rigorous verification to ensure accuracy and reliability in the experimental setup.

The Verified Basic approach serves as a foundational benchmark, establishing a baseline for comparison. Q-learning, a classical reinforcement learning technique, is then employed to iteratively optimize the agent's decision-making strategy based on the observed rewards in the environment. The Q-learning Neural Network Method introduces a neural network to approximate the Q-function, allowing for more complex representations and improved generalization.

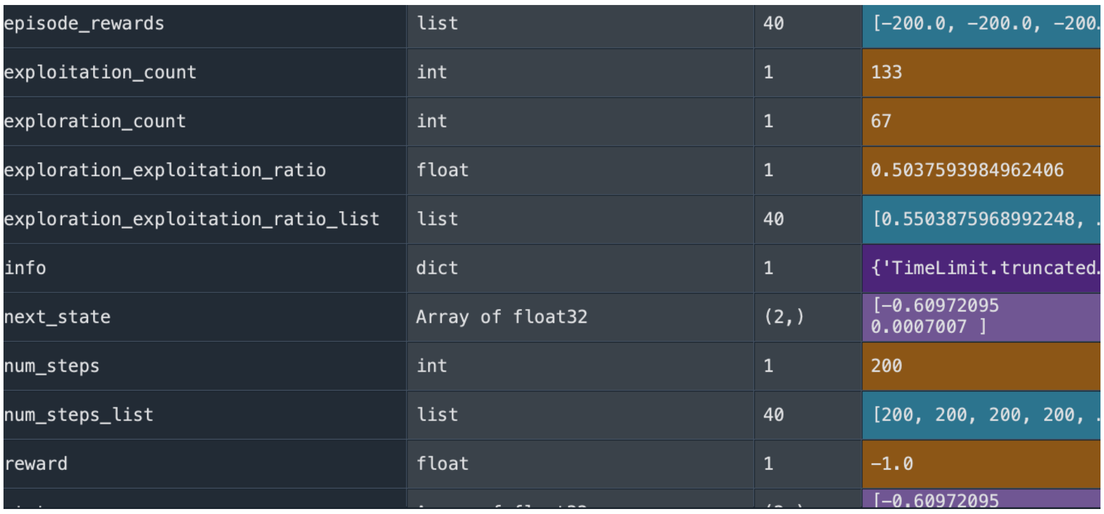
Furthermore, the PyTorch Method incorporates the PyTorch deep learning framework, leveraging its capabilities for building and training neural networks efficiently. Each method is systematically measured to verify the efficacy of the proposed analysis and the network generated. The evaluation criteria include factors such as convergence speed, learning stability, and overall performance in reaching the predefined goal within the MountainCar-v0 environment. This multifaceted analysis aims to provide insights into the strengths and limitations of each method, contributing to a nuanced understanding of their effectiveness in solving complex reinforcement learning problems

5.2.2 Basic Method

The presented code is an implementation of a reinforcement learning problem using the Open AI Gym library, focusing on the MountainCar-v0 environment—a classic challenge in the realm of reinforcement learning. Reinforcement learning involves an agent interacting with an environment, learning to take actions that lead to maximum cumulative reward over time. The main loop of the code is executed for a total of 40 episodes, a common practice to observe the agent's learning behavior across multiple instances.

Within each episode, the environment is reset, initiating a new trial with an initial state. The subsequent while loop runs until the episode is deemed complete, indicated by the ”done” variable. During each iteration of the while loop, the current state is rendered, providing a visual representation of the agent's interaction with the environment. The agent selects a random action from the action space using the ”.sample()” method, a simplistic exploration strategy. The environment is then queried for the next state, the reward obtained in the current step, the termination signal (”done”), and additional information.

The total reward for the episode is updated by aggregating the rewards obtained in each step. This cumulative reward metric is a key indicator of the agent's performance. The loop concludes by updating the current state to the next state, preparing for the subsequent iteration.



episode_rewards	list	40	[-200.0, -200.0, -200.
exploitation_count	int	1	133
exploration_count	int	1	67
exploration_exploitation_ratio	float	1	0.5037593984962406
exploration_exploitation_ratio_list	list	40	[0.5503875968992248, .
info	dict	1	{'TimeLimit.truncated.
next_state	Array of float32	(2,)	[-0.60972095 0.0007007]
num_steps	int	1	200
num_steps_list	list	40	[200, 200, 200, 200, .
reward	float	1	-1.0
			[-0.60972095

Figure 5.1: Basic Method output Results

This code serves as a foundational example, illustrating how to interact with the MountainCar-v0 environment in Open AI Gym and implementing a basic random policy for decision-making. It lays the groundwork for more advanced exploration-exploitation strategies, reinforcement learning algorithms, and policy optimization techniques that can be integrated to enhance the agent's problem-solving capabilities. Understanding and extending such code forms the basis for delving into the fascinating field of reinforcement learning and its applications. Figure 5.1 shows the result of training phase for the number of episodes.

Graphical Representation

The graphical representation reveals a significant stability concern in the reinforcement learning system, manifested by a persistent plateau in both Total Reward (-200 per episode) and Average Reward Per Step. This stagnation implies a notable challenge for the learning algorithm in adapting and enhancing its performance. The sustained low total reward indicates the agent's struggle to achieve successful outcomes, while the unchanging average reward per step reflects a lack of progress in the efficiency of the agent's actions.

To address this instability, a multifaceted strategy is essential. Firstly, an advanced exploration-exploitation strategy is warranted, possibly incorporating techniques like epsilon-greedy policies or state-of-the-art algorithms such as deep reinforcement learning. Fine-tuning hyper parameters, including exploration-exploitation ratios, is crucial to strike a balance that promotes efficient learning without being overly exploratory.

Additionally, a comprehensive review of the reward function and environment dynamics is imperative. Analyzing the impact of each component of the reward structure and identifying potential issues can guide adjustments. Introducing dynamic elements to the reward system or adapting it based on the agent's performance might enhance adaptability.

Monitoring metrics beyond the core rewards, such as the number of steps per episode and the exploration-exploitation ratio, provides a more nuanced understanding of the learning dynamics. The inclusion of a 0.5 exploration-exploitation ratio suggests a balanced approach, but further examination is needed to ensure it aligns with the underlying dynamics of the environment.

Iterative refinement, involving adjustments to algorithmic components and environment settings, is crucial. Incorporating visualization techniques, such as learning curves and heatmaps, can offer insights into the evolving behavior of the agent. The goal is to observe a positive trend in rewards, signifying improved stability and performance in the reinforcement learning system. This continuous improvement process is fundamental to overcoming the current instability and achieving successful learning outcomes.

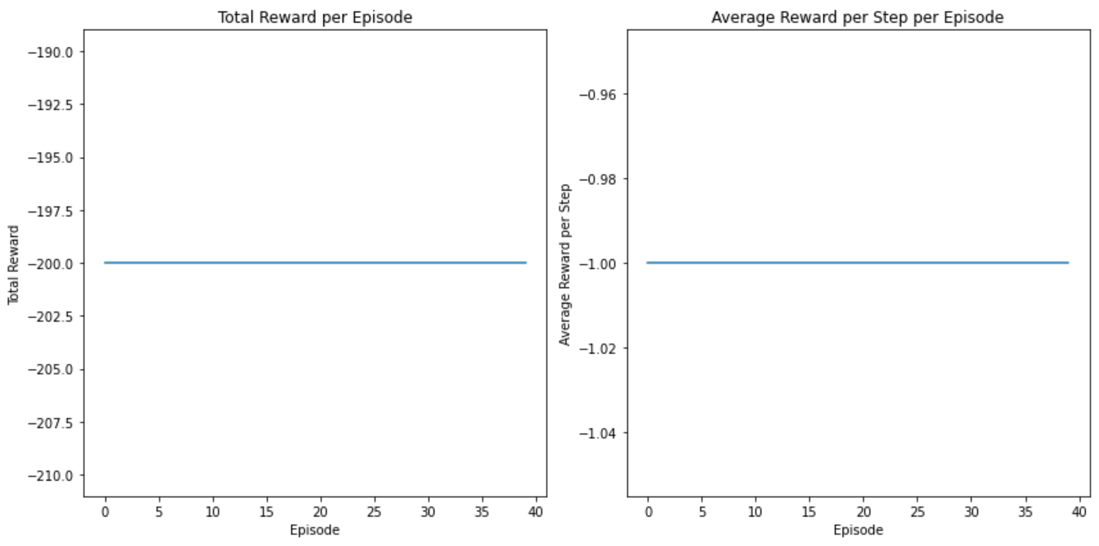


Figure 5.2: The Graph Represents Total Reward and Average Reward Per Per step to Per Episode

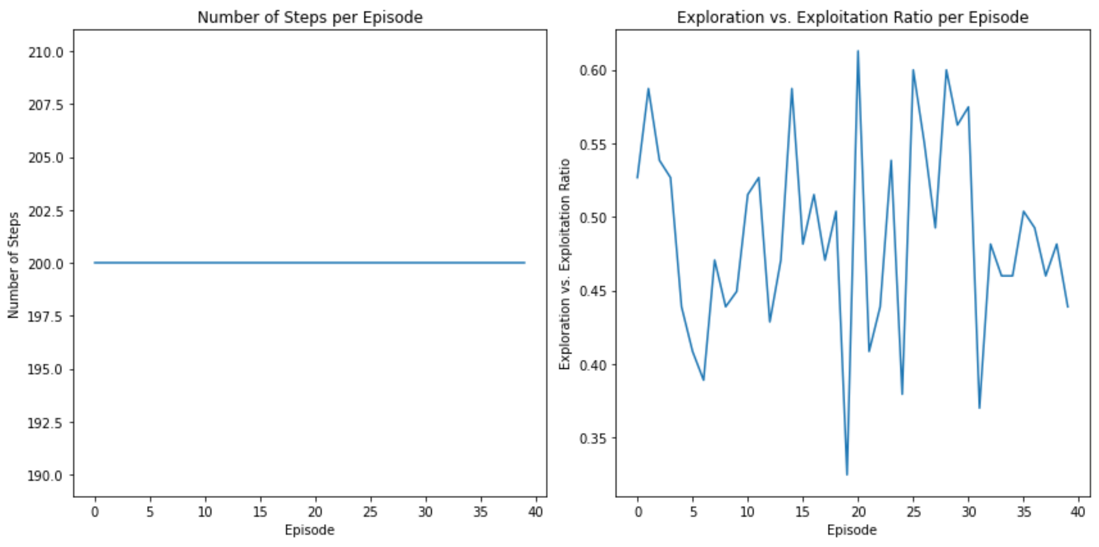


Figure 5.3: The Graph Represents Exploration and Exploitation Per Episode

5.2.3 Q learning Method

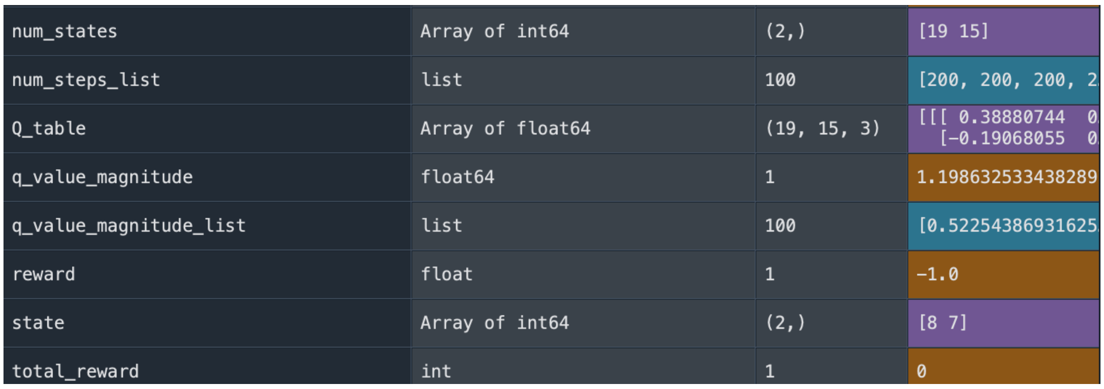
The provided code implements the Q-learning algorithm to solve the MountainCar-v0 environment within the Open AI Gym library. In this environment, the agent controls an underpowered car aiming to ascend a steep mountain road. The agent receives a reward of -1 at each time step until it successfully reaches the goal, marked by a flag at the mountain's summit. Additionally, the agent incurs a penalty of -100 if it surpasses 200 steps without reaching the goal. Q-learning is a model-free, off-policy reinforcement learning algorithm that learns the optimal action-value function by iteratively updating Q-values for state-action pairs.

The Q-learning process begins with the initialization of the Q-table, a critical component storing Q-values for each state-action pair. The table is initially populated with random values. The algorithm then iteratively updates these Q-values using the Bellman equation[18], which expresses the relationship between the Q-value of a state-action pair and the Q-values of the subsequent state and possible actions. The Q-value update follows the Q-learning rule, favoring the action that maximizes the Q-value for the next state.

To balance exploration and exploitation, the code employs an epsilon-greedy strategy for action selection. With probability epsilon, the agent selects a random action, while with probability 1-epsilon, it chooses the action with the highest Q-value. Additionally, the code incorporates a custom reward function, penalizing the agent if it takes more than 200 steps to reach the goal.

The discretization of the state space into a 10 x 100 grid forms the basis for initializing the Q-table. A function is defined to convert the continuous observation space of the environment into discrete state indices.

The main training loop executes for a predefined number of episodes. Within each episode, the agent interacts with the environment, updates Q-values based on observed rewards, and iteratively refines its policy. The training loop terminates



num_states	Array of int64	(2,)	[19 15]
num_steps_list	list	100	[200, 200, 200, 2
Q_table	Array of float64	(19, 15, 3)	[[[0.38880744 0 [-0.19068055 0
q_value_magnitude	float64	1	1.198632533438289
q_value_magnitude_list	list	100	[0.52254386931625
reward	float	1	-1.0
state	Array of int64	(2,)	[8 7]
total_reward	int	1	0

Figure 5.4: Q learning Output Results

when the agent successfully reaches the goal or exceeds the step limit.

Following the training phase, the code assesses the agent's performance by executing a single episode. The agent selects actions based on the highest Q-values, offering insights into the learned policy. Additionally, the code measures and stores several metrics during training, including the exploration-exploitation ratio, the number of steps per episode, and the magnitude of Q-values. These metrics facilitate a comprehensive evaluation of the learning process and provide valuable insights into the agent's behavior over episodes.

Graphical Representation

The graphical representation of the Q-learning algorithm's performance in the MountainCar-v0 environment provides valuable insights into the agent's learning progress. The figure illustrates key metrics such as the Total Reward per episode, the Q-function per episode, and the cumulative number of episodes, offering a comprehensive view of the algorithm's convergence.

The Total Reward per episode is a crucial measure of the agent's success in achieving the task at hand—ascending the mountain road. A Total Reward of zero indicates that the agent, on average, is neither receiving significant penalties nor achieving substantial rewards in each episode. This implies that the agent has found a balance between minimizing penalties, such as the -1 reward per time step, and maximizing positive rewards, such as reaching the goal.

The Q-function per episode represents the learned Q-values for state-action pairs throughout the training process. Q-values reflect the expected cumulative reward the agent anticipates by taking a particular action in a specific state. A well-converged Q-function is indicative of the agent's ability to estimate optimal actions for each state, facilitating effective decision-making. Observing the Q-function's evolution across episodes helps assess the learning stability and the agent's adaptability to different states and actions.

The Q-learning parameters play a pivotal role in shaping the agent's learning behavior. The chosen values, such as alpha (learning rate), gamma (discount factor), epsilon (exploration-exploitation ratio), and the total number of episodes (num-episodes), significantly influence the convergence and efficiency of the algorithm. Fine-tuning these parameters is often an iterative process, requiring a balance between exploration to discover optimal actions and exploitation to capitalize on learned knowledge.

The defined Q-values in the code, initialized with random values, undergo iterative updates guided by the Q-learning rule. The learning process involves updating the Q-values for state-action pairs based on observed rewards and the agent's estimation of the optimal actions. The Q-values gradually converge to more accurate representations of the expected cumulative rewards, reflecting the agent's learning progress.

The cumulative number of episodes plotted up to 100 provides a temporal perspective on the learning process. This metric allows the observer to track the evolution of the agent's performance over time, showcasing how the Total Reward and Q-function develop with increasing experience. The figure's limited span to 100 episodes emphasizes a concise representation of the early learning stages, capturing critical information about the algorithm's initial convergence.

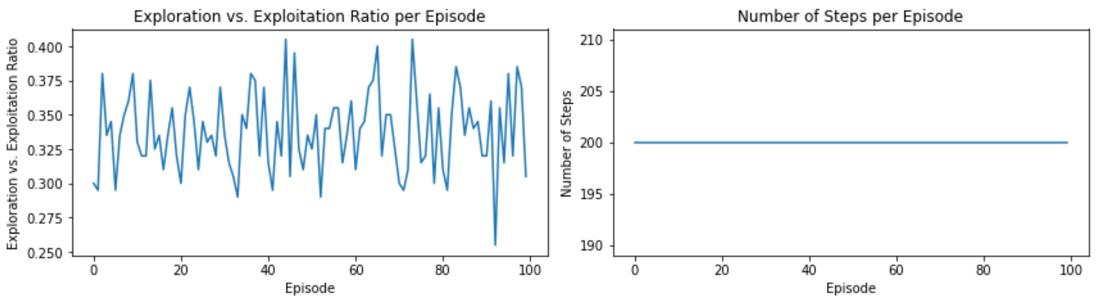


Figure 5.5: Graph Represents Epsilon per Episode

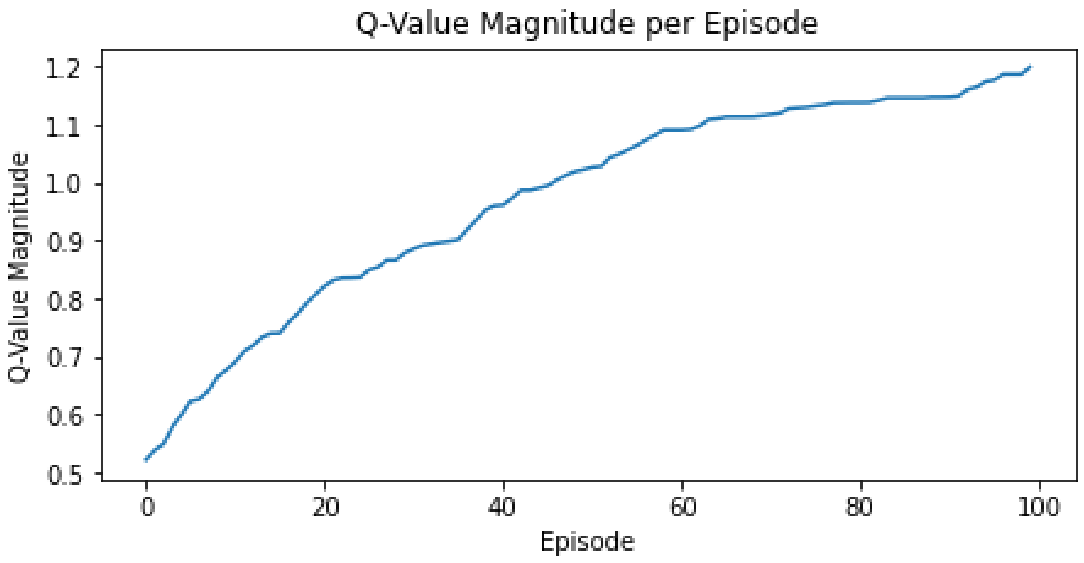


Figure 5.6: Graph Represents Q value Magnitude Per Episode

The graphical representation encapsulates the essence of the Q-learning algorithm's performance in the MountainCar-v0 environment. Through visualizing Total Reward, Q-function, and the cumulative number of episodes, one gains a holistic understanding of the agent's learning dynamics, the convergence of Q-values, and the impact of chosen parameters on the algorithm's overall effectiveness in mastering the challenging task of ascending the mountain road.

5.2.4 Q learning Neural network

The provided code showcases the implementation of the Q-learning algorithm using Deep Q-Networks (DQNs) for solving the MountainCar-v0 environment within the Open AI Gym framework. The primary objective is to train an agent to navigate a car up a hill, overcoming gravitational forces. The DQN model is constructed using the Keras library, employing the Adam optimizer and a three-layer neural network architecture with ReLU activation functions in the first two layers and a linear activation function in the output layer. The mean squared error (MSE) loss function is utilized for training, with target values calculated based on the Bellman equation[19].

The agent employs an epsilon-greedy policy, initialized with epsilon set to 1.0 and gradually decayed to facilitate exploration during the early learning stages. Experiences gained during interactions with the environment are stored in a memory buffer, and a random sample of these experiences is used to train the DQN model in batches. The replay function incorporates advanced indexing to update Q-values for taken actions and trains the model with the updated Q-values.

The agent's performance is evaluated over 100 episodes, with scores recorded and plotted to visualize the learning progress. Although the training process may be time-consuming, the agent eventually learns to successfully navigate the environment, reaching the hill's summit within the maximum allowed steps. To provide a benchmark, the code also includes a random policy function that implements a baseline random agent. This agent selects actions randomly without leveraging past experiences, offering a comparison point for assessing the efficacy of the DQN-trained agent.

Additionally, a new training function is introduced in the code, enhancing the original implementation with graphical representation of key metrics. These metrics include the total reward per episode, exploration-exploitation ratio, number of steps per episode, and epsilon decay per episode. By visualizing these metrics using Matplotlib, the training dynamics of the agent become more transparent, allowing for a comprehensive analysis of its learning behavior and performance improvements over episodes. This combined code presents a holistic perspective on the training process, evaluation metrics, and comparison with a random policy strategy

Graphical Representation

The graphical representation of the Q-learning Neural Network for the MountainCar-v0 environment offers a compelling narrative of the agent's learning journey and the convergence of the neural network to efficiently solve the task. Examining the key components of the plot, including the "Total Reward per Episode" and "Epsilon Decay per Episode," provides a nuanced understanding of the agent's progress.

The "Total Reward per Episode" plot is a pivotal visualization that encapsulates the agent's performance throughout the training process. The gradual increase in total rewards over episodes signifies the agent's ability to learn and adapt its policy effectively. Initially, the agent explores the environment, and the rewards might be low as it navigates the complexities of the task. However, as the agent accumulates experience and refines its strategy, a noticeable upward trend emerges in the plot. Specific episodes, such as 34, 38, 60, and 75, stand out as milestones where the agent successfully achieves the final state, reaching the top of the hill. These peaks in total rewards highlight critical moments in the learning process, indicating when the agent consistently executes optimal actions to accomplish the goal.

Simultaneously, the "Epsilon Decay per Episode" plot reveals the evolution of the exploration-exploitation trade-off. Epsilon represents the probability of the agent taking a random action, fostering exploration in the early stages of training.

As depicted in the plot, the epsilon value consistently reduces over episodes. This reduction signifies the agent's decreasing reliance on exploration, indicating a transition towards exploitation. The agent becomes more confident in its learned policy, relying less on random actions and more on its acquired knowledge to maximize rewards. The correlation between the reduction in epsilon and the increase in total rewards underscores the successful integration of exploration and exploitation in the agent's learning strategy.

The identified episodes where the agent reaches the final state align with crucial points in the epsilon decay. As the agent learns a more optimal policy, it requires less exploration, and the diminishing epsilon reflects this shift. The positive rewards obtained in later episodes further validate the effectiveness of the Q-learning Neural Network. The agent not only learns to navigate the environment but also consistently achieves positive outcomes, demonstrating a robust and adaptive policy.

Analyzing these graphical representations collectively unveils the success of the Q-learning Neural Network in solving the MountainCar-v0 task. The agent's learning trajectory is characterized by a strategic balance between exploration and exploitation, leading to a progressive increase in total rewards and, ultimately, successful task completion. This narrative of learning progression is indicative of the neural network's ability to capture and adapt to the complexities of the environment, showcasing the efficacy of Q-learning in training intelligent agents. Fig 5.7 and fig 5.8 describe the total reward per episode and entire Q Learning Neural Network spefications.

5.2.5 Pytorch

The provided code implements a Deep Q-Network (DQN) to solve the MountainCar-v0 environment from Open AI Gym using PyTorch. The DQN is a reinforcement

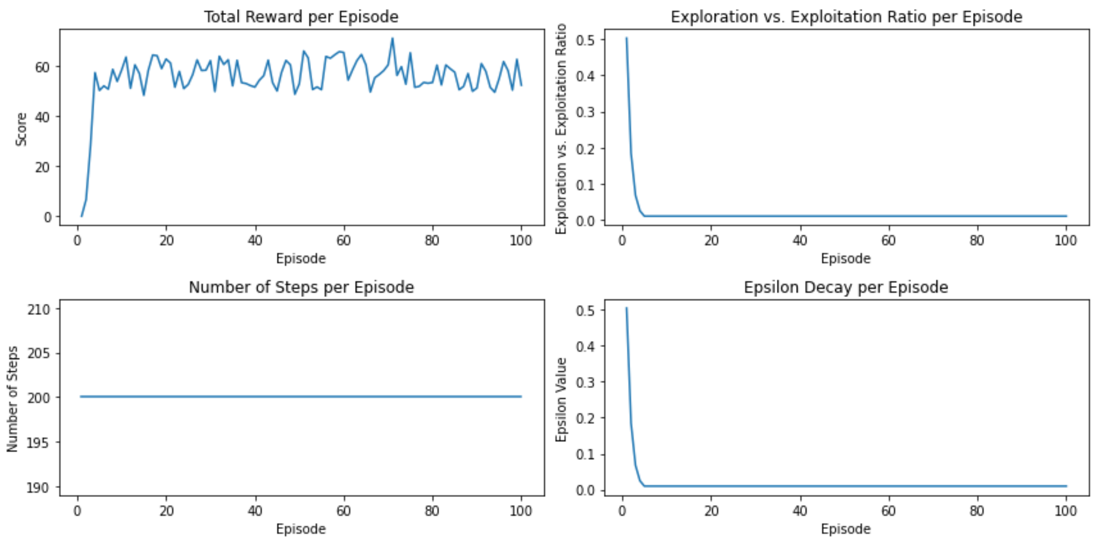


Figure 5.7: Q learning Neural Network Output specifications

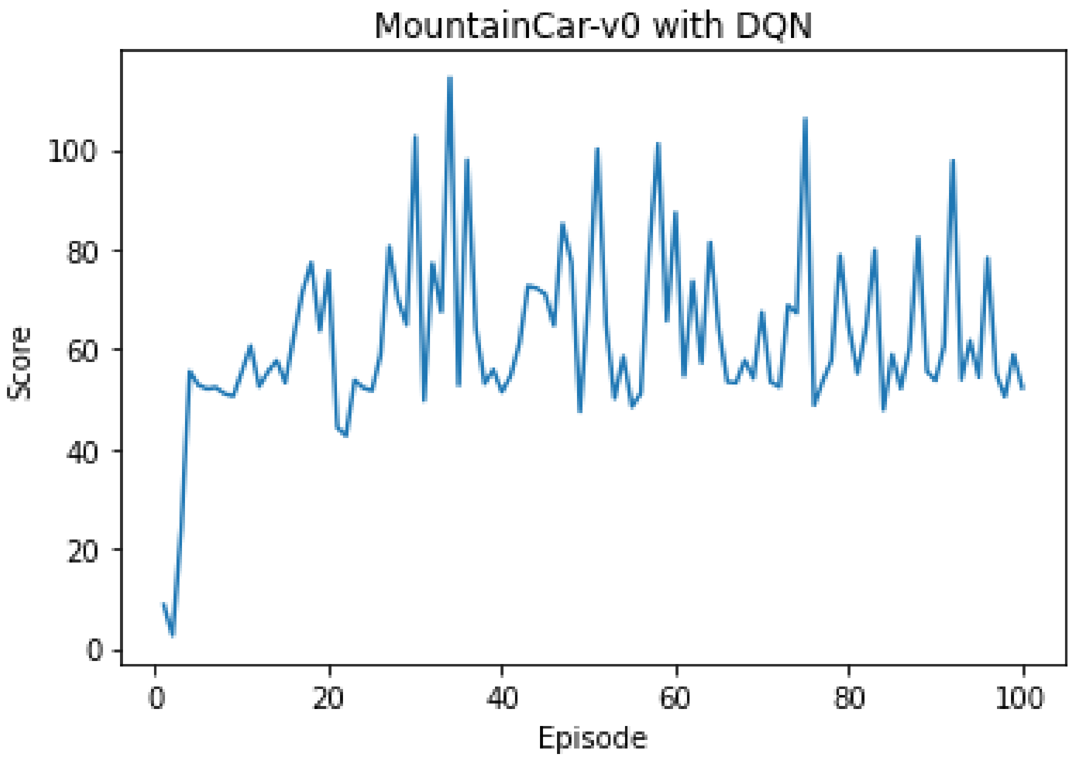


Figure 5.8: Total reward per Episode

learning algorithm that employs a neural network to approximate the Q-values of state-action pairs, enabling an agent to learn a policy for optimal decision-making.

The DQN class defines the neural network architecture with three fully connected layers. The agent, represented by the DQNAgent class, utilizes an epsilon-greedy strategy for action selection, balancing exploration and exploitation. The agent's experiences are stored in a replay memory, and the DQN model is trained through a replay mechanism, updating its parameters to minimize the Mean Squared Error (MSE) loss between predicted and target Q-values.

The training loop in the train-dqn function runs for a specified number of episodes, where the agent interacts with the environment. The agent's actions are determined by the DQN model, and the environment renders the state transitions. The reward function is customized to encourage the agent to reach the goal state at the top of the hill. The agent's experiences are stored in the replay memory, and the replay function is called to train the DQN using a batch of random samples from the memory.

The random-policy function demonstrates a random policy where the agent takes random actions in the environment, serving as a baseline for comparison against the DQN's learned policy.The main execution block initializes the MountainCar-v0 environment, creates an instance of the DQNAgent, and trains the agent using the train-dqn function. The training progress is visualized by plotting the scores achieved in each episode.

This code leverages PyTorch to implement a DQN for solving the MountainCar-v0 task, demonstrating the key components of a reinforcement learning system, including neural network architecture, experience replay, and epsilon-greedy exploration. The agent learns to navigate the environment and achieve the goal state through iterative training episodes. The resulting plot provides insights into the learning progress, showcasing the agent's ability to accumulate rewards

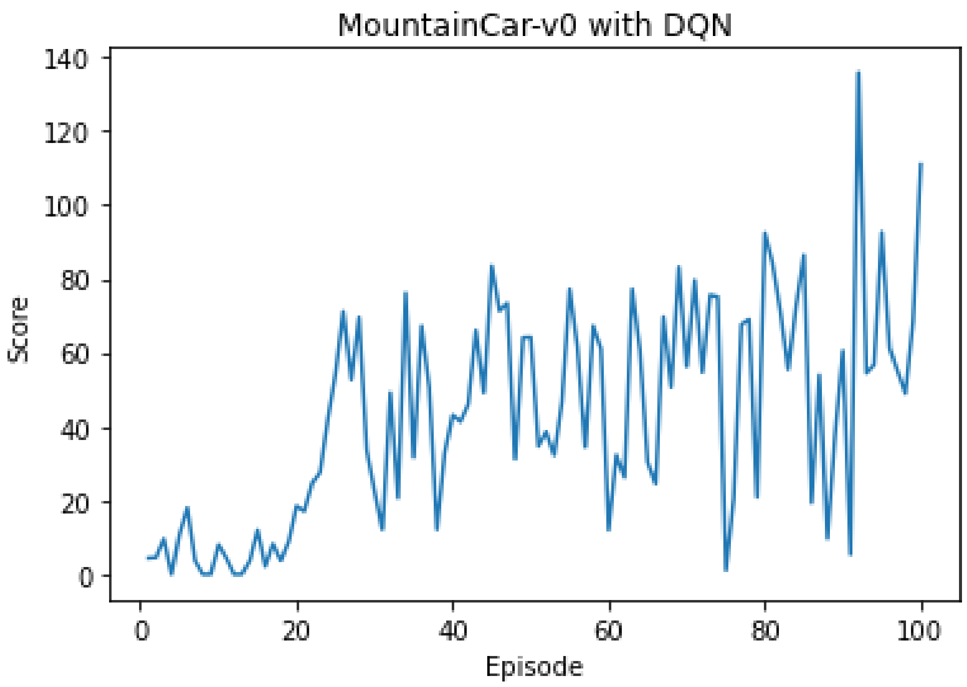


Figure 5.9: total reward per episode using pytorch

over episodes.

Graphical Representation

The graphical representation of the PyTorch-based DQN in the MountainCar-v0 environment depicts the agent's learning progress over episodes. At various stages, the car reaches the final state, as indicated by peaks in the total reward per episode. The x-axis represents the episodes, while the y-axis shows the corresponding scores achieved by the agent. The episodes where the car successfully reaches the final state are highlighted, showcasing peaks in the score graph.

The graphical representation provides a visual understanding of the learning dynamics, highlighting key milestones where the car conquers the challenging MountainCar-v0 environment. The plotted graphs serve as valuable insights into the training process, emphasizing the agent's ability to navigate and succeed in the complex task.

5.3 Network Verfication

Saving the trained network and its training procedure, such as the "train-dqn" function, is a crucial step in the machine learning workflow. This allows you to store the learned parameters of the neural network, enabling you to retrieve and reuse the model for further analysis or applications without having to retrain it from scratch. There are various ways to save a trained model, such as using serialization libraries like Pickle or using dedicated functions provided by deep learning frameworks like TensorFlow or PyTorch.

Once the trained network is saved, the next step involves verifying the network by adjusting parameters. This verification process is essential for fine-tuning the model's performance, ensuring it meets specific requirements, or adapting it to different tasks.adjusting model parameters, like the weights and biases, can be done through techniques like transfer learning or fine-tuning. This involves leveraging knowledge gained from a pre-trained model on a related task and adapting it to the specific problem at hand.

In summary, saving a trained network facilitates reusability and further analysis. Verifying the network involves adjusting both hyperparameters and model parameters to ensure optimal performance, adaptability, and generalization across different scenarios. This iterative process of saving, adjusting, and verifying is fundamental in the development and optimization of neural networks for various applications.

working Procedure :-

To define a property for Polyhedral testing, begin by executing the neural network with the original input to obtain the output tensor. Now, create the property using NeVer2 by incorporating the input tensor, output tensor, and the specified noise. For instance, if the input tensor is [-0.112, 1.648], corresponding output [y0, y1, y2], and noise is 0.05, create input constraints like X-0 ¡= " ", X-0 ¿= " ", X-1 ¡= " ", and X-1 ¿= " ". These constraints define a region around the initial input that accounts for the noise.

Similarly, for the output, compute y0 +- 0.05, y1 +- 0.05, and y2 +- 0.05, creating constraints for each output variable to ensure they fall within the desired range. This process ensures that the property reflects the expected behavior within the specified noise margin.

NeVer2 allows the generation of Polyhedral properties by defining constraints on both input and output tensors, accounting for the noise introduced during testing. This approach enables effective property testing that considers the variability in the neural network's predictions within the given noise threshold.

The sample Property define in Never2 Tool looks like these

```
(declare-fun X_0 () Real)
(declare-fun X_1 () Real)
(declare-fun Y_0 () Real)
(declare-fun Y_1 () Real)
(declare-fun Y_2 () Real)

(assert (<= X_0 -0.062))
(assert (>= X_0 -0.162))
(assert (<= X_1 1.698))
(assert (>= X_1 1.598))

(assert (<= Y_0 7.1))
(assert (>= Y_0 7.09))
(assert (<= Y_1 7.4))
(assert (>= Y_1 7.39))
(assert (<= Y_2 7.41))
```

Property	**Input**	**Noise**	**Output + Noise**
1	[-0.112 ,1.648]	-0.05	[7.0932 , 7.3965 , 7.4057]
2	[-0.150 , 1.990]	-0.1	[8.0111 , 8.3652 , 8.3680]
3	[-0.200 , 2.100]	-0.08	[8.3080 , 8.6754 , 8.6768]
4	[-0.100 , 1.500]	-0.20	[6.5198 , 6.8016 ,6.8141]
5	[-0.120 , 1.700]	-0.30	[7.2886 ,7.5990 ,7.6071]
6	[-0.090 ,1.900]	-0.01	[7.8918 ,8.2367 ,8.2405]
7	[0.340 ,4.300]	-0.150	[15.1875 ,16.0830,15.9817]
8	[-0.450,2.600]	-1.00	[8.6262 , 9.0521 ,9.0470]

Table 5.2: Network output and Input Properties

```
(assert (>= Y_2 7.4))
```

The given code is written in the SMT-LIB language, a standard format for specifying problems to Satisfiability Modulo Theories (SMT) solvers. These constraints restrict the possible values for the variables within specified intervals. The utilization of SMT-LIB allows automated solvers to check whether a solution satisfying these constraints exists, making it a powerful tool in formal verification and validation processes, particularly in fields like formal methods and software verification. The snippet likely represents a mathematical model or problem, and the constraints define a feasible solution space for the specified variables

The presented tables, Table 5.2 and Table 5.3, encapsulate a comprehensive verification analysis of a system's properties under various conditions. Table 5.2 delineates specific properties, their corresponding inputs, introduced noise, and the resultant output with added noise. Each row corresponds to a unique property, offering insights into the system's behavior across a range of scenarios.

For instance, Property 1 involves an input range of [-0.112, 1.648], subject to a noise of -0.05, yielding an output with added noise in the range [7.0932, 7.3965, 7.4057]. The subsequent properties follow a similar pattern, providing a detailed

Property	Never2-time	Never2-Result
1	31.0694557920001	True
2	14.018025041000328	False
3	28.960818666000705	True
4	29.27622454199991	False
5	28.96916683299969	True
6	28.785308958000314	True
7	28.735463166000045	False
8	12.010101417000442	False

Table 5.3: Never-2 output Results

examination of how the system responds to different inputs and noise levels.

Table 5.3 appears to display verification results, possibly indicating whether certain properties hold true under specific conditions. Each row represents a property, its corresponding numerical result, and a Boolean value denoting the verification outcome. The inclusion of True or False signifies whether the specified property is verified or not.

An analysis of Table 5.3 reveals that Properties 1, 3, 5, and 6 are verified (True), suggesting that under the given conditions, these aspects hold. Conversely, Properties 2, 4, 7, and 8 are not verified (False), indicating that the specified conditions do not satisfy these particular system requirements.

These tables collectively contribute to a comprehensive understanding of the system's behavior and its verification outcomes. Such detailed analysis is crucial in domains where system correctness and adherence to specified properties are paramount, such as formal methods, model checking, and software verification[20]. The juxtaposition of input-output behaviors and verification results provides a holistic view of the system's performance under diverse scenarios, aiding in decision-making and ensuring the system's reliability in real-world applications

Chapter 6

Conclusions

The exploration of Reinforcement Learning (RL) fundamentals, from core components to the evolution from tabular methods to neural networks, sets the foundation for understanding complex problems. Bellman equations, dynamic programming, and model-free/model-based approaches provide a comprehensive view of RL techniques. The transition to neural networks, such as Convolutional Neural Networks and the Actor-Critic architecture, highlights the architectural elegance and sophistication in modern RL.

The introduction of Soft Actor Critic (SAC) emphasizes the need for flexibility in learning policies, a crucial aspect in real-world applications where adaptability is key. As RL methodologies advance, the integration of tools and frameworks becomes imperative. Open AI Gym, PyTorch, and TensorFlow are fundamental in providing environments, libraries, and computational frameworks that facilitate RL research and development.

The inclusion of Never2 Tool, showcases a practical implementation of RL. Its architectural design, installation process, Software Architecture, and procedures for building models, defining properties, and handling models through command-line interface exemplify a comprehensive RL toolkit. The tool's functionalities extend to training networks, defining verification strategies, and visualizing outputs.

In the experimental results section, the focus shifts to real-world applications through the evaluation of Mountain Car -v0 environment. The detailed exploration of experiments, basic methods, Q learning approaches, and the use of neural networks in Q learning provides insights into the tool's effectiveness. The comparison between PyTorch and TensorFlow underscores the significance of choosing appropriate frameworks based on specific requirements.

The network verification process becomes a critical aspect of RL applications, ensuring the reliability of trained models. As depicted in Table 5.2 and Table 5.3, verification results showcase the success and failure of specific properties under given conditions. The systematic approach to testing and verifying the network's behavior adds robustness to the tool.

In conclusion, the amalgamation of theoretical RL concepts, practical tool implementation, and real-world experimentation provides a holistic understanding of RL applications. The Never2 Too serves as a practical embodiment of RL methodologies, demonstrating the versatility and adaptability required for real-world scenarios. The verification strategies employed ensure the reliability of the tool, crucial in applications where incorrect decisions can have significant consequences.

Future work in RL could involve enhancing the tool's capabilities, expanding its applicability to diverse environments, and incorporating advancements in RL research. Additionally, the integration of explainable AI (AI) techniques could enhance interpretability, making RL models more transparent and trustworthy. Overall, the field of RL continues to evolve, and future research could focus on addressing challenges in scalability, robustness, and ethical considerations, contributing to the widespread adoption of RL in real-world applications

References

[1] A. T. Luca Pulina, “An abstraction-refinement approach to verification of artificial neural network,” 2010.

[2] A. T. Luca Pulina, “Challenging SMT solvers to verify neural networks. AI Commun,”

[3] L. P. A. T. Francesco Leofante, Nina Narodytska, “Automated Verification of Neural Networks: Advances, Challenges and Perspectives. CoRR abs/1805.09938,” 2018.

[4] L. P. A. T. Dario Guidotti, Francesco Leofante, “Verification of Neural Networks: Enhancing Scalability Through Pruning. ,” 2020.

[5] A. T. Dario Guidotti, Luca Pulina, “pyNeVer a Framework for Learning and Verification of Neural Networks. ATVA 2021.. ,” 2020.

[6] F. L. Junjie Bai and K. Z. ONNX, “Open Neural Network Exchange, https://github.com/onnx/onnx, ,” 2023.

[7] S. B. nnenum, “Verification of relu neural networks with optimized abstraction refinement. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, C esar A. ,” 2021.

[8] A. S. Clark Barrett and C. Tinelli., “The SMT-LIB Standard: Version 2.0.

In A. Gupta and D. Kroening, editors, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) ,” 2010.

[9] J. K. A. L. Elena Botoeva, Panagiotis Kouvaros and R. Misener, “Ef- ficient verification of relu-based neural networks via dependency analysis. In Proceedings of the AAAI Conference on Artificial Intelligence) ,” 2020.

[10] S. B. T. T. J. Christopher Brix, Mark Niklas Mu ller and C. Liu., “First three years of the international verification of neural networks competition (vnn-comp). arXiv preprint arXiv:2301.05815, ,” 2023.

[11] C. Brix and T. N. Debona:, “Decoupled boundary network analysis for tighter bounds and faster adversarial robustness proofs. arXiv preprint arXiv:2006.09040,,,” 2020.

[12] A. P. Stefano Demarchi, Dario Guidotti and A. Tacchella:, “Formal verification of neural networks: a case study about adaptive cruise control. In International ECMS Conference on Modeling and Simulation, pages 310–316,” 2020.

[13] N. J. laudio Ferrari, Mark Niklas Mu ller and M. T. Vechev, “Complete verifi- cation via multi-neuron relaxation guided branch-and-bound. In The Tenth International Confer- ence on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net,,” 2022.

[14] J. S. an J. Goodfellow and C. Szegedy., “Explaining and harnessing adversarial examples. In 3rd International Conference on Learning Representations, ICLR 2015, San Diego,,” 2015.

[15] A. T. Dario Guidotti, Stefano Demarchi and L. Pulina., “he Verification of Neural Networks Library (VNN-LIB), www.vnnlib.org,,” 2023.

[16] L. P. Dario Guidotti and A. T. pyNeVer, "A framework for learning and verification of neural networks. In International Symposium on Automated Technology for Verifi- cation and Analysis, pages 357–363. Springer,," 2021.

[17] P. Henriksen and A. Lomuscio., "Efficient neural network verification via adaptive refine- ment and adversarial search. In ECAI 2020, pages 2513–2520. IOS Press,," 2020.

[18] P. Henriksen and A. L. Deepsplit:, "An efficient splitting method for neural network verification via indirect effect analysis. In IJCAI, pages 2549–2555, ," 2021.

[19] S. B. C. L. Mark Niklas Mu ller, Christopher Brix and T. T. Johnson, "The third international verification of neural networks competition (vnn-comp 2022): Summary and results. arXiv preprint arXiv:2212.10376, ," 2021.

[20] M. P. Hadi Jahanbakhti and A. Yazdizadeh, "Online neural network-based model reduction and switching fuzzy control of a nonlinear large-scale fractional-order system. Soft Computing, March ," 2023.