Neural Network Architecture Optimization Using Reinforcement Learning

by

RAGHAV VADHERA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2023

To my father Kewal Krishan and my late mother Vishwa
who set the example and instilled a love of learning and hard-work in me and made me who I am.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervising professor, Dr. Manfred Huber, for his continuous motivation, encouragement, and invaluable guidance throughout my doctoral studies. I am grateful to my academic advisors, Dr. Farhad Kamangar and Dr. David Levine, for teaching me relevant courses and providing valuable feedback on my research. I also want to thank Dr. John Carbone for his keen interest in my research and for taking the time to serve on my dissertation committee.

I would like to extend my appreciation to my company, Raytheon, for providing financial support during my doctoral studies. I am grateful to Jonathan Clyburn and Shane Zabel from Raytheon for assisting me in selecting my research topic and offering valuable feedback promptly. A special thank you goes to Dr. Abhaya Asthana for his interest in my research and the helpful discussions & invaluable comments he provided.

I am thankful to all the teachers who have taught me throughout my academic journey, from my early years in India to my time in the United States. I would like to extend my gratitude to Dr. John Paul from Harvard for inspiring and encouraging me to pursue PhD studies.

Above all, I am grateful to my friend, Rashmi Verma, Sr. Audit Manager, Financial Crimes, an AI enthusiast whose unwavering support and encouragement made this work possible while working late nights. I also want to thank my wife Ritu Vadhera, helping me overcome challenges and celebrate achievements.

May 10, 2023

ABSTRACT

Neural Network Architecture Optimization Using Reinforcement Learning

RAGHAV VADHERA, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professor: Manfred Huber

Deep learning has emerged as an increasingly valuable tool, employed across a myriad of applications. However, the intricacies of deep learning systems, stemming from their sensitivity to specific network architectures, have rendered them challenging for non-experts to harness, thus highlighting the need for automatic network architecture optimization. Prior research predominantly optimizes a network for a single problem through architecture search, necessitating extensive training of various architectures during optimization.

To tackle this issue and unlock the potential for transferability across tasks, this dissertation presents a novel approach that employs Reinforcement Learning to develop a network optimization policy based on an abstract problem and architecture embedding. This approach enables the optimization of networks for novel problems without the burden of excessive additional training. Leveraging policy learning and an abstract problem embedding, the method facilitates the transfer of the policy across problems by capturing essential characteristics of the network domain and target task that permit the approach to optimize the networks for new challenges based on characteristics learned from previous problems.

Initial evaluations of this method's capabilities were conducted using a standard classification problem, demonstrating its effectiveness in optimizing architectures for a specific target problem within a given range of fully connected networks. Subsequent experiments were performed using a variety of complex problems, further showcasing the approach's capabilities. To address these more complex networks, Siamese networks were employed to establish a coherent embedding of the network architecture space. In conjunction with a problem-specific feature vector, which captures the intricacies of the problem, the Reinforcement Learning agent was able to acquire a transferable policy for deriving high-performing network architectures across a spectrum of problems.

Experiments performed in this dissertation specifically reveal that the proposed system successfully learns an embedding space and policy that can derive and optimize network architectures nearing optimality, even for unencountered problems. Multiple datasets, each possessing unique feature vectors representing distinct characteristics entities or problems, were utilized to facilitate the optimization of one problem at a time. A random initial policy was employed to construct trajectories in the embedding space during training. To assess the performance and functionality of various network components, a series of pre-training steps were undertaken, focusing on distinct components and examining the outcomes prior to training subsequent components.

Building upon these foundations, the dissertation takes initial steps to examine the scalability of the method to larger and more intricate network architectures with the intent of broadening its applicability across a diverse array of problem domains.

To validate the generalizability of the learned policies, the dissertation examines their performance on real-world problems, spanning various industries and

domains, including healthcare, finance, sports, human psychology and auto. These case studies aim to demonstrate the practical utility of the proposed approach in addressing real-world challenges and uncover potential areas for further refinement and improvement.

In addition to these empirical investigations, the dissertation discusses the theoretical underpinnings of the method, examining the convergence properties, stability, and robustness of the learned policies. These investigations provide valuable insights into the factors that influence policy transferability and optimization performance across diverse problem domains, offering guidance for future research in the field of deep learning and network architecture optimization.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

OVERVIEW

1.1 Introduction

Deep Neural Networks have emerged as a robust and versatile instrument for tackling a wide array of problems in recent years. Nonetheless, their performance is largely contingent upon the specific network architecture and the dataset at hand. Benefits of automatic neural architecture design has been hinted at recently through initial experiments indicating its ability to unearth powerful network architectures that excel in some challenging learning tasks. The last few years have witnessed the remarkable accomplishments of deep neural networks in various demanding applications, including speech recognition [3], image recognition [4] [5] and machine translation [6] [7] [8]. This success has prompted a paradigm shift from feature design to architecture design, moving from methodologies such as SIFT [9], and HOG [10], to advanced architectures like AlexNet [5], VGGNet [11], GoogleNet [12], and ResNet [13]. Despite these advancements, designing architectures still necessitates considerable expertise and time investment. The potential benefits of automatic neural architecture design to address some of these difficulties has been hinted at recently through initial experiments indicating its ability to unearth powerful network architectures that excel in some challenging learning tasks.

Although Automatic Neural Architecture (ANA) design [14] has been an area of research for a number of years, certain obstacles persist in the development of an optimized ANA model. This results in lack of assurances that the devised architecture will be well-suited for the problem under consideration. Employing

an appropriate neural network architecture, in conjunction with a sufficiently large dataset, allows a deep learning network to learn any mapping from one vector space to another, rendering deep learning an incredibly potent tool for various machine learning tasks. Designing an optimal neural network , however, entails crafting the ideal architecture to yield the best possible outcomes. This "optimum" is often an ambiguous concept, as it hinges on striking a balance between model performance and the computational resources required for training and prediction. Despite this loose definition of "optimum," the initial step in constructing any neural network involves establishing a starting point and an initial set of hyperparameters.

In general, devising a suitable architecture is heavily influenced by the properties of the dataset. For sequential models involving multilayer perceptrons (MLP), for example, one of the primary starting points (hyperparameters) is determining the number of hidden layers and the nodes required for these layers [15]. Without additional data, this decision is exceedingly, underscoring the need for more sophisticated automatic network architecture design tools to open up use of these techniques to a wider non-expert audience.

The objective of this research is to develop a more comprehensive framework that employs Reinforcement Learning, coupled with a flexible architecture modification action space, to perform network architecture optimization on a wider task domain through by building on effective architecture and target task embedding. This framework will also facilitate planning via a performance prediction [16] component to reduce the need for incremental training for architecture evaluation, ultimately learning a transferable policy for architecture optimization. Preliminary experiments presented in this paper demonstrate that the framework, when trained on a set of classification problems, can successfully learn a policy that re-

sults in an optimized target architecture for a new problem without necessitating any significant retraining.

Expanding on this foundation, the research aims to explore the practical implications and applications of the proposed framework. It performs a preliminary investigation of the scalability of the method to larger and more complex network architectures, with the eventual goal to expand to convolutional and recurrent neural networks as well as more general mulitibranched networks.

In order to assess the applicability of the acquired policies, this study investigates their effectiveness when confronted with real-world data issues spanning various sectors and fields, such as healthcare, finance, sports, human psychology, and automotive. These case studies aim to highlight the practical advantages of the suggested framework for tackling real-world obstacles, as well as pinpointing areas where further enhancements and developments are needed.

In addition, the research experiments with convergence characteristics, stability, and resilience of the formulated policies through different training and pretraining approaches. These examinations yield crucial insights into aspects that impact policy transferability and optimization success across diverse problem domains, offering direction for future inquiries in the realm of deep learning and automated neural architecture design.

Overall, this research endeavors to establish a framework that incorporates Reinforcement Learning and an adaptable architecture modification action space in order to streamline the optimization of deep neural network structures. Through extensive testing and assessment, the study showcases the potential of this framework to substantially enhance the automated neural architecture design process, rendering it more user-friendly and efficient for both novices and seasoned professionals. By connecting intricate network architectures with real-world applications,

this approach lays the groundwork for future progress in deep learning and AI-driven solutions across many sectors and fields.

1.2 Different Approaches.

The various approaches adopted to address Automatic Neural Architecture (ANA) modeling in the past have explored many different aspects of the process, from data collection and pre/post-processing to intricate training schemes and algorithms. Over the years, ANA modeling has seen numerous approaches aimed at optimizing neural network architecture design automatically. There has been a growing focus on developing a systematic method to determine an appropriate architecture, as opposed to the current trial-and-error approach, which is time-consuming and produces uncertain results [17]. One complicating factor in this endeavor has been that optimal network architectures are generally not known for wide ranges of problems and thus Reinforcement Learning (RL) has often been used and proven effective in learning a policy to incrementally construct neural network classifiers for specific problems [17] [1].

In reinforcement learning based methods, the choice of a component of the architecture is regarded as an action. A sequence of actions from a given initial network defines a resulting neural network architecture, with its performance improvement serving as the reward. Unfortunately, this approach has so far mainly focused on simple classification problems using a limited action space with small fully connected networks or been trained individually on a single problem and has thus in its previous applications not been applicable to a broader range of problems. Many previous techniques rely on a small, predefined set of architecture modification actions [17][2], which restricts their applicability to specific architec-

ture types. One attempt to address this limitation involves using an architecture embedding space to allow for a general, continuous action space [1] [18]. However, target network performance in this embedding space is often highly non-convex, leading to numerous local extrema. This issue confines optimization in these systems to single problems, necessitating intensive retraining for each target problem and extensive evaluation training for each individual architecture optimization.

The work in [2] tried to address the need to retrain on every problem and focused on generalizing target problems by learning a generic network optimization policy. The method learns not only a single network but an ensemble of networks, enabling faster adaptation to problem changes. However, this approach is limited by its simple action space which operate on ensembles of small fully connected networks and is not easily extendable to a wider range of problems and architectures. The method learns not only a single network but an ensemble of networks, enabling faster adaptation to problem changes. Part of the method's limitation stems from its use of specific modification actions that add or remove nodes and layers or disable and enable elements in the ensemble. Similar Reinforcement Learning-based methods also perform search within discrete architecture space, which is natural given the inherently discrete choices of neural network architectures, such as the filter size in CNNs and connection topology in RNN cells. Nonetheless, direct searching for the best architecture within discrete space is inefficient due to the exponential growth of the search space as the number of choices increases, leading to embedding-based approaches [1] [18] which aim to generate a generic, continuous network architecture, and thus network modification, space. In this dissertation we expand on this and experiment with mapping architectures into a continuous vector space (i.e., network embeddings), augmenting it with a target task embedding, and optimizing within this continuous space through policy gra-

dient methods. The rationale for such an embedding-based approach is two-fold. Firstly, akin to the distributed representation of natural language,[19], [20], a continuous architecture representation is more compact and efficient in conveying its topological information. Secondly, optimizing in a continuous space is considerably easier than directly searching within discrete space due to improved smoothness. The proposed architecture's core is an encoder-decoder model responsible for mapping and recovering a neural network architecture into a continuous representation. Multiple models for the autoencoder are developed and tested here, with the most competent being an LSTM model equipped with an attention mechanism that facilitates precise recovery. The three components (i.e., encoder, performance (accuracy) predictor, and decoder) are jointly trained in a multi-task setting, which benefits the continuous representation: the decoder's objective of recovering the architecture further enhances the quality of the architecture embedding, making it more effective in predicting performance (accuracy).

The following outlines some of the other notable methods adopted in the field of Automatic Neural Architecture (ANA) modeling , beyond reinforcement learning-based systems.

1.2.1 Genetic Algorithms (GAs)

Genetic algorithms are inspired by the process of natural selection, utilizing mutation, crossover, and selection operators to evolve a population of neural architectures. This approach has been employed to optimize various aspects of network design, such as layer connections and the number of hidden nodes [21]. However, it is generally limited by its high computational complexity.

1.2.2 Particle Swarm Optimization (PSO)

PSO is a population-based optimization algorithm inspired by the social behavior of birds and fish. It has been used to optimize hyperparameters and architecture design in neural networks by iteratively updating candidate solutions. Similar to Genetic Algorithms, it finds its limitations in the computational complexity of the approach, despite its ability to be highly parallelized.

1.2.3 Bayesian Optimization

Bayesian optimization is a sequential model-based optimization technique that leverages Gaussian processes or other surrogate models to approximate the objective function. This approach is used to optimize hyperparameters and network design by balancing exploration and exploitation in the search space [22]. In many of its applications in this domain, however, it suffers from the need for significant network re-evaluation for new problems and thus does not generalize efficiently to new problems.

1.2.4 Neural Architecture Search (NAS)

NAS is a search algorithm that explores the space of possible neural network architectures using various search strategies, such as reinforcement learning, evolutionary algorithms, and gradient-based methods. The objective is to find the optimal architecture for a given task with minimal human intervention [23]. In its broader interpretation, the approach developed in this dissertation falls under this framework where it attempts to combine the benefits of a continuous action space on network and task embeddings and the ability or reinforcement learning to overcome local minima in performance space through the use of a utility function to permit efficient generalization across networks and tasks.

1.2.5 Differentiable Architecture Search (DARTS)

DARTS is a gradient-based approach to neural architecture search, which formulates the architecture optimization problem as a continuous relaxation of the search space. By doing so, the architecture can be optimized using gradient descent, leading to efficient search processes [24]. However, the presence of significant local maxima in performance space requires expensive local re-training during optimization, making rapid application to a new problem difficult.

1.2.6 Transfer Learning and Meta-Learning

These approaches focus on leveraging prior knowledge learned from related tasks or architectures to accelerate the search for an optimal architecture in the target task. Transfer learning involves reusing pre-trained models or weights, while meta-learning aims to learn a general optimization policy across tasks [25]. The approach introduced here aims the capabilities of prior transfer and meta learning approaches to minimize the need for training on the target problem through additional task embedding.

These different methodologies present a wide spectrum of concepts to address Automatic Neural Architecture modeling. While each method has its own advantages and limitations, their combined advancements contribute to the ongoing progress in the field of deep learning and automated neural architecture design.

1.3 Summary and Contributions

Leveraging Reinforcement Learning to learn representations that can adapt to changing task specifications and building on previous work, the objective of the

research in this dissertation is to develop a more comprehensive framework that can also create policies with larger action spaces in terms of network architecture modification options and that can generalize across target tasks. This allows for the representation of a broader learning task domain by embedding it in an architecture embedding space with a derived performance prediction component to facilitate planning.

This dissertation presents an architecture that addresses these requirements by integrating actor-critic reinforcement learning with an encoder-decoder approach and applying it to various target machine learning problems. This facilitates the inclusion of structures that can adapt to larger spaces, eventually potentially including modular network construction as well as use on streaming data [26] and moving targets [2], where RL-based model learning has demonstrated some success. Drawing inspiration from previous work on network embeddings [1], [2], [18], [27] and various approaches to network architecture search [17], [28], [29], [14], a novel approach is proposed that combines the concepts of a network embedding space with target problem features (for fully connected networks and beyond). This approach aims to learn a transferable policy that allows for more general architecture spaces and reduces the need for extensive retraining for new problems. Furthermore, it facilitates the transfer of optimization strategies across problems.

To address the challenges posed by the complex dynamics of network performance under architecture transformations in an embedding space, the approach consolidates and organizes legal network embeddings using Siamese networks. Additionally, it employs a Reinforcement Learning framework for complex policy formation that enables the system to more reliably achieve high-performing final architectures, even in the presence of significant local extrema for task performance.

1.4 Outline

The remainder of this dissertation is structured as follows: Chapter 2 provides a review of the state-of-the-art approaches in network architecture optimization, with a focus on Reinforcement Learning, in order to establish the background and context for the related work. Chapter 3 introduces the fundamental concepts that underpin our proposed architecture, elucidating the roles of Reinforcement Learning, Siamese Networks, Autoencoder, Network Accuracy, Action (Embedding) Space, Policy, Q-Value (Q-Function), Twin Delayed Deep Deterministic Policy Gradient (TD3), Actor-Critic Methods, and Policy Gradient with respect to the Embedding space.

Chapter 4 presents an overview of the proposed architecture and delves into various implementation-related aspects of key network components, such as the 'encoder model', 'decoder model', 'accuracy model', 'actor network', 'critic network' and 'reinforcement learning agents'. Chapter 5 emphasizes various experimentation-related aspects, including the training of the Embedding Space, pre-training and full training of Actor and Critic components and finally actor-critic training with reinforcement. In Chapter 6 we share results of various experiments using our proposed architecture.

Lastly, Chapter 7 concludes the dissertation by summarizing the innovative deep learning framework that employs Reinforcement Learning to learn a network architecture modification policy. This approach holds the potential to enable network optimization for diverse problems without necessitating substantial and expensive retraining, ultimately summarizing the results.

CHAPTER 2

RELATED WORK AND BACKGROUND

2.1 Related Work

Neural network architectures have been the driving force behind numerous breakthroughs in deep learning, leading to significant advancements in computer vision, natural language processing, and other domains. The design of these architectures has traditionally been a manual and labor-intensive process, prompting the development of automated methods that use approaches such as reinforcement learning (RL) for optimizing network architectures. The growing complexity of neural network architectures in deep learning necessitates efficient optimization methods to enhance their performance and to make them accessible and usable by a wider range of users.

Deep learning techniques have facilitated the creation of intricate neural network architectures, achieving state-of-the-art performance across various tasks [30]. However, conventional architecture optimization methods, which largely consist of trial and error and rely heavily on the expertise and intuition of the expert designer, can be time-consuming and labor-intensive. Reinforcement learning (RL) has emerged as a promising approach for automating network architecture optimization through intelligent decision-making due to its ability to learn in situations where the correct solution is not known for a sufficiently large number of training instances. Numerous past research efforts have investigated the application of reinforcement learning (RL) for network architecture optimization, examining its potential benefits and challenges.

Moreover, some advances in deep learning illustrate the complex representation learning capabilities of multi-layered architectures Figure 2.1. Neural networks assume certain preconditions regarding the type and distribution of training data. Specifically, the provided training data should be Independent and Identically Distributed (IID) with a fixed underlying distribution and necessitate complete supervision in the form of ground truth or complete gradients concerning individual instances in the data. These requirements are often unattainable for many learning problems, thus limiting the application of neural networks in various problem domains where such uncertainties might emerge in real-world data.

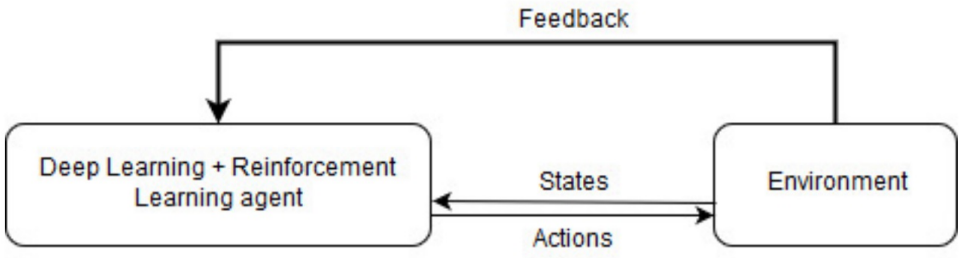


Figure 2.1: DeepRL Architecture

This section explores the application of RL in network architecture optimization, discussing the benefits, challenges, and methodologies involved. It also provides a comprehensive review of research in the field of neural network architecture optimization using RL, examining key milestones, methodologies, and challenges. By integrating RL into the optimization process, we can potentially develop more efficient and robust models, enabling advancements in various fields, including computer vision, natural language processing, and beyond. In this sec-

tion we highlight pertinent literature and emphasize key milestones encompassing primary methodologies, challenges, and future directions in the field in order to provide an understanding of the current landscape of neural architecture optimization.

2.1.1 Early works on Neural Architecture Search using RL

2.1.1.1 Reinforcement Learning in Neural Architecture Search (NAS)

Neural Architecture Search (NAS) is a widely adopted approach for automating network architecture optimization, aiming to identify the optimal architecture for a specific task by exploring an extensive space of possibilities [23]. NAS employs various search strategies, including random search, evolutionary algorithms, and Bayesian optimization. In recent years, Reinforcement Learning has emerged as a promising option in NAS [17] [31] due to its efficient learning abilities in the absence of labeled data.

Zoph and Le [23] were among the first to propose using reinforcement learning for NAS. Their work introduced an innovative method for employing RL to search for optimal neural network architectures. The authors utilized a recurrent neural network (RNN) controller to generate variable-length strings encoding the architecture of a neural network, thereby helping to generate architectural configurations for convolutional neural networks (CNNs). The controller was trained using a policy gradient method to maximize the expected accuracy of the generated architectures on a validation set. The performance of these generated architectures served as a reward signal to train the controller using policy gradient methods [32]. Their approach led to the discovery of novel architectures that outperformed hand-designed architectures on image classification tasks.

Following the success of Zoph and Le [23], several other works have applied reinforcement learning to NAS, including Baker [17], who employed Q-learning [33] to search for architectures, and Tan et al [34], who used Proximal Policy Optimization [35] to optimize architectures for mobile devices. While these approaches were a large step forward and pointed out a promising avenue, they were limited to single problems and specific architectures and required very large numbers of experiences on each individual target problem.

In reinforcement learning-based approaches the problem of network architecture optimization is generally formulated as a Markov Decision Process (MDP), in which an agent interacts with an environment over a series of discrete time steps [36]. In the context of Neural Architecture Search (NAS), the agent is responsible for generating architectural configurations, while the environment corresponds to the performance of the generated architectures on a given task. The agent's goal is to discover an architecture that maximizes its expected cumulative reward, which is the sum of the rewards obtained over time. One of the main benefits of this approach is that it only requires the availability of a reward function, which is generally much easier to design and estimate than a complete performance function for each network. Moreover, its ability to construct a utility function automatically allows it to address the problem of excessive local extrema in the original performance space.

Various RL algorithms have been employed for NAS, including policy gradient methods [32], Q-learning [33], and actor-critic algorithms [37]. Policy gradient methods directly optimize the parameters of the policy, whereas Q-learning and actor-critic algorithms aim to learn value functions that estimate the expected cumulative reward. In recent years, advanced RL algorithms, such as Proximal Policy Optimization [35], have also been applied to NAS, providing improved sample ef-

ficiency and stability [34]. Another direction in which advances have occurred is the development of strategies to reduce the search space and computational cost associated with training and evaluating candidate architectures.

2.1.1.2 Designing Neural Network Architectures Using Reinforcement Learning

In citebaker2016designing, Baker and colleagues presented a meta-learning framework for designing neural network architectures using reinforcement learning. The authors used a two-level hierarchical approach, where the first level optimized the macro-level architecture, and the second level focused on micro-level details such as layer types and hyperparameters. In this approach agent is limited to a finite state and action space and constricted the state-action space using coarse, discrete bins to accelerate convergence.

2.1.1.3 Advances in Efficient Neural Architecture Search

Pham et al.[38] introduced an efficient approach to neural architecture search by sharing parameters across different architectures, which significantly reduced the computational requirements. The method used a single, large model that subsumed multiple sub-networks, allowing the simultaneous evaluation of multiple architectures. Efficient NAS (ENAS) [38] is an example of such an approach. It employs a weight-sharing strategy between different architectures, allowing for faster evaluation and training. By sharing weights, ENAS significantly reduces the computational requirements without compromising the quality of the discovered architectures. This has led to state-of-the-art results on benchmark datasets, such as CIFAR-10 and ImageNet [5], [39]. However, Efficient NAS (ENAS) [38] suffer from a low-rank restriction as for arbitrary matrices A and B, one always has the

inequality: $\text{rank}(A \cdot B) \leq \min \text{rank}(A), \text{rank}(B)$. Due to this limit, it fails to find architectures that perform well in the normal training setups, where the weights are no longer restricted.

2.1.1.4 Differentiable Architecture Search (DARTS)

Another area of focus is incorporating domain knowledge into the search process to guide the exploration of the architecture space. In the Differentiable Architecture Search (DARTS)[24] framework, the search process is formulated as a continuous optimization problem, allowing the use of gradient-based optimization methods. This differentiable approach reduces the search time and computational resources needed while still identifying high-quality architectures [40].

2.1.1.5 Transfer learning

Transfer learning has also been explored as a means to improve the efficiency of RL-based NAS methods. For instance, MetaQNN [17] leverages meta-learning to transfer knowledge from previous tasks, thus reducing the time required to search for optimal architectures for new tasks. This approach demonstrates the potential of using prior knowledge to accelerate the search process and improve the performance of discovered architectures [41].

2.1.1.6 Proxy-Less-NAS

Cai [29] and colleagues proposed a direct neural architecture search on the target task and as well as hardware to accelerate it that eliminated the need for proxies, such as training on smaller datasets or model simplification. Instead, the

method directly optimized the architecture's performance on the target task and hardware, which led to more accurate and efficient architectures.

2.1.1.7 Exploration vs. Exploitation

Balancing exploration and exploitation is crucial in RL-based NAS due to the large size of the space of neural networks. Exploration involves generating diverse architectural configurations to discover novel architectures, while exploitation focuses on refining the current best-performing architectures. Striking the right balance between exploration and exploitation is essential to ensure the RL algorithm converges to an optimal architecture [36].

Various techniques have been employed to balance exploration and exploitation in RL-based NAS. For example, Zoph and Le [23] used an ε-greedy exploration strategy, which involves taking random actions with probability ε and taking the best-known action otherwise. Alternatively, entropy regularization can be employed to encourage exploration where not sufficient information is available by adding the entropy of the policy to the objective function with a regularization coefficient [42].

In summary, the application of reinforcement learning in neural architecture search has led to significant advancements in automating the discovery of high-performing neural network architectures. By employing various RL algorithms and incorporating strategies like weight-sharing, differentiable search, and transfer learning, researchers continue to push the boundaries of NAS, enabling advances in various fields, such as computer vision, natural language processing, and beyond. As the field of NAS progresses, including through the work presented in this disserta-

tion, we expect to witness further developments in RL-based techniques and novel approaches to improve the efficiency and effectiveness of architecture search.

2.1.2 Closely Related Methods

Many approaches exist for network architecture optimization using Reinforcement learning. All these approaches propose an automatic method for the search of an optimized neural network architecture given a specific task. Luo[1] allows the exploration of a multidimensional embedding space of possible structures, including the choice of the number of neurons, the number of hidden layers, the types of synaptic connections, and the use of transfer functions. Luo[1] has also introduced a novel strategy which is capable to generate a network topology with over-fitting being avoided in the majority of the cases at affordable computational cost. However, the used architecture optimization is a combination of local gradient ascent and significant heuristic search, limiting optimization to a single learning task and incurring significant computational expenses.

Alternatively Bose[2] has taken a different approach focusing on ensembles of simple networks that uses Reinforcement learning to learn a policy to incrementally build neural network classifiers for a broad distribution of problems, and subsequently applied it to new data to learn a classifier for a specific new problem. Bose[2] has used state to represent an accuracy of a network, which is problem-independent and thus allows learned policies to be applied to new problems. However, Bose[2] uses a very small, pre-defined set of network modification operations, limiting it to ensembles of simple fully connected binary and one-vs-all classifiers [43] and making it hard to expand to other network architectures such as convolutional or recurrent networks.

2.1.3 Challenges and Future Directions

One of the main challenges of RL-based NAS is the high computational cost associated with training and evaluating numerous architectures. This issue has led to the development of techniques to reduce the computational burden, such as weight sharing [38] and early stopping [44]. However, further research is needed to develop more efficient algorithms for RL-based NAS.

Another challenge is leveraging the knowledge gained from one task to optimize architectures for other tasks, which is known as transfer learning. Some work has been done in this area, such as the use of meta-learning for NAS [17] and the progressive search for architectures [45]. However, more research is needed to improve the effectiveness of transfer learning in RL-based NAS.

In addition to optimizing the architecture itself, RL could potentially be applied to other aspects of network optimization, such as network pruning and quantization. These techniques aim to reduce the model size and computational complexity without significantly degrading performance [46]. Exploring the integration of RL with these techniques is a promising direction for future research.

We can summarize the challenges into following three sections:

2.1.3.1 Scalability

A key challenge in neural architecture optimization using RL is scalability, as the search space grows exponentially with the number of layers and possible layer configurations.

2.1.3.2 Transferability

Another challenge is the transferability of learned architectures across different tasks and datasets, which could help alleviate the need for expensive architecture searches for every new problem.

2.1.3.3 Hardware-aware optimization

The increasing importance of efficient deployment on diverse hardware platforms motivates the development of hardware-aware optimization methods, which directly incorporate hardware constraints into the search process.

2.2 Background

Network Architecture Optimization (NAO) plays a crucial role in improving the performance of deep learning models by systematically exploring and refining the architecture space. It focuses on discovering optimal network architectures that yield the best performance for a given task. NAO techniques involve various search strategies, such as random search, evolutionary algorithms, Bayesian optimization, and more recently, Reinforcement Learning.

In this dissertation we present an approach that employs Reinforcement Learning in a network embedding space to learn a policy capable of producing optimized network architectures. In our approach, Siamese Networks are employed to compress legal networks within the embedding space, allowing the system to efficiently explore and optimize network architectures. This embedding space transforms the space of network architectures into a lower-dimensional continuous space, thereby defining a continuous action space. Reinforcement Learning is later

applied to the network embedding space to learn policies capable of generating optimized network architectures.

Here a value function estimates the expected cumulative reward that an agent would receive after taking an action in a given state and following a specific policy, guiding it towards finding optimized network architectures. To enable generalization across target problems and facilitate planning, the system also incorporates a feature vector encoding target problem complexity and a derived performance prediction component [16].

To better understand our proposed architecture, it is essential to first examine the roles of the different components, including Siamese Networks, Reinforcement Learning, Ensemble Learning, Action Space, Policy, and Q-Value as well as the Advantage value in relation to the embedding space.

2.2.1 Network Architecture Optimization (NAO)

2.2.2 Neural Networks

Artificial neural networks are a powerful modeling technique that can be used for representation learning on complex high dimensional input data. Recent advances in multi-layered neural networks or Deep learning mechanisms [1] illustrate the effectiveness of such algorithms at discovering intricate structures in high dimensional data extending their application domains to many real world problems.

Deep learning techniques have made major advances in solving complex supervised learning problems in science, business and government applications, ranging from image recognition [5], speech recognition and language modeling [3] [47], predicting the behavior of new drug molecules [48], and modeling biological systems as in reconstructing brain circuits [49] and predicting effects of gene

mutations [50] [51]. Neural networks are capable of modeling arbitrarily complex functions in high dimensional continuous spaces and have been shown to generalize well for unseen data far from the training input.

2.2.2.1 Recent Advancements and Challenges

It is widely recognized that neural network models do not inherently favor discovering the underlying structural information of the data, which could enhance the expressive power of the learned representations. This open research area has seen various architectural constraints proposed for the hidden representations in the form of lateral or recurrent connections among nodes. However, neural networks trained with traditional backpropagation algorithms and their variants, which offer some convergence guarantees for standard feed-forward networks, often fail to converge when constrained by such recurrent connections.

In order to address these challenges and limitations, researchers have been exploring alternative methods and techniques that can potentially improve the learning capabilities of neural networks. One such approach is the incorporation of reinforcement learning (RL) techniques for network architecture optimization. By integrating RL into the optimization process, it is possible to create more efficient and robust models that can handle complex tasks and adapt to changing data distributions.

Another promising direction is the use of Siamese networks for representation learning. These networks have demonstrated their ability to learn robust and discriminative representations by considering the similarities and differences between input pairs or triplets. This can lead to improved generalization and potentially better performance on various tasks.

Additionally, exploring ensemble learning techniques can further enhance the performance and robustness of neural network models. By combining the predictions from multiple diverse models, ensemble methods can reduce the impact of individual model biases and overfitting, resulting in improved overall performance.

Finally, the development of more efficient training methods and algorithms is essential for addressing the challenges associated with large-scale and complex neural network architectures. Novel optimization techniques, regularization methods, and learning rate schedules can contribute to faster convergence and better generalization, enabling neural networks to tackle a wider range of problem domains.

By addressing these challenges and exploring these research directions, the field of neural networks can continue to advance, yielding more powerful and versatile models capable of tackling complex tasks and adapting to the uncertainties present in real-world data.

2.2.3 Siamese Networks for Robust and Efficient Feature Space Learning

Siamese networks have emerged as a powerful deep learning technique for learning feature representations in tasks that involve comparing and relating different data samples, such as image recognition, signature verification, and one-shot learning [52]. In this section, we provide an introduction to Siamese networks, focusing on their architecture, learning process, and applications. We also discuss recent advancements and challenges in Siamese network research, highlighting potential avenues for future exploration.

Deep learning has revolutionized various domains, such as computer vision, natural language processing, and speech recognition. One of the key challenges in

these domains is learning meaningful feature representations from the limited set of training data. More and more data is needed at times by Neural networks to perform with high accuracy. However, at times we do not have luxury to have enough data to train Neural networks to perform with high accuracy. To solve these kinds of tasks siamese networks play a very important role [52]. Also, Siamese networks have demonstrated remarkable performance in tasks that involve comparing and relating different data samples.

2.2.3.1 Siamese Network Architecture

A Siamese network consists of two or more identical sub-networks, each having the same architecture and sharing weights. These sub-networks, called "sibling networks," process the input data samples independently and produce feature representations. The outputs of the sibling networks are then combined and passed through a comparison function, which measures the similarity or dissimilarity between the feature representations as shown in Fig 2.2.

2.2.3.2 Learning Process in Siamese Networks

Siamese networks are trained using a process called contrastive learning. During training, the network is fed with pairs of input samples, along with a label indicating whether the samples belong to the same class or different classes. The network learns to generate feature representations that minimize the distance between samples of the same class and maximize the distance between samples of different classes.

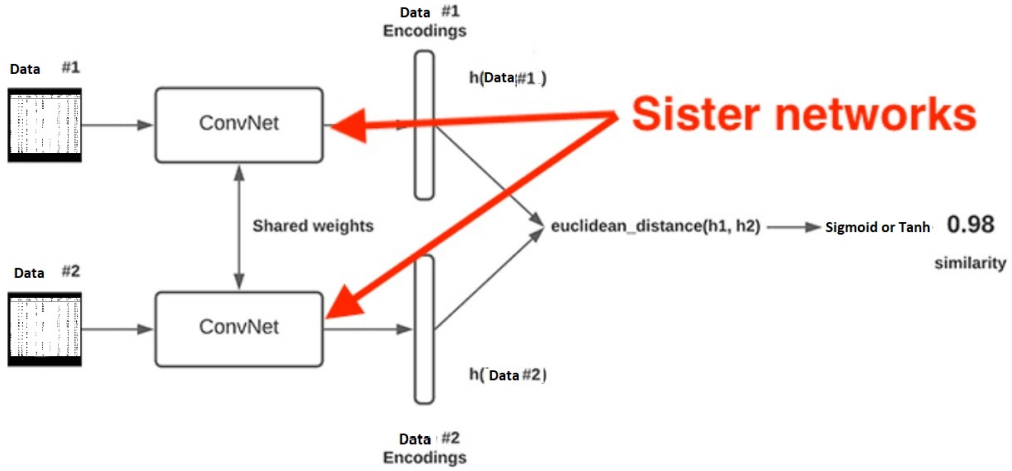


Figure 2.2: Siamese Network Architecture
Ref: https://pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/

2.2.3.3 Applications of Siamese Networks

Siamese networks have been successfully applied to various tasks, including:

One-shot learning: One-shot learning aims to recognize new objects based on only a few examples. Siamese networks have demonstrated impressive performance in this task, as they can learn to generalize from a small number of samples [53].

Signature verification: Siamese networks have been used to verify signatures by comparing the feature representations of a reference signature and a test signature. The network learns to identify genuine and forged signatures with high accuracy [54].

Image recognition: In image recognition tasks, Siamese networks have been employed to learn robust feature representations that are invariant to variations in lighting, pose, and other factors that affect the appearance of images [55].

2.2.3.4 Summary

Siamese Neural Networks (SNNs) can be described as a category of neural network architectures that comprise two or more identical sub-networks. The term "identical" refers to the fact that these sub-networks have the same structure, parameters, and weights. Parameter updates are synchronized across all sub-networks, as they are employed to identify similarities between inputs by comparing feature vectors.

Siamese networks are known for their ability to yield accurate predictions using only a few images or a small amount of data. This capacity to learn from limited data has made Siamese networks increasingly popular in recent years. As the need for robust and efficient feature learning techniques continues to expand, Siamese networks present significant potential for advancing deep learning performance across various domains. My research aims to investigate and effectively utilize Siamese networks that reduce the distance between samples (legal/illegal networks) belonging to the same class while increasing the distance between samples of different classes.

Building on the capabilities of Siamese networks, further research can explore various methods to enhance their performance and adaptability. By developing more efficient training techniques and addressing challenges related to scalability and overfitting, Siamese networks can be tailored to suit a wider range of applications.

Moreover, exploring the integration of Siamese networks with other deep learning architectures and techniques could uncover new possibilities for improved representation learning. For instance, investigating the impact of various architectural constraints on hidden representations, such as lateral or recurrent connections

among nodes, may reveal novel ways to enhance the expressive power of learned representations.

Recent advancements in Siamese network research encompass the development of triplet loss, which extends the contrastive learning process by incorporating an additional negative sample, and the introduction of self-supervised learning methods that leverage Siamese networks for unsupervised feature learning. However, challenges persist, such as enhancing the scalability of Siamese networks, addressing overfitting issues, and developing more efficient training methods.

2.2.4 Ensemble Learning

Ensemble learning algorithms are often designed for problems involving large datasets with concept drift properties. These algorithms typically maintain a fixed ensemble size or employ evolutionary algorithms to construct ensemble networks [56],[57] ,[18]. However, this approach necessitates randomly re-exploring valid architectures from scratch for each new problem, significantly limiting the applicability of such algorithms. Additionally, integrating structural information often requires manual incorporation of context into the training data.

In the context of ensemble learning algorithms, addressing challenges associated with incorporating structural information, scalability, and optimization remains paramount. While existing methods have demonstrated some success, they often require manual intervention and hand-engineered solutions, which can limit their applicability and effectiveness.

2.2.5 Reinforcement Learning: An Overview of Techniques and Applications

Reinforcement learning (RL) is a subfield of machine learning that focuses on training intelligent agents to make decisions by interacting with an environment.

It has been successfully applied to a wide range of applications, from robotics to finance. Reinforcement learning (RL) is a machine learning paradigm that allows agents to learn how to make decisions and take actions by interacting with an environment. Unlike supervised learning, RL does not rely on labeled data; instead, agents learn from trial and error, receiving feedback in the form of rewards or penalties.

2.2.5.1 Fundamentals of Reinforcement Learning

The theory of sequential decision problems includes formulations of both deterministic and stochastic problems [51]. In such problems an agent interacts with a discrete time stochastic dynamical system by observing the current system state and selecting an action at each time step. Sequential decision problems are popularly represented by the mathematical framework of Markov Decision Processes (MDP) in stochastic domains [58].

A Markov Decision Process is defined as a tuple $\langle$ S, A, Ψ, P, R $\rangle$, where S is a finite set of states, A is a finite set of actions, $\Psi \subseteq S \times A$ is the set of admissible state-action pairs. $A_s = a|(s, a) \in \Psi \subseteq A$ defines the set of actions admissible in state s, assuming that $\forall s \in S$, A_s is non-empty. $P : \Psi \times S \mapsto [0, 1]$ is the transition probability function with P(s, a, s′) being the probability of transition from state s to state s′ under action a where s, s′ $\in$ S and a $\in A_s$. $R : \Psi \mapsto \mathbb{R}$ is the expected reward function, with R(s, a) being the expected reward for performing action a in state s.

Addressing an MDP entails learning a policy, π, that maximizes an expected utility derived from the rewards. Solving such problems requires learning from sparse delayed feedback in stochastic dynamical environments. Reinforcement

learning [59],[36] is a learning paradigm that acquires control policies without the need for extensive outside supervision. This formulates a framework for reactive control which learns from interactions with the environment utilizing supervision provided in the form of simple, scalar rewards and punishments defined by the learning task (Figure 2.3).

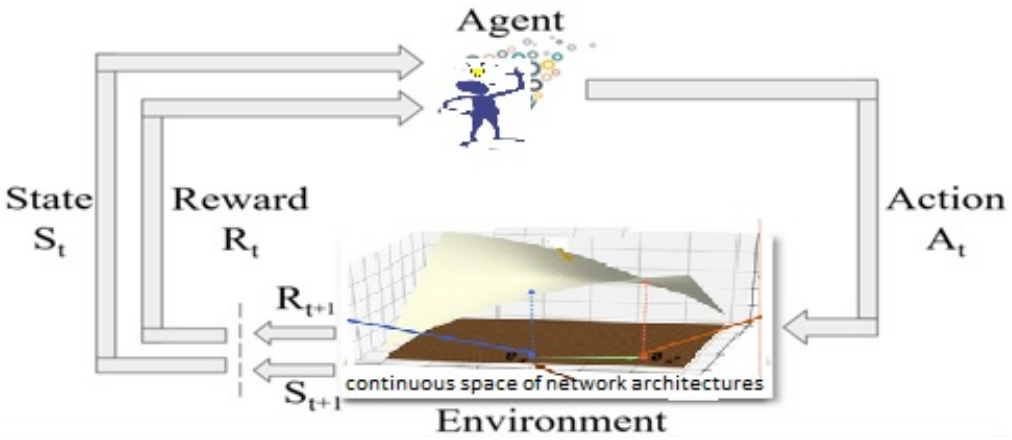


Figure 2.3: Reinforcement Learning With Embedding Space (Part of the figure borrowed from [1])

Neural networks and reinforcement learning algorithms have been combined to form a powerful learning tool capable of modeling various uncertainties in observed data. Typically, these algorithms are restricted to the domain of sequential decision-making. Neural networks are employed to learn complex representations from continuous, high-dimensional datasets, while reinforcement learning algorithms model sparse feedback and address problems in dynamic environments.

Recent developments in neural networks and reinforcement learning algorithms have opened up new avenues for tackling these challenges. By combining the strengths of these two learning paradigms, researchers have been able to create

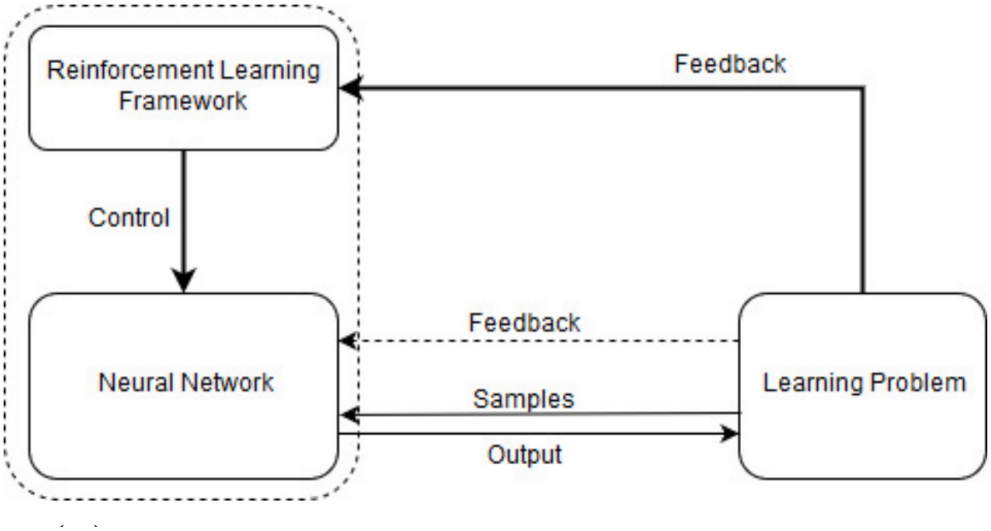


Figure 2.4: Overview of Reinforcement Learning Agent

powerful tools that can effectively model uncertainties in observed data and adapt to dynamic environments.

Bose's work [2] showcases the potential of reinforcement learning techniques when applied to problem domains characterized by uncertainty. However, Bose's approach relies on predefined state spaces and separates the action space and state space, resulting in a less integrated solution. Unlike Luo's [1] approach, which establishes a direct relationship between action space and state space, Bose's state space is a vector of accuracies indicating how well a particular network is currently performing on a given problem (Figure 2.4). Using MDP, Bose calculates accuracies for a predefined state space and effectively employs three state spaces to construct three separate entities. In contrast, Luo's method [1] employs a single POMDP for all network modifications, allowing for a more unified approach that directly relates action space and state space. This approach enables greater flexibility and adaptability when addressing various machine learning problems.

In conclusion, both Bose's and Luo's work [1] demonstrate the value of incorporating reinforcement learning techniques in the context of ensemble learning algorithms and neural networks. As research in this area continues to progress, it is likely that more advanced and integrated solutions will emerge, further pushing the boundaries of deep learning performance and applicability in a wide range of problem domains.

The main advantage of Reinforcement learning is that it does not require a priori datasets and pre-determined labels to train the agent, and can thus be applied to problems where the solution is not known and can not be easily calculated. The agent traverses through a Markov Decision Process (MDP) to find the best sequence of decisions to maximize the reward, thus training itself through trial and error. It is the learned policy which determines the best sequence of decisions which maximizes the accumulated reward and overall performance. Hence, the agent collects a trajectory using its current policy and uses it to update the policy parameter.

In our proposed architecture, or example, the actor generates actions in a learning environment. By using the performance feedback from a critic, the agent then must find the best possible learning path in the embedding space. Therefore to get the best performance and better future predictions, with the help of RL, we need to make temporal changes to be able to predict, not what is at the end of this step but what is reachable at the end of a trajectory of steps in the right direction.

2.2.5.2 Key Techniques in Reinforcement Learning

Value-based Methods: Value-based methods, such as Q-learning and Deep Q-Networks (DQNs), focus on learning the value of taking an action in a state,

represented by a state-action value function (Q-function). These methods aim to find the optimal policy by maximizing the Q-function. Further, most reinforcement learning algorithms can be categorized into offline and online learning algorithms. The Q-learning algorithm [33] is an offline model-free reinforcement learning approach and is a form of asynchronous Monte-Carlo dynamic programming. Here, the utility of a state action pair depends on the current observed reward and the utility of the greedy action choice for the next observed state. Given a state transition tuple $\langle s_t, a_t, s_{t+1}, r_t \rangle$, the Q-learning update step can be defined as :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma \underset{a}{max} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{2.1}$$

Offline Reinforcement learning approaches have the benefit that they can learn from existing data without the need to explicitly execute the policy to be learned. On the other hand, online learning algorithms have several advantages over offline learning and are potentially more robust to errors or omissions in the training set [60]. Instead of selecting the greedy action for the utility, online algorithms choose the next state action pair according to the current policy which allows online learning systems to operate largely without requiring the storage of previously observed state action utilities. The SARSA algorithm [61], [62] is a modified connectionist Q-learning algorithm, where the update step for a given experience tuple $\langle s_t, a_t, s_{t+1}, r_t \rangle$, s defined as:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_t + \gamma Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{2.2}$$

Policy-based Methods: Policy-based methods, such as REINFORCE and Proximal Policy Optimization (PPO), directly optimize the policy without relying on a value function. These methods use gradient ascent to update the policy parameters in the direction that maximizes the expected cumulative reward. Additionally, re-

inforcement learning algorithms allow parameterized policies by applying Actor Critic techniques as policy gradient algorithms [37]. Here, a policy function is used to model the strategy and updated by estimating a gradient which optimize the utility. Allowing step updates and assuming an uncertain problem domain allow for robust learning algorithms that are suitable for solving many decision making problems in the real world. Although rare, reinforcement learning algorithms have also been applied to problems in other learning domains. For example, in clustering problems [63] where the clustering problem is formulated as a reinforcement learning problem utilizing the quality of the cluster as feedback.

The objective of Reinforcement learning is to learn a good decision-making policy π that maximizes rewards over time. The performance function is what determines the policy, which is nothing but the argmax over all possible gradient directions of the performance function.

In reinforcement learning, a policy (π) is a function that maps from states to actions, determining the best action to take in a given state. When considering small changes in action (Δa) and a performance function, we can define the policy as follows:

Given a state 's' and an action 'a', the performance function evaluates the effectiveness of taking action 'a' in state 's'. If the change in action (Δa) is small enough, we can write the policy (π) in terms of the performance function:

$$\pi(s) = argmax_a Performance(s, a + \Delta a) \tag{2.3}$$

This equation states that the policy selects the action 'a' that maximizes the performance function when considering small changes in action (Δa). The policy aims to find the best action to take in state 's' by considering the performance

of slightly perturbed actions around 'a' and choosing the one that results in the highest performance.

Actor-Critic Methods: Actor-critic methods combine aspects of both value-based and policy-based methods. They employ two components: an actor, which represents the policy, and a critic, which estimates the value function. The critic is used to guide the learning process of the actor, improving the stability and convergence of the learning process.

Luo[1] uses predetermined state and action space, where action space causes move in the state space but not a step in that state space. Whereas in case of Bose[2] the action can be 'add me a layer' which will change the accuracy compared to Luo[1] where action is to move a step in the state space. Bose[2] is deploying strict Q-learning therefore by adding an extra layer he can achieve a change in an accuracy of a network. His performance function is task independent and can be defined as:

$$\text{Performance}(s,a) = \text{Expected value of increase in accuracy}$$
$$= E(\text{increase in Accuracy})$$

Exploration vs. Exploitation: One of the key challenges in reinforcement learning is balancing exploration and exploitation. Exploration involves taking random actions to gather information about the environment, while exploitation entails using the current knowledge to choose the action that maximizes the expected reward. Striking the right balance is crucial, as too much exploration may lead to suboptimal performance, while too much exploitation can result in the agent missing out on potentially better actions.

Challenges: Like many learning algorithms, reinforcement learning is also susceptible to the curse of dimensionality while working on many real world problems that are usually represented in high dimensional and potentially continuous domains.

The reinforcement learning framework defines various methods for addressing such issues. For problems with continuous spaces, the states are often aggregated to form an approximate factored representation of the space [64][62] [65]. In such cases the representations used need to respect the underlying dynamical system. Apart from discretized state spaces, a latent representation is often learned for the observed system. Neural networks are most commonly combined with reinforcement learning algorithms to allow decision making and perform value function approximations in various real world problems with high dimensional input representations.

It has been well established that reinforcement learning is a powerful and versatile machine learning paradigm that has shown great promise in a wide range of applications. By allowing agents to learn from trial and error while interacting with their environment, RL provides a flexible approach to solving complex decision-making problems. Future research in reinforcement learning will likely focus on addressing challenges such as sample efficiency, transfer learning, and multi-agent scenarios. Moreover, reinforcement learning will continue to integrate with other machine learning approaches, such as deep learning and unsupervised learning, to further push the boundaries of artificial intelligence.

As RL algorithms continue to improve and become more accessible, it is expected that their adoption across various industries will increase, leading to innovative solutions and transformative breakthroughs in areas such as healthcare, transportation, and entertainment. Reinforcement learning has the potential to significantly impact our world, providing intelligent agents capable of tackling some of the most complex and challenging problems we face today [36].

It is very effectively used in my research as it allows to learn from quantitative feedback and thus addresses the problem in network architecture optimization that

optimal network configurations for the target problems generally are not known. Utilizing the network embedding space, RL operates here by executing actions in embedding space to modify the network architecture and observing resulting reward in the form of changes in actual or predicted network performance. Based on this, a value function is estimated and a policy is learned that optimizes the obtained rewards.

In Summary Reinforcement Learning can be defined as:

1. Agent takes actions and learns from quantitative feedback
2. It does not require datasets and pre-determined labels to train agents
3. Agent/Environment interaction modeled as a Markov Decision Process
4. Learned policy determines best sequence of decision that maximizes the reward and overall performance.

2.2.6 Learning a policy (π)

Learning a general policy for a wide range of problems and then constructing a network for a specific problem may not be the most optimized approach. Instead, there is a need to learn a policy that dictates the appropriate action to take in each state to maximize a particular function. This function represents the mean or expected discounted sum of the sequence of rewards, leading to an optimized network for various problems. Bellman's equation can be used to formalize this concept, and it can be solved iteratively through policy iteration. This results in the unique fixed point of the equation, which corresponds to the optimal policy.

Bellman's equation effectively decomposes the value function into two parts: the immediate reward and the discounted future values [66] Figure 2.6. This simplifies the computation of the value function, allowing the optimal solution to

be found by breaking complex problems down into simpler, recursive sub-problems and solving them optimally.

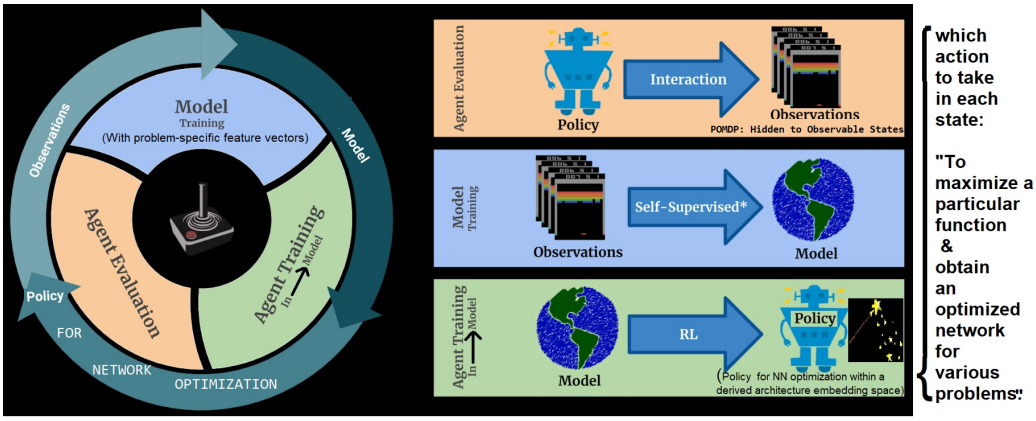


Figure 2.5: Overview of Policy : 1) The agent starts interacting with the embedding environment. 2) The collected observations are used to update the current model. 3) The agent updates the policy by learning inside the model.

Another constraint we face to get to optimal policy is that we don't have complete visibility of the state space, which makes us hard to use MDP, as Markov Decision Process (MDP) assumes that the complete state of the world is visible to the agent. This is clearly highly unrealistic as there is no way to find out what is input and output space for a specific problem will be. To resolve this constraint, POMDPs model effectively uses the information available to the agent by specifying a function from the hidden state to the observables.

By employing POMDPs, the objective becomes more straightforward: to find a mapping from observations (rather than states) to actions. For instance, in a classification problem, a policy (f) can be represented as f(x), where f(x) = y. Here, x is a state (s), and y is the output decision of the policy. In real-world problems, the input may not be a known state (x); instead, it can be substituted by an observation

$X \equiv O$ (Observations). Alternatively, the state may be inferred or embedded in some way.

The primary goal is to learn a strategy (a policy) for network optimization within a derived architecture embedding space. This policy will specify which action to take in each state to maximize a particular function and obtain an optimized network for various problems. The focus is not on finding the best network for all problems; rather, the aim is to encode a policy that helps determine the best network for a given problem. The proposed architecture attempts to estimate the Q-value for any state comprising an embedding point and problem-specific accuracy metrics (Figure 2.7). Conceptually, the policy is the gradient of the Q-function with respect to the embedding space, which provides the direction of movement and thus how to modify the network. In the experiments, the policy and value function take the form of a deep network actor-critic architecture. The approach presented here utilizes an actor-critic reinforcement learning method [37] [67] to perform policy and value function learning in a continuous embedding space augmented with a continuous action space. Additionally, problem-specific parameters (feature vectors) are incorporated into the network, such as network accuracy for a given problem.

The actor-critic reinforcement learning approach allows for more efficient learning and optimization in the continuous embedding space. By incorporating problem-specific parameters like network accuracy, the proposed architecture is better equipped to adapt to various problem domains. This combination of the continuous action space, deep network actor-critic architecture, and problem-specific feature vectors provides a powerful solution to address the challenges of learning a policy for network optimization.

In summary (Fig.2.5), learning a policy in reinforcement learning involves several crucial components:

1. Utilizing Bellman's equation to simplify the computation of the value function by breaking down complex problems into simpler, recursive sub-problems.
2. Employing POMDPs to model the relationship between hidden states and observable states, enabling the agent to make informed decisions based on available information.
3. Leveraging a deep network actor-critic architecture to learn the policy and value function in a continuous embedding space augmented with a continuous action space.
4. Incorporating problem-specific feature vectors into the network to facilitate adaptation and optimization for various problem domains.

By integrating these elements, the proposed architecture can effectively learn a policy that maximizes the chosen function, resulting in an optimized network for a wide range of problems. This approach holds significant potential for improving the performance and adaptability of deep learning models, ultimately pushing the boundaries of what can be achieved with reinforcement learning-based network optimization.

2.2.7 Q-Value (Q-Function)

In Reinforcement learning, the Q-Value (Eq. 2.1) estimates performance as the expected utility of the learning agent in state **s** when performing action **a**, and

otherwise following policy(π). This value function not only considers immediate reward feedback, $r(s,a)$, but also takes into account future payoffs.

$$Q_\pi(s,a) = \sum_{s'} P^a_{ss'}(r(s,a) + \gamma \sum_{a'} \pi(a'|s').Q_\pi(s',a') \tag{2.4}$$

$$V_\pi(s) = \sum_{a} \pi(a|s). \sum_{s'} P^a_{ss'}(r(s,a) + \gamma.V_\pi(s')) \tag{2.5}$$

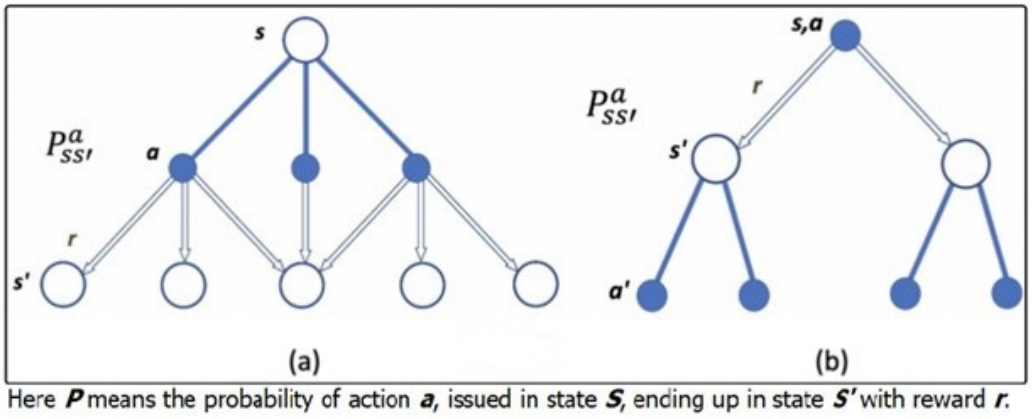


Figure 2.6: Bellman equation (a) for the state-value function, (b) for the action-value function

The Bellman equation [66] for the State-value function provides the basis for performance estimation and policy learning and tells us how to find the value of a state (Fig. 2.6) following a policy (Eq. 2.5). Note, Q-Learning is a very effective way to evaluate the performance of an action in a particular state but Luo[1] is not using Q-Learning and is directly evaluating his performance function (Eq.2.1) under supervised learning. His Network Architecture Optimization (NAO) is truly an episodic task where Q value and reward are identical Q(s) = r(s). It is a single step problem, where Luo[1] is using supervised training to train it's architecture until they find its full reward, which tells us how high its performance is? In contrast

this is not true in case of Bose[2] as data is a complete network and there is no next step, unlike Luo[1] where, we sample the neural network space in the local neighborhood where we expect the solution to be. Therefore, for Luo[1] Q-function effectively is discounted sum of the rewards as same step in different problems leads to different improvements.

In our application the performance function must be somewhat problem independent, making a general utility function such as the Q-Function. The Q-Function (Eq. 2.2) is the total discounted sum of future rewards, thus taking into account not only immediate payoffs but also future effects of actions. Also reward represents increase in performance achieved by a network modification action and the Q-value tries to predict expected improvements achievable by policy π. The state is provided by concatenating the embedding vector (representing the current network architecture) and a problem specific feature vector (capturing problem complexity related information), while possible actions are continuous displacement vectors in the embedding space. Fig.2.7 shows the connection of the embedding space and the value function.

To utilize this in our application where we want to learn to optimize network architectures in an embedding space, we need to decide two important factors:

1. How to define the reward and action space used by RL?
2. What are the problem specific parameters that we can give as an additional input to allow generalization across learning problems ?

Since we don't have a fixed performance value at every point in the state space, we will use Reinforcement learning to train and calculate a Q-Value for all local modifications. This way we train a policy on the network space and after n iterations

we may end up at the goal, and along the way we will learn values for the entire state space.

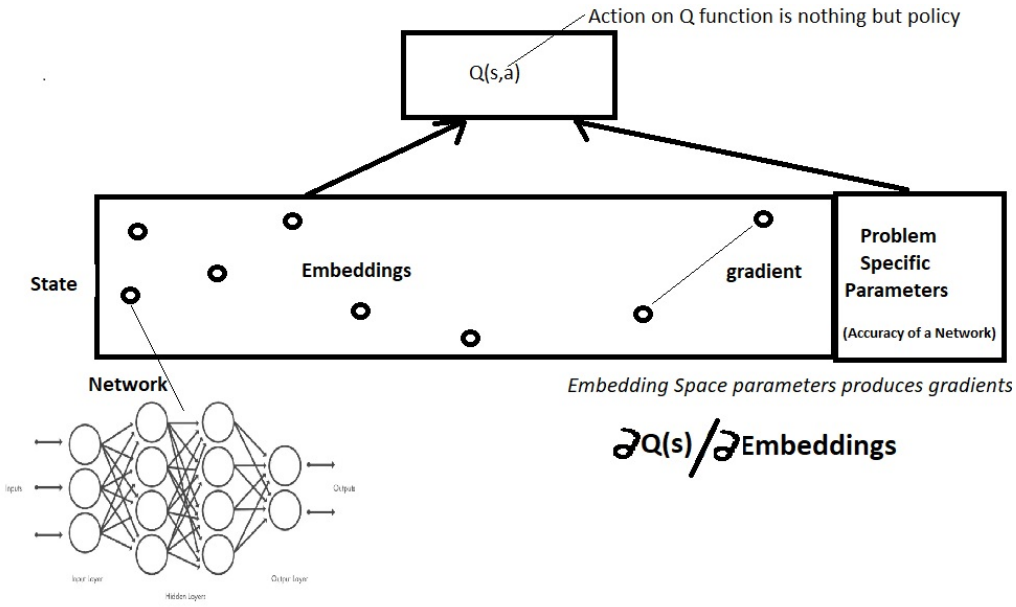


Figure 2.7: Network Embedding Space (left) and Problem Features (right) form State Representation for the Reinforcement Learning Component (top)

2.2.8 Embedding Space

Network architecture optimization has to deal with the problem that it needs to be able to represent the space of all applicable networks, and if using Reinforcement learning, an action space of network modifications that permits it to change any network into another network architecture. Moreover, it is well established that for different problems, different networks are optimal, implying that any network modification policy will have to have access to some problem specific features and characteristics.

To efficiently represent the set of networks for use by a deep learning system, much like Luo[1] we map a network description language into an embedding space using an encoder-decoder component. In addition, we will need the second problem specific input to train the policy on to facilitate the potential for transfer across problems. Here we embed the network and the merge the embedding vector with problem specific parameters to form the state representation for our policy learning components. Use of the lower-dimensional latent embedding for networks here also defines the policy's continuous action space as displacement vectors in the embedding space. The network embedding vector, augmented with a feature vector encoding the problem for which the network architecture is to be optimized, form the state for the RL problem while displacements in the embedding space provide the RL learner's action space.

One problem with such a derived embedding space is that it can fracture the space of legal network embeddings into disconnected regions. While this does not affect local optimization using gradient ascent as in [1], it makes learning RL policies for optimization more difficult as they need to cross regions of the embedding space which do not decode to valid network architectures. To address this and condense the space of legal networks, a Siamese Neural Network (SNN)[52] is used here in the encoder structure to organize the embedding space into coherent legal network regions.

In essence the embedding space (E) characterizes the space of all available networks modification functions where network modification actions correspond to moves in the embedding space that increase the utility of the resulting network for the given problem calculate gradients of this space in Reinforcement learning context.(Figure 2.7) shows an overview how the embedding space together

with problem-specific parameters forms the underlying representation for the Q-function of the Reinforcement learning framework.

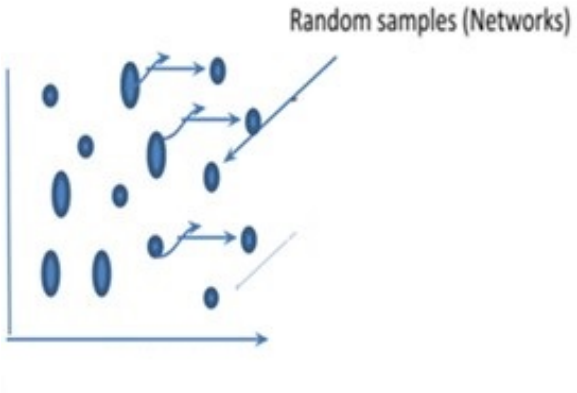


Figure 2.8: 2-Dimensional Embedding Space with Random Neural Network with their performance

My approach is partially inspired by Luo[1], where the two-dimensional embedding space is learned by taking bunch of networks randomly and figuring out performance for each of them (Figure 2.8).

2.2.9 Actor-Critic Methods and Policy Gradient

Utilizing an embedding space in this approach allows for a continuous action space, which is then optimized using actor-critic and policy gradient methods [68] [69]. In this context, the Q-value is learned by parameterizing the Q-function with a neural network in the augmented embedding space. The 'critic network' estimates the value function, including action-value (Eq.2.4), while the 'actor network' updates the policy distribution using policy gradients based on the direction suggested by the 'critic network'. Both the critic and actor functions are parameterized with neural networks in the embedding space, as shown in Figure 2.9.

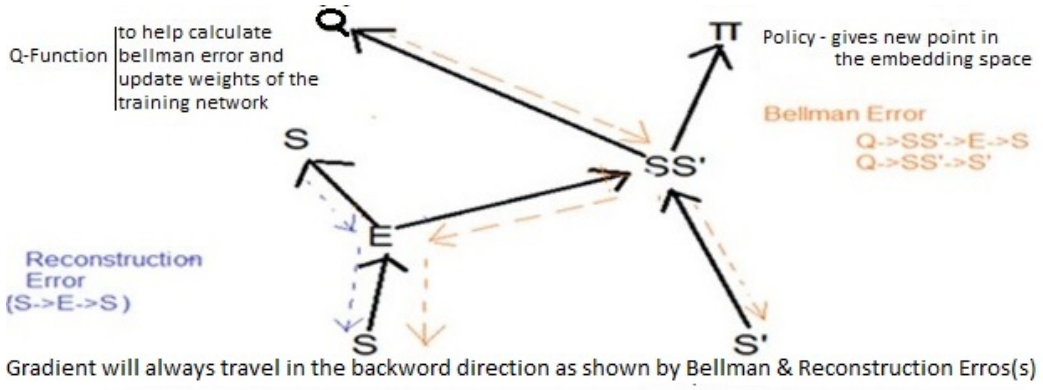


Figure 2.9: Schematic of the Connections between Embedding, Actor, and Critic

Policy gradients optimize the actor in the embedding space by updating the probability distribution of actions, ensuring that actions with higher expected utility (i.e., Q-value) have a higher probability for a given observed state. The proposed architecture employs a continuous (finite) action (embedding) and a stochastic (non-deterministic) policy. The goal is to learn a policy (π) that maximizes the cumulative future reward from any given time (t) until the terminal time (T). If r(t+1) represents the reward received by performing action a(t) at state S(t), then:

$$r(t+1) = R(s(t), a(t)) \tag{2.6}$$

To learn a policy, the reward must capture the incremental improvement achieved when modifying the network architecture. In supervised classification problems, the most direct way to capture reward is by measuring the expected improvement in network accuracy after a modification compared to before the modification. Since continuous evaluation during improvement is not feasible, the architecture includes a performance prediction component that predicts network performance using data from previously trained problems. To limit the need for task-specific training, the input to this network contains problem-specific features, alongside

the network embedding, to facilitate generalization across tasks.

Further, In order to effectively learn a policy, it is crucial to efficiently utilize the performance prediction component within the architecture. This component not only reduces the need for continuous evaluation during improvement but also allows for better adaptability across different tasks.

By incorporating problem-specific features into the input of the performance prediction component, the architecture is better equipped to generalize across various tasks. This helps in streamlining the learning process and enables the network to adapt to different problem domains without the need for extensive task-specific training.

The combination of continuous action space, actor-critic methods, and policy gradients allows the proposed architecture to optimize the policy and navigate the embedding space effectively. This ultimately results in improved network performance and the ability to tackle diverse problem domains with greater efficiency.

In summary, the proposed architecture aims to optimize network performance by learning a policy that maximizes cumulative future rewards. The use of an embedding space with a continuous action space, actor-critic methods, and policy gradients enables the architecture to adapt to various tasks effectively. The performance prediction component, combined with problem-specific features, further enhances the architecture's ability to generalize across tasks and improve the overall efficiency of the learning process.

2.2.10 Reward and Performance Prediction:

In order to learn an effective network optimization policy, the reward must represent the incremental improvement achieved through network modification. In the context of supervised classification, this can be directly measured as the expected accuracy gain of the target network after modification. To minimize the need for re-training target networks and enable planning, the architecture incorporates a performance prediction [16] component that estimates network performance based on previously trained problems. To reduce task-specific training requirements, the input to this network not only includes the network embedding but also features problem-specific elements, promoting generalization across various tasks.

In this approach, the performance prediction component plays a crucial role in efficiently determining the success of network modifications. By leveraging information from previously trained problems, it can estimate the impact of changes on the target network's accuracy without requiring extensive re-training. This allows for a more streamlined optimization process, saving valuable time and resources.

Additionally, incorporating problem-specific features within the input helps the architecture generalize across a wide range of tasks. This enables the network optimization policy to be more adaptable and versatile, making it applicable to various domains and problem types.

In summary, the combination of performance prediction and problem-specific feature integration enhances the architecture's ability to learn an effective network optimization policy. This approach facilitates efficient and adaptable network

modifications, ultimately leading to better performance in supervised classification tasks.

2.2.11 Twin Delayed Deep Deterministic Policy Gradient (TD3)

The twin-delayed deep deterministic policy gradient (TD3) algorithm [67] is a model-free, online, off-policy reinforcement learning method. A TD3 agent is an actor-critic reinforcement learning agent that searches for an optimal policy that maximizes the expected cumulative long-term reward. It is an extension of the DDPG (Deep Deterministic Policy Gradient) algorithm [70], where DDPG agents can overestimate value functions to produce sub-optimal policies. To reduce value function overestimation, the TD3 algorithm includes the following modifications of the DDPG algorithm.

- Agent can learn two Q-value functions and uses the minimum value function estimate during policy updates.
- Agent can updates the policy and targets less frequently than the Q functions.
- Agent can add noise to the target action when updating the policy, which makes the policy less likely to exploit actions with high Q-value estimates.

One can use a TD3 agent to implement one of the following training algorithms, depending on the number of critics one specify.

- Train the (TD3) agent with two Q-value functions. This algorithm implements all three of the preceding modifications.
- Train the agent (Delayed DDPG) with a single Q-value function. This algorithm trains a DDPG agent with target policy smoothing and delayed policy and target updates.

TD3 agents can be trained in environments with the observation space (continuous or discrete) and action spaces (continuous). They use deterministic policy

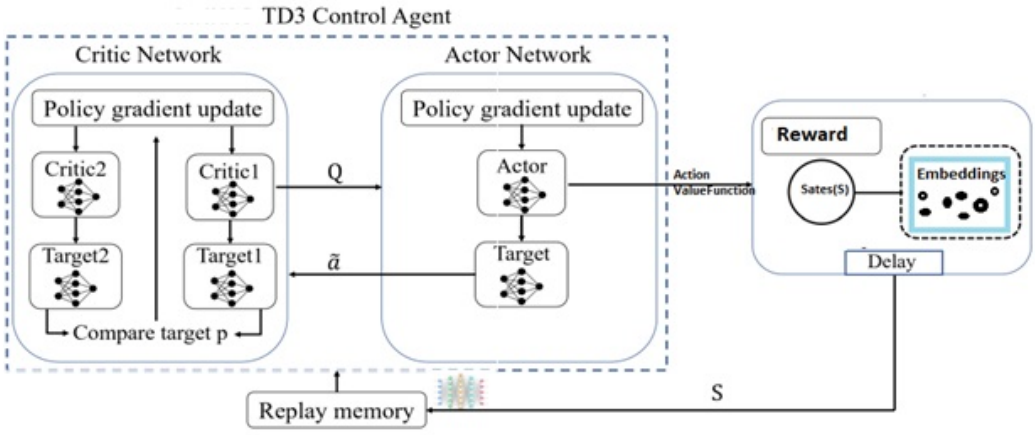


Figure 2.10: Internal structure and the implementation diagram of the TD3 policy gradient controller.
Ref: https://www.mdpi.com/1996-1073/14/20/6695

actor $\pi(S)$ and one or more Q-value function critics Q(S,A). During training, a TD3 agent updates the actor and critic properties at each time step during learning and stores past experiences using a circular experience replay buffer. The agent further updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer and perturbs the action chosen by the policy using a stochastic noise model at each training step.

A range of deterministic and probabilistic actor-critic and policy gradient algorithms have been proposed and implemented as show in the figure 2.10. Since deterministic policy methods have a tendency to produce target values with high variance when updating the critic therefore we need a strategy for smoothing the target policy to address this high variance caused by over-fitting to spikes in the value estimate. Our architecture takes advantage of the TD3 method [71] and handles this issue by focusing on reducing overestimation bias using a pair of

critic networks along with delayed actor updates while managing action noise regularization.

2.3 Combining Reinforcement Learning on Embedding Space, Policy and Neural Networks

Many existing methods, whether based on Reinforcement Learning (RL) or Evolutionary Algorithms (EA) [23], conduct architecture searches in a discrete space, which is highly inefficient. To address this issue, Luo [1] proposed using an autoencoder structure to learn a network architecture embedding space, facilitating automatic neural architecture design through continuous optimization. Luo[1] effectively employed an encoder to embed neural network architectures into a continuous space, followed by a predictor that takes the continuous representation of a network as input to predict its accuracy.

The performance predictor and encoder enable Luo [1] to perform gradient-based optimization in the continuous space to find an optimized embedding for a new architecture. This can be easily decoded from a continuous representation of a network back to its architecture, potentially improving accuracy. The network embedding is essentially a mapping from the observation (O) to a state (S), transforming the problem into a deterministic one as each observation (Neural Network) maps to a specific state, even though the actual state is unknown until the embedding is learned.

Luo's [1] has effectively used the concept of POMDP to model its embedding space, which makes this problem deterministic as every observation (Neural Network) is mapped to a particular state, but that state is not known till we learn

its embedding. The embedding (e) in this case is this mapping of observations to states: O ≡ S.

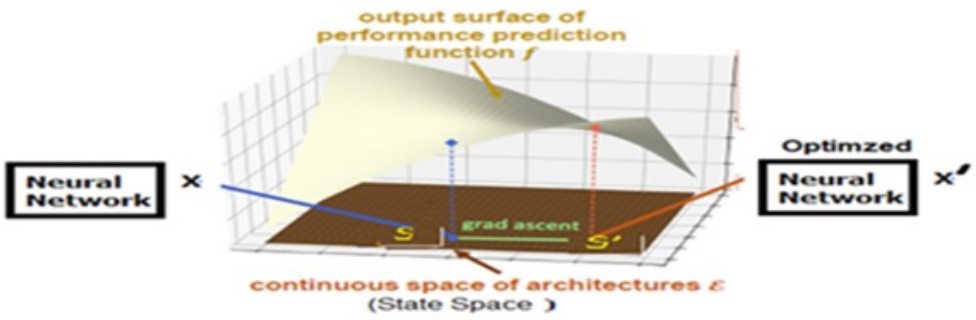


Figure 2.11: State Space (Part of the figure borrowed from [1])

Therefore, for x(network); embedding is nothing but a mapping x → s and policy is simply a gradient accent policy: π(s) → a (gradient direction); where it takes a state and maps it into an action (gradient direction). Figure 2.11 shows that Luo[1] has limited his action space into a two-dimensional direction and a neural network architecture(x) is embedded to state (s) with the help of encoder and new state s′ is accomplished after gradient ascent is performed by the using performance prediction function. This state s′ is later decoded to an optimal neural network architecture(x′)

Luo[1]'s embedding space leads to a continuous action repertoire for their trajectory-based optimization. However, the trajectories that yield improved networks are derived based on optimized local accuracy estimates, requiring significant amounts of additional training data for the specific problem to be generated during network

optimization. This results in a slow network optimization process and no generalization to other problems.
In contrast, Bose's[2] approach uses discrete and pre-determined network modification actions resulting from a Reinforcement Learning-derived policy, which is applicable to new problems without the need for large amounts of additional network training. This approach adapts well to data drift, streaming data, and new problems.

He has not established any direct relation between action space and state space directly as in his case the state space is a vector of accuracies, indicating how well a particular network is currently doing on a given problem. Bose[2] is using MDP therefore the accuracies are calculated for predefined state space. He has effectively used three state spaces to build three entities rather than one MDP for all modifications. Whereas Luo[1] is using one MDP (POMDP) for all network modifications and all actions exist within the embedding space.

However, it is possible that multiple X (Networks) map to same S (State $\rightarrow$ embedding). To find the best performing architecture for one single problem, Luo[1] runs bunch of networks on two high performing GPU for week to train approximately 1000 Networks to evaluate their performance as shown in Figure 2.12. Sadly, in case of data drift, this process will be obsolete, which means that all these old, trained Networks will not be valid. He trains his architecture for one problem at a time which is good for a fixed set of problems but will not give good results for a new problem. Another major challenge with Luo[1] architecture is that it did not address incrementally extending input space, as we all know that Neural Networks are notoriously bad at incrementally extending the input space because they will overfit to the data they get.

Performance functions in Luo's[1] approach are highly nonlinear, making it difficult to calculate gradient descent on the entire function. Additionally, the method only considers top-performing networks, which may cause the best-performing network to be missed if none of the initial networks were close to the best performance. One solution to this problem is to apply an effective exploration technique to the policy.

In summary, Luo'sLuo[1] architecture provides a larger action space, enabling it to easily handle various problems, while Bose's[2] architecture has a more effective performance function. Each approach has its own strengths and weaknesses, and a combination of the two methods could potentially yield a more robust and efficient solution for network optimization.

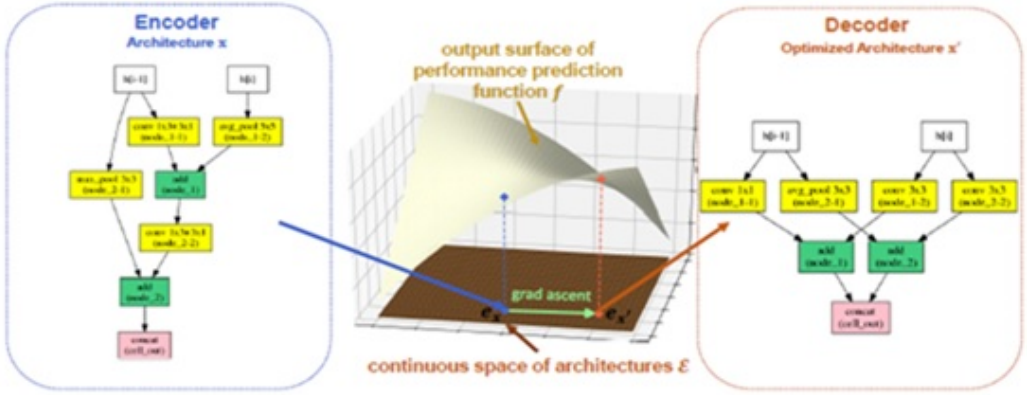


Figure 2.12: Neural Architecture Optimization (encoder-decoder-predictor) Luo[1]

2.3.1 Comparison

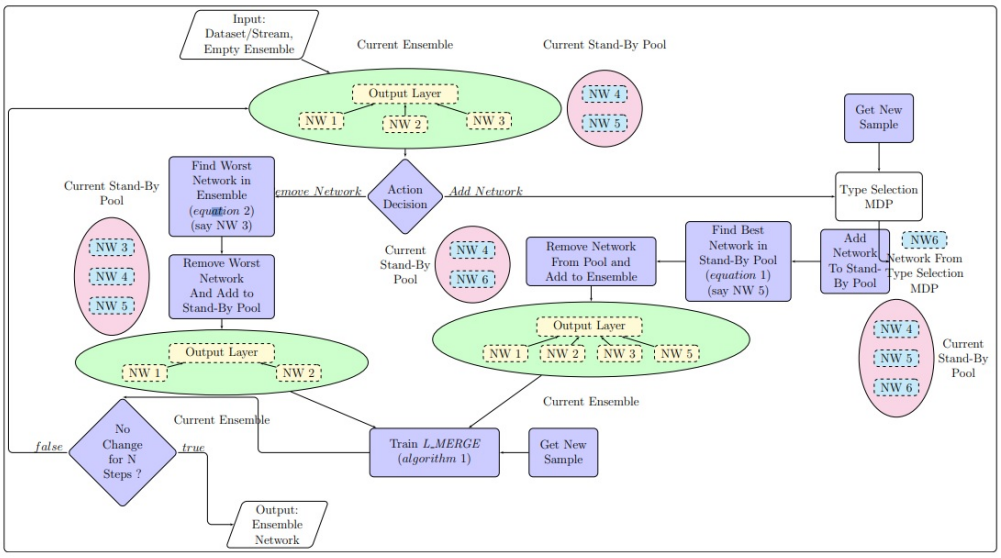
Performance function is horrendously nonlinear in case of Luo[1] and embedding space is huge, which makes it impossible to calculate gradient decent on the whole function. Also he ignores all bad points and only considers good points

which are nothing but top performing networks, making performance function to learn about best performing networks in some neighborhood in the embedding space. This helps him to formulate their policy which is nothing but the gradient. It is evident that this policy is executed for certain number of predetermined steps to give him whole bunch of new Networks and their performance can be evaluated. This process is repeated by ignoring old networks which are not performing to help learn a new performance function and thereby build a more precise gradient. Only if this newly generated policy is reasonable enough then there is a high chance of generating better performing network.

If initially he generates broad enough networks, then there is a strong possibility that there will be some networks which are close to some good performing network resulting into one region of the embedding space where Luo[1] will get the best performing Network. Repeated training helps Luo[1] to get a better performance function. But on the flip side, there is a chance to miss the best performing network, if none of the initial networks were close to the best performance. Further, if the embedding space is nonlinear then his architecture might never lead to the best performing network. One way to solve this problem is to apply an effective exploration technique to this policy. Also, we need to keep track of entire network design which includes layers, weights, number of nodes and all hyper parameters etc., unlike Luo[1], which uses dynamic approach to adjust learning rate.

In contrast Ensemble network Bose[2] adapts well with data drift as well as with streaming data and to new problems. At a given point of time the streaming window only shows one part (subset) of the object but over the time it might show other parts of the object. Hence, we will only get performance of the subset of

the original input of the problem. In his approach he first uses Reinforcement Learning to learn a policy to incrementally build neural network classifiers for a broad distribution of problems and subsequently applies it to new data to learn a classifier for a given specific problem. As we see in the Figure 2.13, with the help of mixtures of experts in its stand-by pool he can clearly build a good network for a given input data. If the network in hand no longer works due to data drift and if the input data does not match with the previous input data (due to Drift) then the system can select a network from its pool that may work for this data. This way if the data drifts then it just need to grab appropriate network from the back up and it don't have to build it again from scratch. This is a huge win with respect to Luo[1], as in his case it is all or nothing as his design has no provision for data drifts. Luo[1] must throw away all those networks as they become invalid for change in input data.



Ensemble Learning MDP Workflow

Figure 2.13: Ensemble Learning MDP Workflow) Bose[2]

The key point to note with Bose[2] design is that every network is trained for slightly different dataset and hence for a different purpose. This is bit problematic, as it means that it is order specific not order invariant. Bose[2] has deployed three MDP's that handle three different scenarios. First MDP helps to decide if it needs to build a new network from the scratch, or should he pick a new network or pick an existing network that is currently not in use or drop a network from its ensemble. He can only pick a network with highest performance from a pool of pre-selected networks. The performances are evaluated based upon the streaming data to see how they are doing. Dropped networks are saved in the pool of unused networks and only worst performing networks are dropped Figure 2.13. Second MDP also known as Network Creation MDP - is used to modify the network by adding a layer or a add a unit to the deepest layer. Output layer is always kept fixed as it is directly associated with the problem. Third MDP is Type Selection MDP, which is used to decide between a One vs Rest or an All vs All network resulting in multi-class classifier or one against all classifiers.

Note all three policies take binary decision as it must do Q-learning therefore the action space have to stay discrete. The advantage we have with Luo[1] is that the action space (embedding) is not discrete which can help to embed all possible modification to that network into continuous embedding space to allows to build gradient over the time. The benefit of embedding is that it takes a very high dimensional discrete space and represents it with an approximation into a lower (two) dimensional continuous space. This is a huge win as one of the biggest advantages of the continuous space is that it has gradient potential which is not possible in discrete space.

The action space in case of Bose[2] is independent of the state space therefore Reinforcement learning does not require differential elements there and action space is not large because he is only building fully connected networks. Whereas action space in the case of Luo[1] is very large and high dimensional and if there is a need to build different types of layers for different parts of the Network, then this space of the modifications becomes big and unmanaged. As far as action space is concerned, clearly, Luo's[1] architecture wins as it gives a large action space so that it can easily deal with all kind of problems. On the other hand, performance function works much better in Bose's[2] architecture.

2.4 Summary

For all neural network architectures, we need a very high dimensional action space. The way we can potentially address this is by heaving an embedding space where one can descend/ascend the Q-function. This embedding will be nothing but RL space and will encode all networks so that we don't have to worry about adding a layer or dropping a layer or node etc. All directions in this embedding space are nothing but modifications of the given Network. Since we need to map high dimensional space to lower dimensional space and theoretically lower dimensional space can represent all given Network's. In some practical scenarios it is possible that we get into a situation where more than one Network (X1 and X2) map to same Embedding (s) leading to a potential error which is not avoidable.

$$X1 \rightarrow S \leftarrow X1$$

We also know that every problem is unique and to avoid any conflict in the embedding space mapping, we will need the second problem specific input (Feature Vector's) as part of our embedding space to train it on. This strategy will help

us get different embeddings for different problems, otherwise we will try to learn the universal network that does not exist. It is a cleaner solution, that we embed the network and merge it with problem specific parameters to do predictive function. This will make action space on modified networks stay uniform across all problems. Since embedding space characterizes the space of all available network modification functions and actions are used to calculate gradients of this space in RL context. Proper autoencoder training can also help us to further avoid this conflict.

Another challenge we need to deal with is, to evaluate, if we can use embedding space like Luo[1] to build a bigger network space while able to maintain following two important features:

1. System should not build a pure monolithic network so that if the input data changes, we can recover fast.
2. Learn a Policy that has a fixed policy of modification at least for the early stages to find the neighborhood of network architectures and only at the very end network is evaluated purely based upon its performance.

After some research we found that one of the possible solution to the problem above is that we can have generic embedding to map the network (X) to a state (S) with its performance (accuracies). The Figure 2.14, represents the state space of the problem. It contains both things; the Network and attributes of each Network in the ensemble.

It is evident that Bose[2] has effectively used Ensemble Network Architecture to use accuracy as a state space, whereas Luo[1] uses performance of network in its embedding space. It will be interesting to see if we can get the expected final value derived by the potential improvement over the time. Since the expected value of

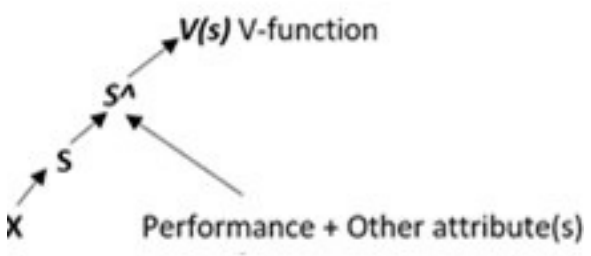


Figure 2.14: State Space with Performance)

the accuracy is achieved after gradient descent is completed. This can safely be called an 'Expected Accuracy' of an Optimized network. The new state $\hat{S}$ of the proposed architecture should contain the network information along with performance function. The biggest win we get from our proposed architecture is that accuracy is independent of the problem action(embedding) space.

My new State $\hat{S}$ will get me the expected value of the accuracy achieved after the gradient descent is completed of the optimized network. Therefore, when we apply different problems on the same network, we may get different performances unlike Bose[2] where he get the same ensemble no matter what problem is feed into his Architecture. The Proposed Architecture can use embedding space as a continuous action space and not as a discrete action space which will get different performances for different problems in the embedded states.

In summary, the goal of my research is to investigate if we can combine the concepts of a network embedding space from Luo[1] with general problem representation and Reinforcement Learning framework used by Bose[2] to construct a network optimization architecture that can utilize the increased action space to address more complex networks and still learn complete optimization policies using Reinforcement Learning, that might be applicable to new problems. Taking inspiration from

a range of previous work done on network embeddings, network architecture optimization (NAO) and also carefully evaluating work done by Luo[1], Ensembles Bose[2], Zhou[18] Granitto[27] and various approaches to network architecture search by Baker[17], Bowen[16], Brock[28], Cai[29] and Kapanova[14]. We decided to focus on an architecture that utilizes embedding and Reinforcement Learning concepts to provide the promise to extend to various network types as well as to transfer across problems without the need for extensive re-training for each problem, thus potentially opening up ways to address streaming and drifting data in a more complex form as in Bose[2]. Figure 3.1 shows the complete network architecture, including embedding, performance prediction, and policy learning networks.

CHAPTER 3

Approach

This dissertation proposes a novel approach for automatic network optimization that utilizes Reinforcement Learning (RL) on top of a learned network embedding space. The embedding space is obtained using an encoder-decoder network and it provides a compressed representation of all network architectures for the problem task. Additionally, it provides a corresponding fixed-dimensional, continuous network modification action space represented as direction vectors within the embedding space. Based on this embedding, the approach utilizes TD3 actor-critic networks to learn a network architecture modification policy. To reduce the need for target network retraining and facilitate planning, the RL approach uses an accuracy prediction network to generate reward predictions. The accuracy prediction network uses the network embedding and the target problem feature vector to predict the achievable classification accuracy for the target problem, as well as whether the embedding corresponds to a legal decoded network.

To further organize the embedding space, a Siamese Neural Network (SNN) [52] is employed to co-locate embeddings of valid network architectures. A loss term is added to the SNN to represent the precision of target problem accuracy prediction. To allow generalization across both the RL components and the accuracy prediction network, an additional feature vector is introduced that abstractly encodes the complexity of the target problem.

The main advantage of this approach over previous techniques is that it combines the ability to learn over a flexible network embedding space with the

potential to transfer the learned policy across new target tasks. Moreover, it also improves upon previous work by incorporating an accuracy prediction network that reduces the need for target network retraining and facilitates planning. The use of a Siamese Neural Network to co-locate valid network embeddings in the embedding space here adds an extra level of organization, making it easier to navigate and manipulate the space. This organization is further enhanced through the inclusion of the loss term in the SNN also ensures that the embeddings are accurate and relevant to the problem task.

Generalization across target tasks is in part facilitated here through a task embedding driven by a feature vector that abstractly encodes the complexity of the target problem. This feature vector allows for transfer and ensures that the RL components and the accuracy prediction network can be utilized across a variety of scenarios.

Put together, this dissertation proposes an innovative approach for automatic network optimization that utilizes a learned embedding space, a Siamese Neural Network for organization, and an accuracy prediction network for reward prediction. The approach provides an effective solution for optimizing network architectures and can be utilized across a variety of problem tasks.

3.1 Architecture

As indicated before, to address the network optimization, the approach we would like to take is to use reinforcement learning on top of a network embedding space to utilize it as a continuous space and to form a structure that provides the promise to eventually be able to generalize across problems and thus to be able to learn a policy that can improve network architectures for novel problems without

the need for excessive re-training. To achieve this, we propose the integrated architecture shown in Figure 3.1.

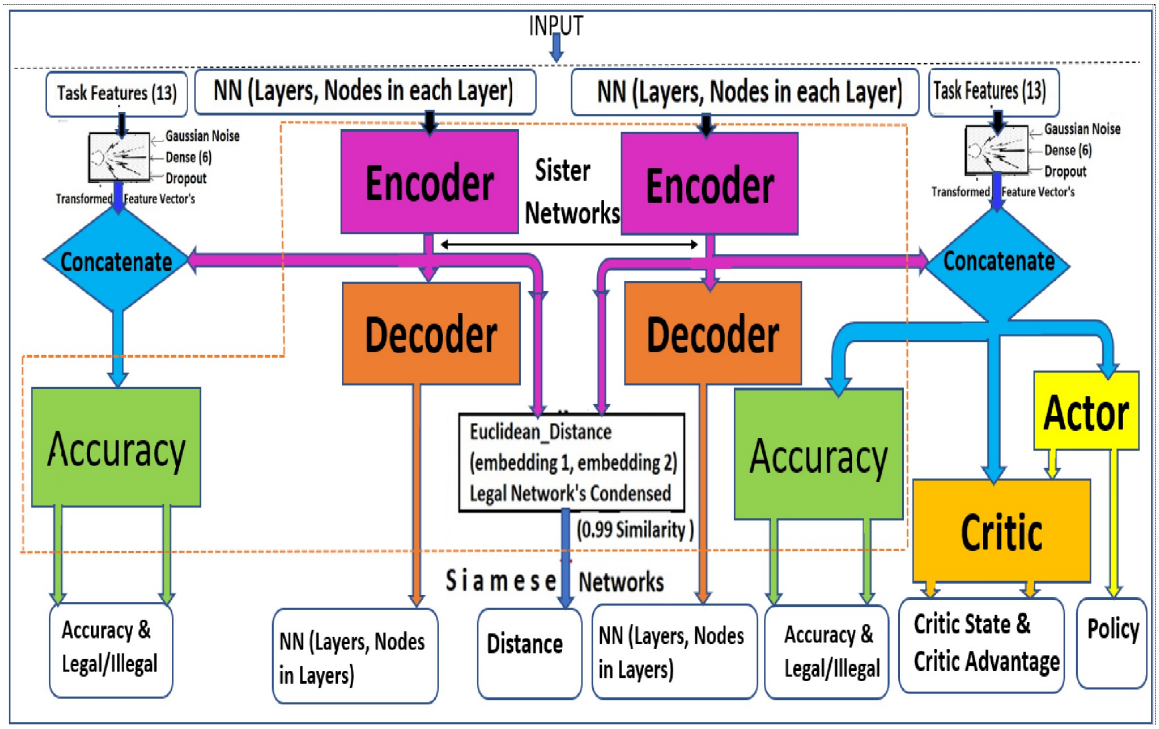


Figure 3.1: Architecture Overview. Embedding Space is Derived Using a Encoder-Decoder (left) and Augmented with Problem Features to Form the Input to an Accuracy Prediction Network (center) which Predicts Rewards, and an Actor-Critic Network Pair (right) which Learn the Utility Function and the Network Modification Policy Using Policy Gradient

This architecture takes the 'network architecture description' and the 'problem description' as input. The former represents the network characteristics, such as number of layers and nodes, and the latter represents features reflecting the complexity of the target problem, such as dimensionality, data size, feature correlations, or accuracy achieved by baseline architectures.

The 'encoder' model transforms the network description into an embedding space which, with the problem description, serves as input to the 'accuracy' pre-

diction, 'actor' and 'critic' networks. The 'decoder' model is used to translate the embedding back to a corresponding network architecture for use on the target task. To improve the organization of the embedding space, a Siamese network [52] is also used, encapsulating the encoder-decoder and action prediction networks as indicated as a dashed box in the Figure 3.1.

3.2 Key Components of the Architecture

The architecture was evolved over the time, component by component and layer by layer. The key network building blocks of the architecture are:

1. Encoder
2. Decoder
3. Accuracy
4. Predictor
5. Actor
6. Critic

The network architecture description encodes the number of layers, number of nodes in each layer and connectivity of the network architecture for the target problem. The second input in the form of a problem description vector, also known as a task feature vector contains relevant aspects of the target task and is aimed at allowing transfer to new target tasks for both the actor policy and the accuracy predictions, thus minimizing the need for re-training when applied to a never before seen target task. For initial training, the approach also saves the accuracy of a range of randomly generated target task networks, network status (legal/illegal), and network properties as truth values.

We split the input into network state properties and problem description (Feature Vector) as shown in Figure 3.2.

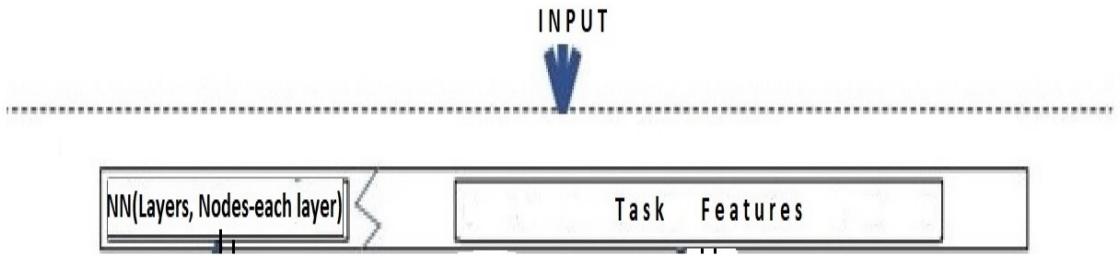


Figure 3.2: Input Layer

3.2.1 Encoder Model

The 'network architecture description' here is converted by the 'encoder model' into the continuous embedding space. It takes the Network description such as number of layers and number of nodes in each layer and maps it into 2-D plane as shown in Figure 3.3

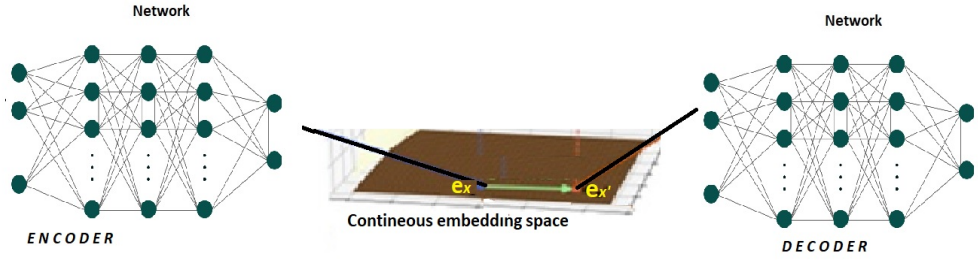


Figure 3.3: Encoder-Decoder

3.2.2 Decoder Model

The decoder model takes the encoded Network properties from the 2-D embedding space and decodes it back to the original Network properties such as, number of layers and number of nodes in each layer, etc, with great accuracy. Figure 3.3 shows the basic concept of the enoder-decoder architecture. The left side output in the overall architecture in Figure 3.1 shows the 'decoder model' aimed at allowing to re-create the network from the embedding space.

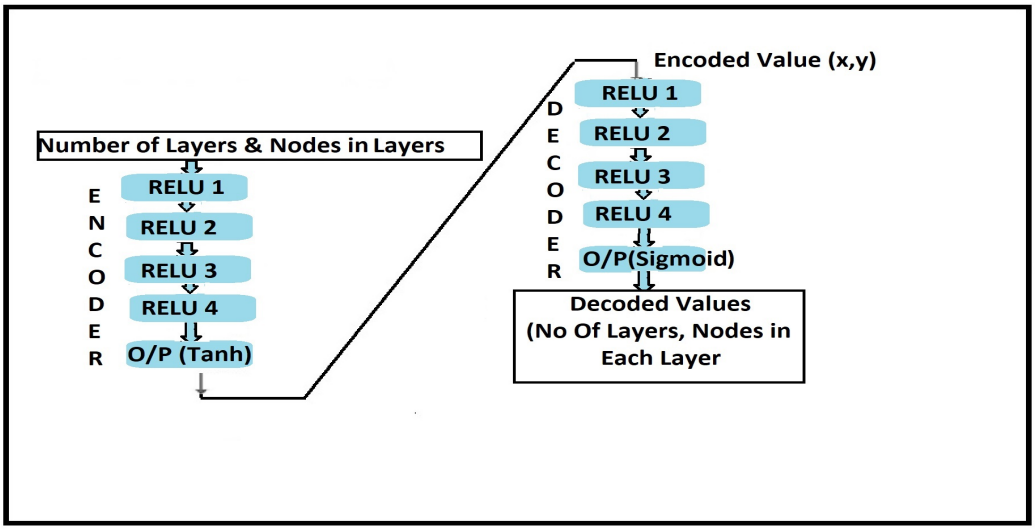


Figure 3.4: Encoder-Decoder Layered Diagram

The 'encoder model' and 'decoder model' are trained here as an autoencoder to form the embedding space and then fine tuned by the addition of the 'accuracy prediction' network using a data set of previously trained networks for various learning problems. Initial experiments were done on simple fully connected networks where the standard autoencoder layered diagram is shown in Fig.3.4.

3.2.3 Sequence-to-sequence Autoencoder

After the success with constrained simple fully connected networks, there was a need to generalize the network architecture to more general fully connected networks and eventually beyond fully connected networks across a wider set of network types such as convolutional networks. This can be achieved by introducing a modular network component input in terms of the characterization of the network, instead of just passing the number of layers and number of nodes in each layer. This will require to change our input vocabulary, to be able to input more complex networks. For an initial version of this we plan to utilize a sequence of characters in pairs with start and stop condition as an input to the encoder which will allow it to eventually characterize different types of networks such as RNN,CNN and other networks. The paired sequence allow us to present more complex networks with arbitrary layers and nodes and utilizes a sequence-to-sequence recurrent encoder-decoder architecture [72].

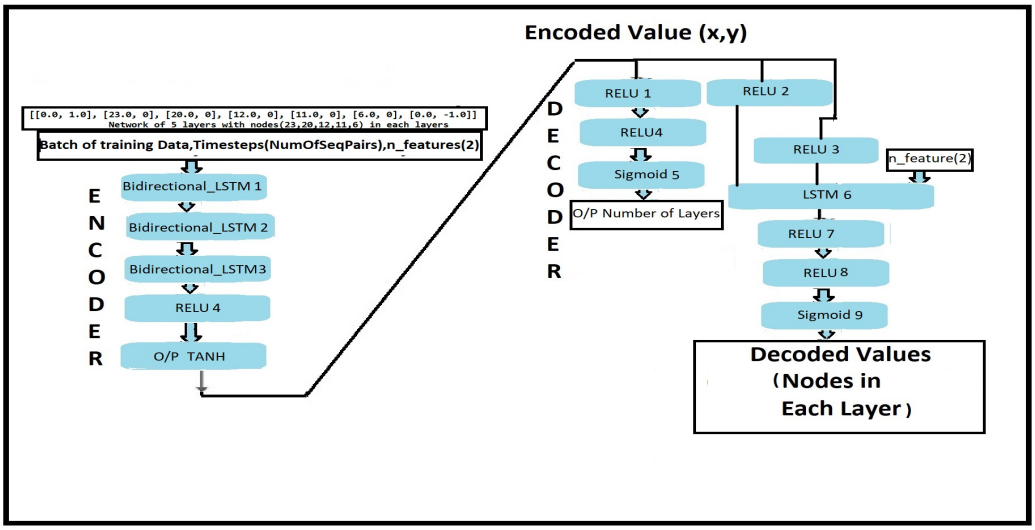


Figure 3.5: Layered diagram of Sequence-to-sequence Autoencoder

The model architecture consists of an encoder-decoder structure, wherein Long Short-Term Memory (LSTM) [73] layers are employed for both encoding and decoding operations. The purpose of this model is to learn to predict the subsequent values in a time series, given an input sequence of integer values, which is nothing but representing in its simplest form the number of nodes and connectivity in each layer.

For example: A 5 layered network with (23,20,12,11,6) nodes in each layers can be represented for the encoder-decoder architecture shown in Figure 3.5 as:

[[0.0, 1.0], [23.0, 0], [20.0, 0], [12.0, 0], [11.0, 0], [6.0, 0], [0.0, -1.0]]

Here [00,1.0] represents the start of the network description and [0.0.-1.0] represents the end of the sequence.

3.2.4 Accuracy Model

As accuracy prediction is problem specific, the accuracy model's input is comprised of the concatenation of the network architecture embedding and a 'problem description' (feature vector) together with encoded network as a network state and predicts a network performance measure of the network for the target problem as well as whether the embedding state corresponds to a legal architecture. It serves to help organize the embedding space and to generate simulated rewards in order to minimize the amount of network training that has to happen when optimizing a network architecture. The example layered architecture used in the experiments presented later is shown in Figure (6.8).

During the first phase of training, the Encoder-Decoder and Accuracy models are trained together. We use a custom loss function that adds MSE loss to the weighted square activation of an accuracy layer. We also use custom accuracy

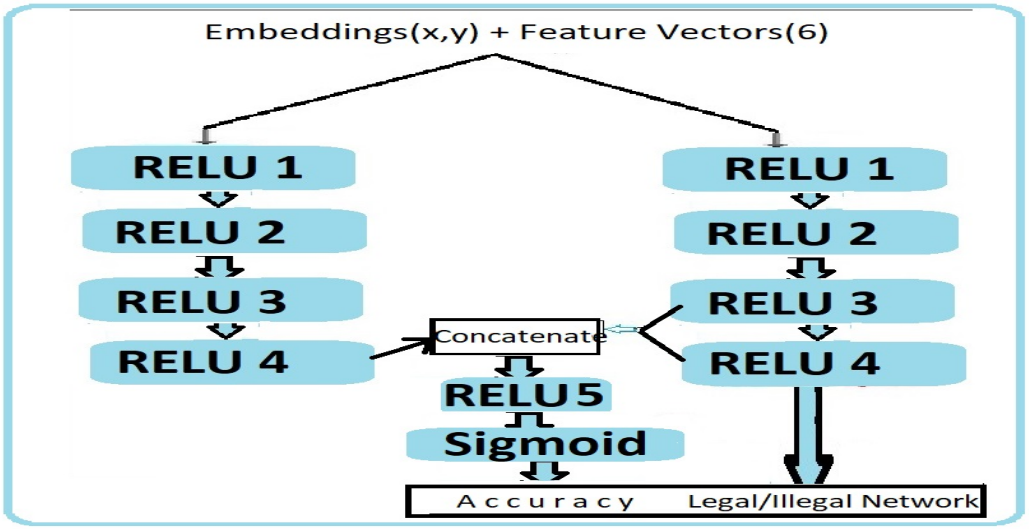


Figure 3.6: Accuracy Network Layers

matrix for decoder by calculating MSE of true and predicted values. This combined Encoder-Decoder-Accuracy model is trained in our initial experiments from a set of 38235 networks for approximately 10,000 epochs, for both legal and illegal networks using '0.0' as an accuracy for illegal networks. This initial training helped to learn about the embedding space. During the Second and final phase of training, Encoder-Decoder and Accuracy models are trained with auto-encoder sequence-to-sequence model for time series data using the Keras deep learning library to train the encoder and decoder along with accuracy network with more complex neural networks architecture.

3.2.5 Siamese network to co-locate embeddings

To avoid discontinuous regions of legal network embeddings and thus to better organize the embedding space, this autoencoder is augmented with loss terms from the 'accuracy prediction' network and arranged in a Siamese network to co-locate embeddings that decode into legal network descriptions. The effect of this

is an embedding space that is easier to traverse by the actor. An example illustrating the resulting compact layout of the learned embedding space is shown in Figure 6.3 where blue regions correspond to legally decodable network architectures while red regions decode to illegal descriptions.

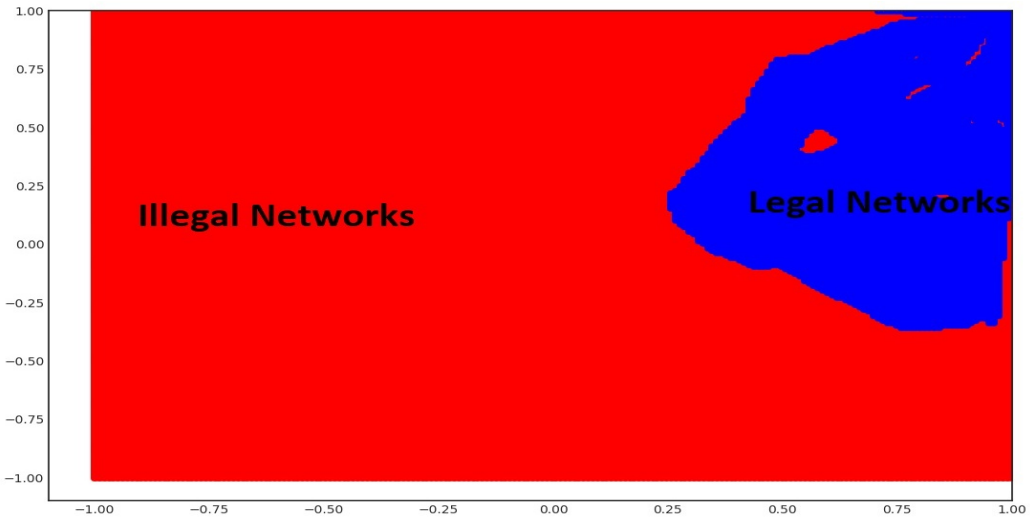


Figure 3.7: Co-located Legal and Illegal Network's

3.2.6 Actor Model

As in the case of the accuracy prediction, the joint embedded state and target task representation also feeds to the 'actor model' to learn the network architecture optimization policy. The right side output of the architecture diagram in Figure 3.1 shows the 'actor network' which gives us an effective policy to optimize a network. The Actor model is used ultimately for the forward propagation operation through the network to compute the policy (π), which helps us to select the next step (action 'a') in the embedding space and then generates the new corresponding network with the help of the decoder.

3.2.7 Critic Model

The critic model evaluates actions taken by the actor based on the given policy and learns the utility function to output 'State' and 'Advantage' values. Here actions taken by the actor are in the context of the augmented embedding state. The critic learns the utility function to output state values, $V(s)$, and Advantage values as:

$$V(s) = E_{\pi}\left[\sum_{t} \gamma^{t} r_{t}\right] \quad , \quad A(s,a) = Q(s,a) - V(s) \tag{3.1}$$

Internally, this critic maintains 2 separate value function estimates for both utility functions as well as a common target network that achieves an action update delay and function smoothing. For the actor critic component, the reward function is derived as the actual (for online learning) or predicted (for planning) improvement of the performance prediction of the 'accuracy' model, leading, to policy trajectories. For the actor critic component, we can interpret the network embedding augmented with the problem-specific features as the state and derive a reward function as the predicted improvement of the performance prediction provided by the 'accuracy model', leading, during execution of a policy trajectory to the state-action-reward sequence shown in Figure 3.8.

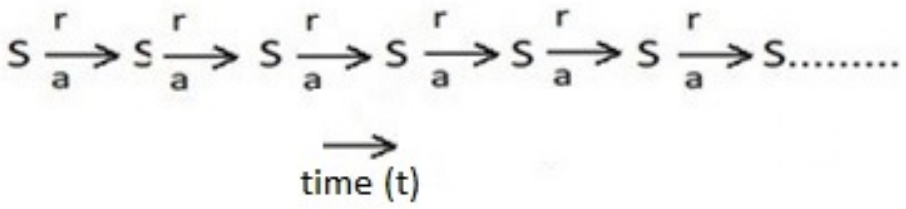


Figure 3.8: State-Action-Reward Sequence for Policy

To train the actor and critic components, the system generates new embedding space trajectories in each iteration as shown in Figure 3.9, which serve as training

data together with the reward extracted from the accuracy network's performance predictions. As the policy changes over time and the learner is largely on-policy, it is essential that old trajectories are discounted and new ones generated continuously.

The outcome of the RL policy is an action vector $\vec{a}$. We start training initially by picking a couple of hundred steps in trajectories. There are two main benefits of this training known as 'Forward' & 'Backward use'. 'Forward use' helps us compute the policy, which tells us what action we should take and as a result of the action what network is produced, where as 'Backward use' helps to learn the embedding space, which is done when we have enough data from trajectories.

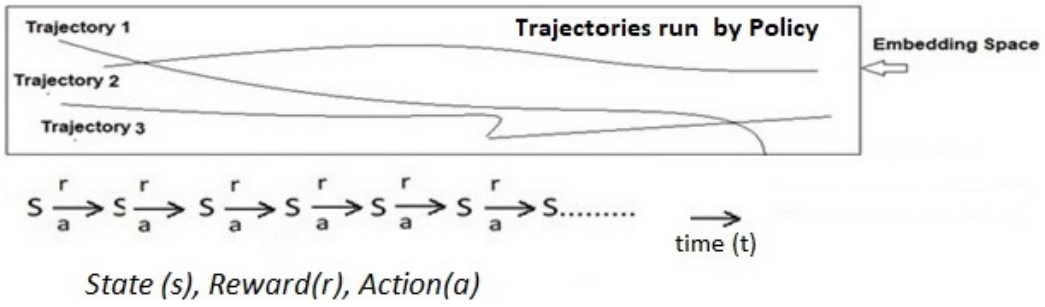


Figure 3.9: Generation of Embedding State Trajectories

This architecture addresses the optimization problem independent of choices that we need to make for the action in-order to be able to optimize the network architecture for a given problem.

3.3 Other Foundational Elements of the Architecture

Essential architectural components serve as the fundamental building blocks that contribute to the overall functionality, stability, and performance of a system. They also form the basis of any successful system, and understanding their roles

and interdependencies is critical for designing and implementing efficient, secure, and adaptable solutions that can effectively address our research needs and technological advancements. These crucial elements encompass 'Actor-Critic Methods and policy gradient', 'REINFORCE algorithm', 'Q Actor Critic as well as Twin Delayed Deep Deterministic Policy (TD3) Gradients', each with its unique set of elements that collaborate to achieve our research goals. Also, by carefully designing and integrating these components, with our architecture, we can create a scalable, modular, and maintainable system that is capable of meeting specific performance and reliability requirements for Network Architecture Optimization. Furthermore, essential architectural components play a critical role in facilitating communication and data exchange between different subsystems, ensuring seamless interoperability and efficient resource utilization.

With the help of the Architecture and its essential components an approach is presented that uses Reinforcement Learning on a network embedding space to learn a policy that can yield an optimized network architecture. The embedding space here converts the space of network architectures into a lower-dimensional continuous space and defines a continuous action space for the RL agent. To permit generalization across target problems and to facilitate planning, the system also utilizes a feature vector encoding target problem complexity, and a derived performance prediction component.

In summary, a well-designed system architecture relies on the careful selection and arrangement of essential components, which together form the backbone of any robust and high-performing system.

3.3.1 Actor-Critic Methods and policy gradient

The Q value, as well as V(s) and A(s, a), can be learned by parameterizing the corresponding value function with a neural network in the embedding space (Figure 2.7 and 2.9). This helps the Actor-Critic [68], to perform their functions as:

- Critic estimates the value function as action-value (the Q value) and state-value (the V value).
- Actor updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

Both Critic and Actor functions are parameterized with neural networks. To understand this better we need to understand how policy gradients are driven derived and how RL implements them. The two most popular classes of RL algorithms are Q-Learning and Policy gradient. Where Q learning is a type of value iteration method that aims at approximating the Q-function, while policy gradients is a proven method to directly optimize in the action space. The policy gradient method is also the "actor" part of Actor-Critic methods. In essence, the policy gradient method updates the probability distribution of actions so that actions with higher expected utility reward have a higher probability value for an observed state assuming we have a continuous (finite) action space and a stochastic (non-deterministic) policy for our experiments.

The objective function for policy gradients is defined as:

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1}] \tag{3.2}$$

In other words, the objective is to learn a policy that maximizes the cumulative future reward to be received starting from any given time t until the terminal time

T. Note that r_{t+1} is the reward received by performing action a_t at state s_t ; r_{t+1} = R(s_t, a_t) where R is the reward function.

Since accumulated reward needs to be maximized, therefore the policy is optimized by performing taking the gradient ascent with the partial derivative of the objective with respect to the policy parameters θ where the policy function is parameterized by a neural network.

$$\theta \leftarrow \theta + \frac{\partial}{\partial \theta} J(\theta) \tag{3.3}$$

Since we want to optimize long term future (predicted) rewards, which has a degree of uncertainty, expectation, also known as the expected value or the mean, is computed by the summation of the product of every x value and its probability.

$$\mathbb{E}[f(x)] = \sum_{x} P(x) f(x) \tag{3.4}$$

Here P(x) represents the probability of the occurrence of random variable x, and f(x)is a function denoting the value of x. Hence, the policy gradient can be defined as:

$$\Delta_\theta J(\theta) = \sum_{t=0}^{T-1} \Delta_\theta log \pi_\theta(a_t \mid s_t) G_t \tag{3.5}$$

The detailed Architecture with specific Actor and Critic Network is shown below:

We have considered two classic variants of actor critic methods, namely 'Q Actor Critic' (Equation 3.6) and 'Advantage Actor Critic' Equation 3.6); as:

$$\Delta_\theta J(\theta) = \mathbb{E}[\Delta_\theta log \pi_\theta(s, a) Q^w(s, a)] \tag{3.6}$$

$$\Delta_\theta J(\theta) = \mathbb{E}[\Delta_\theta log \pi_\theta(s, a) A^w(s, a)] \tag{3.7}$$

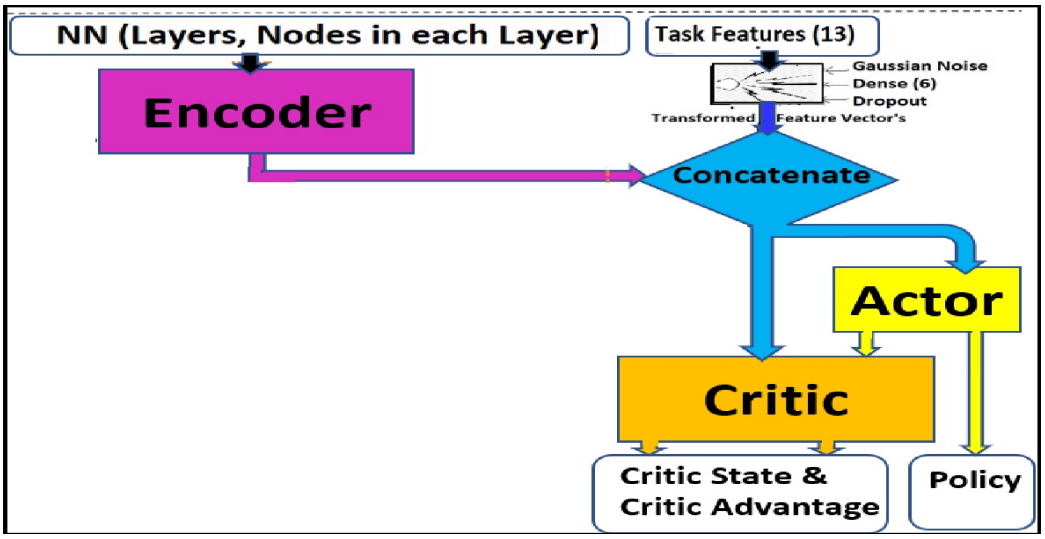


Figure 3.10: Actor-Critic Architecture

3.3.1.1 Q Actor Critic

The Q Actor-Critic algorithm is a type of reinforcement learning method that combines the benefits of both value-based (critic) and policy-based (actor) approaches.

To derive the Actor Critic architecture we revisit the policy gradient once again as :

$$\Delta_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \Delta_\theta log\pi_\theta(a_t \mid s_t)G_t] \tag{3.8}$$

We can then decompose the expectation into:

$$\Delta_\theta J(\theta) = \mathbb{E}_{s_0,a_0,....,s_t,a_t}[\sum_{t=0}^{T-1} \Delta_\theta log\pi_\theta(a_t \mid s_t)\mathbb{E}_{r_{t+1},s_{t+1},....,r_T,s_T}[G_t] \tag{3.9}$$

The second expectation term is the Q value

$$\mathbb{E}_{r_{t+1},s_{t+1},....,r_T,s_T}[G_t] = Q(s_t, a_t) \tag{3.10}$$

Using Equation 3.10 in Equation 3.11 we can rewrite the update equation as:

$$\Delta_\theta J(\theta) = \mathbb{E}_{s_0,a_0,....,s_t,a_t}[\sum_{t=0}^{T-1} \Delta_\theta log\pi_\theta(a_t \mid s_t)]Q_w(s_t, a_t) \tag{3.11}$$

This leads us to the Actor Critic Methods, where:

1. "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value).
2. "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

Both the Critic and Actor functions are parameterized with neural networks. In the derivation above, the Critic neural network parameterizes the 'Q value'. Therefore, it is called 'Q Actor Critic'. As illustrated, we can now update both the Critic network and the Value network at each update step.

Here is a high-level algorithm for the Q Actor-Critic method:

1. Initialize the actor network with parameters θ and the critic network with parameters Ψ.

2. Initialize the target networks for the actor and critic with the same parameters $\theta' = \theta$ and $\Psi' = \Psi$.

3. Initialize the replay buffer D to store experience tuples (s, a, r, s', done).

4. Set the number of episodes and steps for each episode.

5. For each episode:

a. Reset the environment and initialize the initial state s.

b. For each step within the episode:

i. Use the actor network with parameters θ to select an action a based on the current state s (with exploration, e.g., using an epsilon-greedy strategy).

ii. Execute the action a in the environment and observe the next state s', reward r, and done signal (whether the episode is terminated).

iii. Store the experience tuple (s, a, r, s', done) in the replay buffer D.

iv. Sample a minibatch of experiences from the replay buffer D.

v. Compute the target Q-value using the target critic network with parameters Ψ' and the target actor network with parameters θ' : $y = r + \gamma * Q'(s', \pi'(s'; \theta'); \Psi') * (1 - done)$

vi. Update the critic network by minimizing the loss between the predicted Q-values and target Q-values: $L(\Psi) = (Q(s, a; \Psi) - y)$ ^ 2

vii. Compute the gradients for the actor network using the chain rule and the critic network's gradients:
$\nabla\_\theta J(\theta) = E[\nabla\_a Q(s, a; \Psi)|a = \pi(s; \theta)] * \nabla\_\theta \pi(s; \theta)$

viii. Update the actor network parameters θ using the computed gradients.

ix. Update the target networks for the actor and critic using a soft update strategy:
$\theta' = \tau * \theta + (1 - \tau) * \theta'$, $\Psi' = \tau * \Psi + (1 - \tau) * \Psi'$

x. Update the state s = s'.

c. If the episode is done, break the loop and start a new episode.

Note that the algorithm above is a high-level description of the Q Actor-Critic method. There are many variations and improvements to the basic algorithm, such as using different exploration strategies, learning rate schedules, and network architectures.

3.3.1.2 Advantage Actor Critic

Advantage Actor-Critic (A2C) is a reinforcement learning algorithm that combines the strengths of both policy-based (actor) and value-based (critic) methods.

The key idea behind A2C is to use an estimate of the state-value function, V(s), to compute the advantage function, A(s, a), which represents how much better an action 'a' is in a given state 's' compared to the average action. By learning to optimize the advantage function, A2C balances the exploration and exploitation trade-off, leading to more efficient learning.

Much like 'Q Actor Critic' let us derive the equation for 'Advantage Actor Critic'. Using, the V function as the baseline function, we can subtract the V value term from the Q value. This value indicates how much better it is to take a specific action compared to the average, general action at the given state. This value is called 'Advantage value' (A):

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t) \tag{3.12}$$

Using the Bellman optimality equation which shows the relationship between the Q and the V as:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})] \tag{3.13}$$

Replacing the Q value from Equation 3.13 in Equation 3.12 we get:

$$A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}) - V_v(s_t) \tag{3.14}$$

Now, substituting the A value into Equation (12) we get our Advantage Actor Critic (A2C) equation:

$$\Delta_\theta J(\theta) \sim \sum_{t=0}^{T-1} \Delta_\theta log \pi_\theta(a_t \mid s_t)(r_{t+1} + \gamma V(s_{t+1}) - V_v(s_t)) \tag{3.15}$$

Hence:

$$\Delta_\theta J(\theta) = \sum_{t=0}^{T-1} \Delta_\theta log \pi_\theta(a_t \mid s_t) A(s_t, a_t) \tag{3.16}$$

The following is a high-level explanation of the Advantage Actor-Critic algorithm:

1. Initialize the actor network with parameters θ, which represents the policy $\pi(s; \theta)$, and the critic network with parameters Ψ, which represents the value function $V(s; \Psi)$.

2. Set the number of episodes and steps for each episode.

3. For each episode:

a. Reset the environment and initialize the initial state 's'.

b. For each step within the episode:

i. Use the actor network with parameters θ to select an action 'a' based on the current state 's'.

ii. Execute the action 'a' in the environment and observe the next state s', reward 'r', and done signal (whether the episode is terminated).

iii. Use the critic network with parameters Ψ to compute the value of the current state $V(s; \Psi)$ and the value of the next state $V(s'; \Psi)$.

iv. Compute the advantage function, $A(s, a) = r + \gamma * V(s'; \Psi) - V(s; \Psi)$, where γ is the discount factor.

v. Update the critic network by minimizing the mean squared error between the computed target value and the predicted value of the current state: $L(\Psi) = (r + \gamma * * V(s'; \Psi) - V(s; \Psi))$ ^ 2

vi. Update the actor network by maximizing the objective function, which is the product of the advantage function and the log probability of the action: $J(\theta) = log(\pi(a|s; \theta)) * A(s, a)$

vii. Update the state s = s'.

c. If the episode is done, break the loop and start a new episode.

The Advantage Actor-Critic algorithm aims to improve the stability and convergence of the learning process by using the advantage function to guide the policy updates. This helps the agent to focus on actions that are significantly better than the average action, leading to more efficient exploration and exploitation of the environment.

A2C can be implemented using various function approximators, such as deep neural networks, to represent the actor and critic networks. It can also be extended to work with continuous action spaces using methods like the Deep Deterministic Policy Gradient (DDPG) or Proximal Policy Optimization (PPO).

We will be implementing Advantage Actor Critic (A2C) as discussed in Figure 2.10. Further on each learning step, we will update both the Actor parameter (with policy gradients and advantage value), and the Critic parameter (with minimizing the mean squared error with the Bellman update equation).

3.3.2 The REINFORCE Algorithm

'REINFORCE' in RL, is defined as a Monte-Carlo variant of policy gradients [69]. It is a model-free, policy-based reinforcement learning algorithm. It directly learns an optimal policy without the need for a value function estimator. The algorithm optimizes the policy by computing the gradient of the expected cumulative reward and updating the policy parameters using gradient ascent. By taking random samples, the agent collects a trajectory τ of one episode using its current policy and uses it to update the policy parameter as one full trajectory must be completed to construct a sample embedding space. As per the REINFORCE algorithm, policy parameters are updated through Monte Carlo updates (i.e., taking random samples). This introduces inherent high variability in log probabilities (log of the

policy distribution) and cumulative reward values because each trajectory, during training can deviate from each other at great degrees.

Consequently, the high variability in log probabilities and cumulative reward values will lead to noisy gradients and cause unstable learning and/or the policy distribution skewing to a non-optimal direction.

Therefore, REINFORCE is updated in an off-policy way as shown below (Figure 3.11):

1. Step by step perform a trajectory roll-out using the current policy as shown in Figure 3.8
2. Store log probabilities (of policy) and reward values at each step
3. Calculate discounted cumulative future reward at each step
4. Compute policy gradient and update policy parameter
5. Repeat 1–4

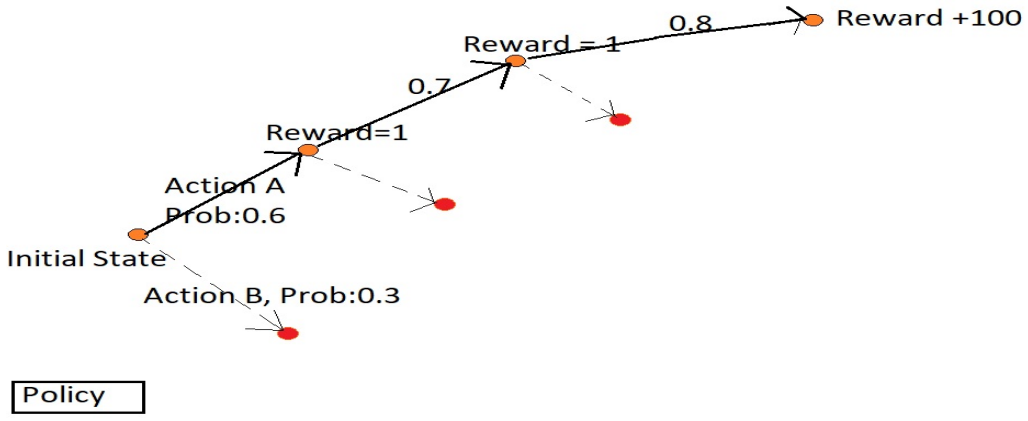


Figure 3.11: Policy - Trajectory

Here is a high-level explanation of the REINFORCE algorithm:

1. Initialize the policy network with parameters θ, which represents the policy $\pi(a|s;\theta)$.

2. Set the number of episodes and steps for each episode.

3. For each episode:

a. Reset the environment and initialize the initial state s.

b. Generate an episode trajectory by sampling actions from the policy network and executing them in the environment: $(s_1, a_1, r_1, (s_2, a_2, r_2), ..., (s_T, a_T, r_T)$, where T is the episode length.

c. For each step **t** in the episode:

i. Compute the return G_t from step t onwards:

$G_t = r_t + \gamma * r_{t+1} + \gamma \text{ ^ } 2 * r_{t+2} + ... + \gamma \text{ ^ } (T\text{-}t) * r_T$,

where γ is the discount factor.

ii. Compute the gradient of the policy network with respect to the parameters θ: $\nabla_\theta \, log(\pi(a_t|s_t;\theta))$

iii. Update the policy network parameters using the computed gradient and return G_t: $\theta = \theta + \alpha * G_t * \nabla_\theta \log(\pi(a_t|s_t;\theta))$, where α is the learning rate.

4. Repeat steps 3 and 4 until the policy converges or a stopping criterion is met.

The REINFORCE algorithm learns the optimal policy by directly optimizing the expected cumulative reward. It does not require a value function estimator or a separate critic network, making it a simple and intuitive approach to reinforcement learning. However, the algorithm can suffer from high variance in the gradient estimates, which can lead to slow convergence and instability during training.

Besides high variance of gradients, another problem with policy gradients can occur if trajectories have a cumulative reward of 0, as in that case both "good" and "bad" actions will not be learned. As we know that, the essence of policy gradient is to increase the probabilities for "good" actions and decrease probabilities of "bad" actions in the policy distribution. Overall, these issues can cause instability and slow convergence of vanilla policy gradient methods. Several techniques can be employed to improve the stability and convergence of the REINFORCE algorithm, such as using baseline functions, which are subtracted from the returns to reduce variance, or combining the REINFORCE algorithm with a value-based approach, such as in the Advantage Actor-Critic (A2C) method or create smaller gradients, and thus smaller and more stable updates, thereby intuitively, making the cumulative reward smaller.

3.3.3 Twin Delayed Deep Deterministic Policy (TD3) Gradients

Twin Delayed Deep Deterministic Policy Gradients (TD3) is an advanced model-free, off-policy reinforcement learning algorithm designed for continuous control tasks. TD3 builds upon the Deep Deterministic Policy Gradient (DDPG) algorithm, introducing several key improvements that address challenges associated with overestimation bias and noise sensitivity in the value function estimation. The algorithm employs three main enhancements: twin Q-networks, delayed policy updates, and target policy smoothing.

In TD3, two separate Q-networks, also known as twin Q-networks, are utilized to estimate the action-value function. The minimum of the predicted Q-values from both networks is used to compute the target value, mitigating the overestimation bias issue that can arise from using a single Q-network. This modification

results in a more stable and accurate value function estimation, leading to improved performance and faster convergence.

The second improvement, delayed policy updates, involves updating the policy network less frequently than the Q-networks. This technique reduces the variance of the policy updates, allowing the Q-networks to provide more accurate value estimates before the policy network is updated.

Finally, target policy smoothing adds noise to the target action before computing the target Q-value. This approach encourages the algorithm to explore a wider range of actions, making it more robust to noise in the policy network and less susceptible to overfitting.

It is well established that deterministic policy methods have a tendency to produce target values with high variance when updating the critic. We need a strategy for smoothing the target policy as this is caused by overfitting to spikes in the value estimate. TD3 [67] addresses this issue by focusing on reducing the overestimation bias by taking care of 3 important features:

1. Use of pair of critic networks
2. Delayed updates of the actor
3. Action noise regularization

TD3 is still one of the most used algorithms and can provide excellent results in continuous problem space such as robotics and autonomous driving. But it has its own drawbacks and much like many RL algorithms, training it can be unstable and heavily reliant on finding the correct hyper parameters for the given task. This is caused by the algorithm continuously over-estimating the Q values of the critic (value) network. Over the time these estimation errors build up and can lead to the agent falling into a local optima or experience catastrophic forgetting. Fujimoto [67]

has very effectively explained TD3 algorithm and provided detailed information on its key improvements over the Deep Deterministic Policy Gradient (DDPG) algorithm. The authors also explains the concepts of twin Q-networks, delayed policy updates, and target policy smoothing, and demonstrate the algorithm's effectiveness through various experiments and comparisons with other algorithms (Figure 3.12 shows an overview of TD3's main features).

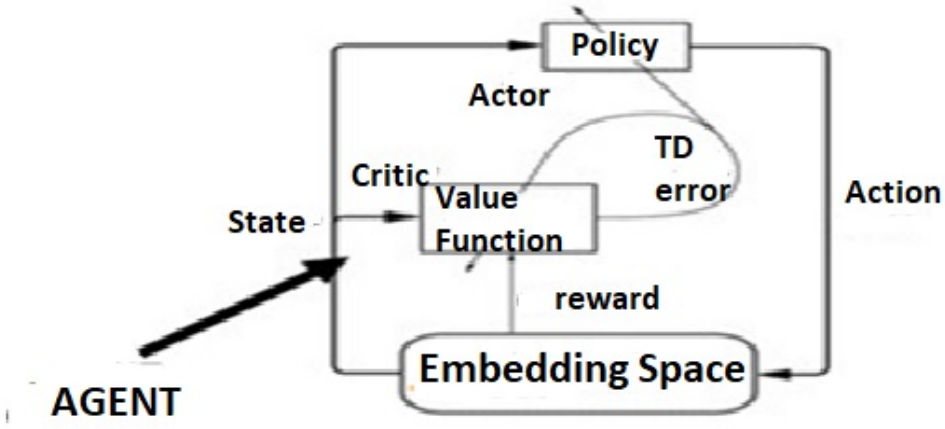


Figure 3.12: TD3 in Action

3.3.3.1 Use of a pair of critic networks

In the Twin Delayed Deep Deterministic Policy Gradients (TD3) algorithm, a pair of critic networks, also known as twin Q-networks, is used to address the issue of overestimation bias in value function estimation. Overestimation bias can lead to suboptimal policies and slow convergence, as the algorithm might overvalue certain actions, resulting in an inaccurate representation of the true action-value function.

The twin Q-networks in TD3 are two separate neural networks that independently approximate the action-value function Q(s, a). When updating the target value, TD3 computes the Q-values for the next state s' and action a' using both Q-networks and takes the minimum of the two estimates. This minimum value is then used to compute the target value for updating the Q-networks.

The rationale behind using the minimum of the two Q-values is that it mitigates the overestimation bias that can arise from using a single Q-network. By taking the minimum estimate, TD3 ensures that it avoids overly optimistic Q-value predictions, which can result from random errors or function approximation errors in the neural networks.

This technique of using twin Q-networks leads to a more stable and accurate value function estimation, which in turn improves the overall performance of the TD3 algorithm and results in faster convergence. The use of twin Q-networks is one of the key differences between TD3 and its predecessor, the Deep Deterministic Policy Gradient (DDPG) algorithm, which uses a single critic network. As shown in Figure 2.10 we need to use two critic networks estimating the current Q value using a separate target value function, thus reducing the bias.

3.3.3.2 Delayed updates of the actor

Delayed updates of the actor network are used to improve the stability of the learning process and address the issue of noisy value function estimates, which can lead to poor policy updates. In TD3, the policy network (actor) is updated less frequently than the Q-networks (critics). Typically, the actor is updated after a certain number of critic updates (e.g., updating the actor every two critic updates).

This delay in actor updates allows the critic networks to provide more accurate value estimates before the policy is updated.

The rationale behind delayed updates is that the critic networks, which estimate the action-value function, are susceptible to noise and errors, especially in the early stages of learning. If the actor network is updated too frequently based on these noisy value estimates, it can lead to suboptimal policy updates, causing the learning process to become unstable and potentially diverge.

By delaying the updates of the actor network, the TD3 algorithm allows the critic networks to learn more accurate value estimates before updating the policy. This reduces the impact of noisy value function estimates on the learning process, leading to improved stability, convergence, and overall performance of the TD3 algorithm.

Delayed updates of the actor network are one of the key improvements introduced in the TD3 algorithm, along with the use of twin Q-networks and target policy smoothing, which collectively address the challenges faced by the Deep Deterministic Policy Gradient (DDPG) algorithm, its predecessor.

To get enhanced stability during agent's training we should use Target networks to help us with delayed updates of the actor. However, in the case of actor critic methods there are some issues to this technique caused by the interaction between the policy (actor) and critic (value) networks. The training of the agent diverges when a poor policy is overestimated. The agent's policy will then continue to get worse as it is updating on states with a lot of error as shown in Figure 2.10.

In order to fix this issue, we simply need to carry out updates of the policy network less frequently than the value network. This allows the value network to become more stable and reduce errors before it is used to update the policy network. In practice, the policy network is updated after a fixed period of time

steps, while the value network continues to update after each time step. These less frequent policy updates will have value estimate with lower variance and therefore should result in a better policy. Based upon the assessment above, we only update it every 2-time steps instead of after each time step, resulting in more stable and efficient training.

3.3.3.3 Action noise regularization

In TD3, action noise regularization, also known as target policy smoothing, is employed to address the issue of noise sensitivity in the value function estimation and to encourage exploration during the learning process. This technique makes the algorithm more robust against noise in the policy network and less prone to overfitting. Therefore, when updating the target value for the Q-networks, noise is added to the target action, which is obtained from the target actor network. This noise is typically Gaussian noise with a clipped range to ensure that the perturbed action remains within the valid action bounds. The target Q-value is then computed using the smoothed target action rather than the original target action.

Please note, ideally there would be no variance between target values, with similar actions receiving similar values. By deploying a regularization strategy, we can reduce this variance by adding a small amount of random noise to the target and averaging over several mini batches. This added range of noise is later clipped in order to keep the target value close to the original action. By adding this additional noise to the value estimate, policies tend to be more stable as the target value is returning a higher value for actions that are more robust to noise and interference. Clipped noise is added to the selected action when calculating the targets to get higher values for actions that are more robust.

The rationale behind action noise regularization is that it helps the algorithm to explore a wider range of actions during learning, which can prevent overfitting to a narrow set of actions and lead to a more robust and generalizable policy. Additionally, by adding noise to the target action, the TD3 algorithm addresses the issue of value function overestimation that can arise due to the sensitivity of the critic network to small perturbations in the action space.

Target policy smoothing is one of the key improvements introduced in the TD3 algorithm, along with the use of twin Q-networks and delayed actor updates. These enhancements collectively address the challenges faced by the Deep Deterministic Policy Gradient (DDPG) algorithm and contribute to the improved stability, convergence, and performance of the TD3 algorithm in continuous control tasks.

3.3.3.4 The TD3 algorithm

As discussed, the main enhancements in TD3 are twin Q-networks, delayed policy updates, and target policy smoothing as discussed above. TD3 employs two separate Q-networks, or critic networks, to estimate the action-value function. The minimum of the predicted Q-values from both networks is used to compute the target value. This technique mitigates overestimation bias, leading to a more stable and accurate value function estimation. In TD3, the policy network, or actor network, is updated less frequently than the Q-networks. This approach reduces the variance of policy updates, allowing the Q-networks to provide more accurate value estimates before the policy network is updated. TD3 adds noise to the target action before computing the target Q-value. This encourages the algorithm to

explore a wider range of actions, making it more robust to noise in the policy network and less susceptible to overfitting.

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyperparameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks:

Trick One: Clipped Double-Q Learning. TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Trick Two: "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Trick Three: Target Policy Smoothing. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Together, these three tricks result in substantially improved performance over baseline DDPG.

TD3 concurrently learns two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, by mean square Bellman error minimization, in almost the same way that DDPG learns its single Q-function. To show exactly how TD3 does this and how it differs from normal DDPG, we'll work from the innermost part of the loss function outwards. For target policy smoothing, actions used to form the Q-learning target are based on the target policy, $\mu_{\theta_{\text{targ}}}$, but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid

action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$).

The target actions are thus:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

Target policy smoothing essentially serves as a regularizer for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

Next: clipped double-Q learning. Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

and then both are learned by regressing to this target: Using the smaller

$$L(\phi_1, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\text{E}}\left[\left(Q_{\phi_1}(s,a) - y(r,s',d)\right)^2\right],$$

$$L(\phi_2, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\text{E}}\left[\left(Q_{\phi_2}(s,a) - y(r,s',d)\right)^2\right].$$

Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

Lastly the policy is learned just by maximizing Q_{ϕ_1}: $\max_\theta \underset{s\sim\mathcal{D}}{\text{E}}\left[Q_{\phi_1}(s, \mu_\theta(s))\right]$, which is pretty much unchanged from DDPG. However, in TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

Exploration vs. Exploitation: TD3 trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training. (We do not do this in our implementation, and keep noise scale fixed throughout.)

At test time, to see how well the policy exploits what it has learned, we do not add noise to the actions. Also my TD3 implementation uses a trick to improve exploration at the start of training. For a fixed number of steps at the beginning (set with the start_steps keyword argument), the agent takes actions which are sampled from a uniform random distribution over valid actions. After that, it returns to normal TD3 exploration.

The TD3 algorithm can be implemented in following steps (Figure 3.13):

1. Initialize actor and critic networks, as well as their corresponding target networks.
2. Collect experience from the environment by executing actions according to the current policy.
3. Store the experience (state, action, reward, next state, done) in a replay buffer.
4. Sample a batch of experiences from the replay buffer.
5. Update the twin Q-networks using the minimum Q-value from both target networks and the target policy with added noise.

6. Update the actor network using the policy gradient, but with delayed updates, which means updating the actor network less frequently than the critic networks.
7. Update the target networks using a soft update strategy, which involves slowly blending the target networks with the main networks.
8. Repeat steps 2 through 7 until the desired performance is achieved or a stopping criterion is met.

In summary, TD3 is an off-policy algorithm and can only be used for environments with continuous action spaces. The Spinning Up implementation of TD3 does not support parallelization. TD3 offers improved stability, convergence, and performance in continuous control tasks compared to its predecessor, DDPG, by incorporating twin Q-networks, delayed policy updates, and target policy smoothing.

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\begin{aligned} \phi_{\text{targ},i} &\leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2 \\ \theta_{\text{targ}} &\leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta \end{aligned}$$

      end if
    end for
  end if
until convergence

Figure 3.13: Step-Wise TD3 Algorithm

CHAPTER 4

Implementation

4.1 Target Problems and Feature Representation

Much like the NOA architecture by Luo [1], the initial training here was focused on fully connected networks up to a certain size, along with some performance data (network accuracy) and network status (legal/illegal) to help construct a simpler modification space. This resulted in defining the network space as fully connected networks with a tanh activation function and softmax outputs. Classification problems from the UCI data set [74] were used with a default reward function as an improvement in performance (network accuracy) due to the change in the network in embedding space.

Each dataset represents separate entities or problems and will help us to run trajectories to optimize one problem at a time. For our training we have used a random initial policy to build trajectories in embedding space. To study the performance and operation of the different network components, we chose here to go through a sequence of pre-training steps which focus on different components and then studied the results before training the next component.

We selected the following features of a set of 9 training problems from the UCI repository [74] and they were saved them in embedding space along with network accuracy and network status.

1. *NoOfAttributes* : Number of attributes in a problem set.
2. *NoOfClasses* : Number of possible resulting classes in a problem set.
3. *DataSetSize* : size of dataset.

4. *AttributeType* : Integer, Real
5. *EntropyLabel* : Average level of "information"
6. *AvgEntrophyFeatures* : Average Entrophy of Features
7. *AvgCorelationBetFeatures* : Average Correlation Between Features
8. *2_6_8_TrainingAccuracy* : Network Training Accuracy with 2 layers with layer 1 has 6 nodes and layer 2 has 8 nodes
9. *2_6_8_TestAccuracy* : Network Test Accuracy with 2 layers with layer 1 has 6 nodes and layer 2 has 8 nodes
10. *1_5_0_TrainingAccuracy* : Network Training Accuracy with 1 layer with layer 1 has 5 nodes
11. *1_5_0_TestAccuracy* : Network Test Accuracy with 1 layers with layer 1 has 5 nodes
12. *2_6_20_TrainingAccuracy* : Network Training Accuracy with 2 layers with layer 1 has 6 nodes and layer 2 has 20 nodes
13. *2_6_20_TestAccuracy* : Network Test Accuracy with 2 layers with layer 1 has 6 nodes and layer 2 has 20 nodes

Figure 5.2 shows normalized Feature Vectors.

4.2 Implementation of Architecture and other Key Components

The implementation and initialization of architecture components are critical aspects of any research project, as they lay the foundation for the overall system's structure and functionality. The process begins with a well-defined architectural design (Figure 3.1) that outlines the components, their relationships, and their interactions. In the implementation phase, we translate the architectural design into executable code by creating the necessary classes, interfaces, and methods for

Problems (From UCI DATASET)	FEATURE VECTORS (Normalized)												
Features ⇒ Problems ⇓	No Of Attributes	No Of Classes	Data Set Size	Attribute Type	Entropy Label	Avg Entropy Features	Avg Correlation Between Features	2_6_8 Training Accuracy	2_6_8 Test Accuracy	1_5_0 Training Accuracy	1_5_0 Test Accuracy	2_6_20 Training Accuracy	2_6_20_ Test Accuracy
FV1 (AreM)	0.75	0.62	0.56	0	0.93	0.43	0.50	0.72	0.72	0.69	0.69	0.73	0.75
FV2 (Somerville Happiness)	0.75	0	0.17×10^{-2}	1	0.99	0	0.73	0.85	0.45	0.72	0.65	0.96	0.75
FV3 (Activity Recognition Healthy older People)	1	0.25	1	0	0	0.16	0.21	0.97	0.97	0.97	0.96	0.97	1
FV4 (Banknote Authentication)	0.25	0	0.02	0	0.98	0.44	0	1	1	0.99	0.98	1	0.25
FV5 (Basketball)	1	0.37	0.70	0	0.91	0.58	0.28	0.6	0.69	0.56	0.57	0.75	1
FV6 (BloodTransfusion)	0.25	0	0.97×10^{-2}	0	0.43	1	1	0.82	0.80	0.79	0.78	0.79	0.25
FV7(Caesarian)	0.5	0	0.08×10^{-2}	1	0.72	0.15	0.48	0.89	0.75	0.79	0.44	0.91	0.5
FV8(Container CraneController)	0	1	0	0	0.93	0.34	0.29	0.90	0	0.90	0	0.28	0
FV9(Cryotherapy)	0.75	0	0.09×10^{-2}	0.5	1.00	0.31	0.54	1	0.94	1	0.94	1	0.75
FV10 (Test 1)-(Ecoli)	1	0.75	0.42×10^{-2}	0	0.27	0.36	0.61	0.91	0.80	0.92	0.76	0.92	1
FV11 (Test 2)-(Car)	0.75	0.25	0.02×10^{-2}	1	1.01	0.16	0.29	0.90	0.88	0.85	0.81	0.95	0.75
FV12 (Test 3)-(VertebralColumn)	0.75	0	0.39×10^{-2}	0	0.78	0.82	0.93	0.86	0.89	0.84	0.90	0.86	0.75

Figure 4.1: Layered Graph Actor Network

each component. To ensure maintainability and scalability, it is essential to follow best practices in software engineering, such as modularity, separation of concerns, and encapsulation. This approach allowed easier debugging, training, testing, and future enhancements.

Once the components have been implemented, the initialization phase commences. During this stage, the various components are instantiated, configured, and connected according to the design specifications. This process may involve setting initial values, establishing communication channels between components, and registering dependencies or services.

It is crucial to pay careful attention to component initialization, as errors or misconfigurations can lead to unexpected behavior, performance issues, or system failures. To mitigate these risks, we employed thorough testing strategies, such as unit testing, integration testing, and end-to-end testing, to ensure that each

component functions as intended and interacts correctly with other components in the system.

In summary, the successful implementation and initialization of architecture components are vital to building robust, maintainable, and scalable software systems. By adhering to best practices in software engineering and rigorously training and testing each component, we created a solid foundation for this research project, ensuring proof of concept, long-term success and adaptability to evolving requirements.

The high-level flow as explained above is a continuous cycle, where the outcomes of each step inform the next, allowing for constant improvement and adaptation to changing circumstances. Initialization and implementation steps are shown as:

⇒ Initialise networks

⇒ Initialise replay buffer

⇒ Select and carry out action with exploration noise

⇒ Store transitions

⇒ Update critic

⇒ Update actor

⇒ Update target networks

⇒ Repeat until sentient

4.2.1 Initial Setup

This is a fairly standard set up for all Encoder, Decoder, Accuracy, Actor and Critic networks. The dimensions of the network input and output layers for actor and critic must match the dimension of the corresponding environment observation and action channels, respectively. Networks for single-output Q-value

function critics (such as the ones used in TD3 agents) must take both observations and actions as inputs.

For actor networks, the dimensions of the input layers must match the dimensions of the environment observation channels and the dimension of the output layer must have a single output layer with an output size matching the dimension of the action space defined in the environment action specification. Networks used in this case have deterministic actors with a continuous action space (such as the ones in DDPG and TD3 agents) . Since the output of an actor network must represent the probability of executing each possible action, sigmoid as an activation function is added as a final output layer. When computing the action, the actor then randomly samples the distribution to return an action.

Determining the number, type, and size of layers for deep neural network can be difficult and is application dependent. However, the most critical component in deciding the characteristics of the function approximator is whether it is able to approximate the optimal policy or discounted value function for the application, that is, whether it has layers that can correctly learn the features of observation, action, and reward signals.

In General following tips are recommended when constructing the network.

1. For continuous action spaces, bound actions with a tanh Layer followed by a Scaling Layer to scale the action to desired values, if necessary.
2. Deep dense networks with relu Layer layers can be fairly good at approximating many different functions. Therefore, they are often a good first choice.
3. Start with the smallest possible network that can approximate the optimal policy or value function.
4. If strong non-linearities or systems with algebraic constraints are observed then, adding more layers is often better than increasing the number of out-

puts per layer. In general, the ability of the approximator to represent more complex (compositional) functions grows only polynomially in the size of the layers, but grows exponentially with the number of layers. In other words, more layers allow approximating more complex and nonlinear compositional functions, although this generally requires more data and longer training times. Given a total number of neurons and comparable approximation tasks, networks with fewer layers can require exponentially more units to successfully approximate the same class of functions, and might fail to learn and generalize correctly.

5. For on-policy agents; the ones that learn only from experience collected while following the current policy parallel training works better if networks are large (for example, a network with two hidden layers with 32 nodes each, which has a few hundred parameters). On-policy parallel updates assume each worker updates a different part of the network, such as when they explore different areas of the observation space. If the network is small, the worker updates can correlate with each other and make training unstable.

4.2.2 Implement and Initialize Encoder-Decoder-Accuracy Network's

Our initial Encoder and Decoder models are designed with 4 dense layers each, where the Encoder takes network properties such as number of layer, nodes in each layer etc. and uses tanh activation function to output a unique embedding in our embedding space. The Decoder model takes an output of the Encoder model as its input and outputs network properties using a sigmoid activation function. We trained Encoder-Decoder together. Planning for the Accuracy model was simply through trial and error and we settled with 9 dense layers and two output layers using sigmoid activation function to output accuracy and status(legal/illegal) of

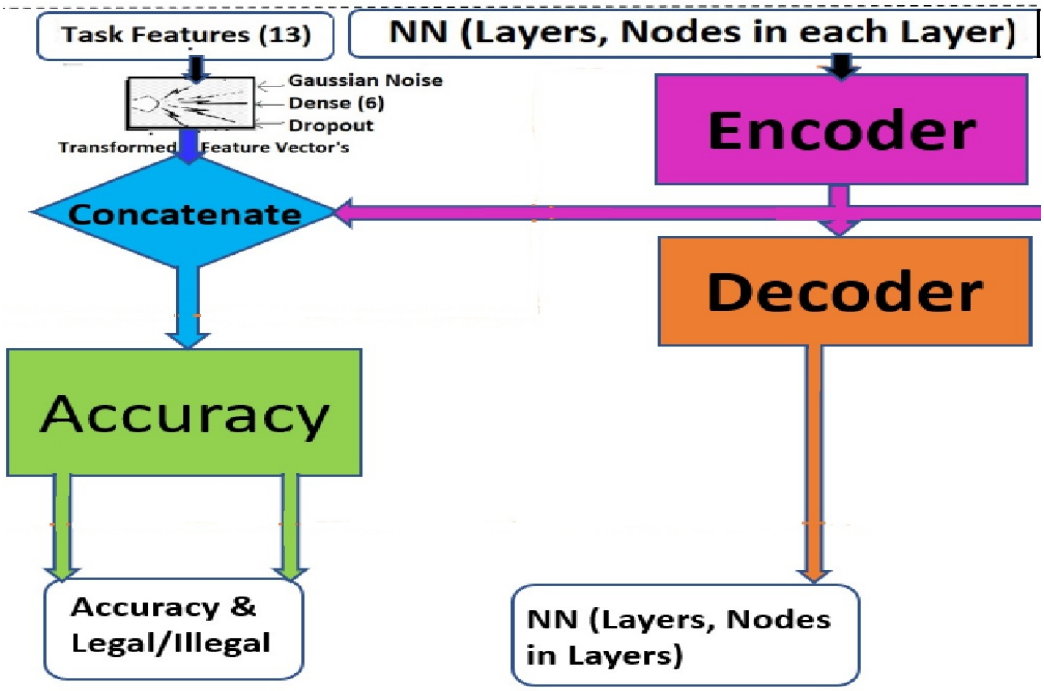


Figure 4.2: Initial Encoder-Decoder-Accuracy Model

the network as shown in Figure 3.4. Input layer for the Accuracy model takes the state vector, which as output from the first Concatenation layer consist of network properties and its 6 transformed Feature Vectors. We run network properties and Feature vectors in parallel through sets of dense layers to yield the best results (Figure 4.2).

Once we achieved our research goals with the very limited set of networks we further generalized our network architecture beyond the simplified set of fully connected networks. The goal is to optimize across a wider set of network types to ultimately facilitate architectures such as convolutional, recurrent networks etc. This can be achieved by introducing the modular network component input in terms of the characterization of the network, instead of just passing number of

layers and number of nodes in each layer. This required to change the input vocabulary, to be able to input different kinds of complex networks.

4.2.2.1 Sequence-to-Sequence (seq2seq) model for autoencoder

Encoder and decoder for ore complex fully connected networks were implemented and trained using a sequence-to-sequence (seq2seq) model to predict the architecture of a neural network given its time series data. Figure 4.3 shows a layout of the input structure for networks with a maximum of 5 layers. The model employs a bidirectional LSTM encoder-decoder architecture with attention mechanisms and can be straightforwardly extended to more complex network architectures by changing the description language. The code covers data preprocessing, model creation, and training. The code is designed for training an autoencoder-based neural network model to predict neural network architectures using time series data.

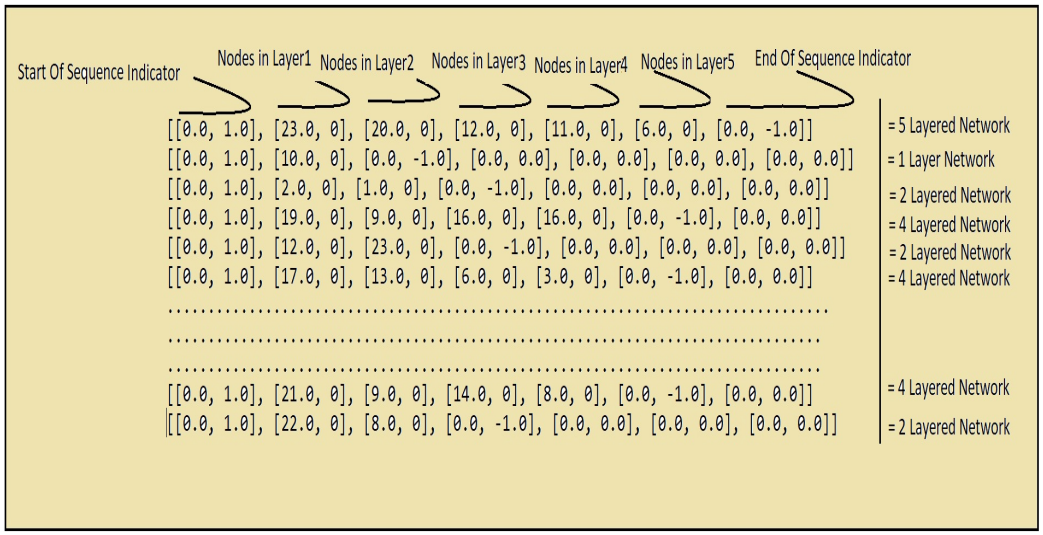


Figure 4.3: Input Network Data in Time Series Sequence

There are three main functions defined in the code:

a. decodePredict(): This function decodes the predicted output sequence from the decoder model using the encoder model's output (embeddings) as input. The decoding process stops when the termination character is found or the length of the decoded sequence exceeds the maximum length.

b. normalize(): This function normalizes the given Data Frame's specified features based on the given max and min values for each feature.

c. deNormalizeRound(): This function denormalizes and rounds the given Data Frame's specified normalized features based on the given max and min values for each feature.

Data preprocessing: The code reads time series data and ground truth values from CSV files. Data normalization is performed on the time series data and ground truth values to transform them into a suitable range.

4.2.2.2 Model Creation

The model is built using a combination of encoder and decoder networks.

a. Bidirectional LSTM-based Encoder Model: The Bidirectional LSTM-based Encoder Model is a part of the autoencoder network used in the code. It is a combination of a forward LSTM and a backword LSTM, which can fit the data from both forward direction and backward direction, and concatenate the prediction. Standard LSTM can only fit the time-related data from one direction. BiLSTM added a reverse directional LSTM so that BiLSTM can capture the patterns that may be ignored by LSTM. The structure of BiLSTM is shown in Figure 4.4. The top L_i represents the forward LSTM while the bottom L_i' represents the reverse

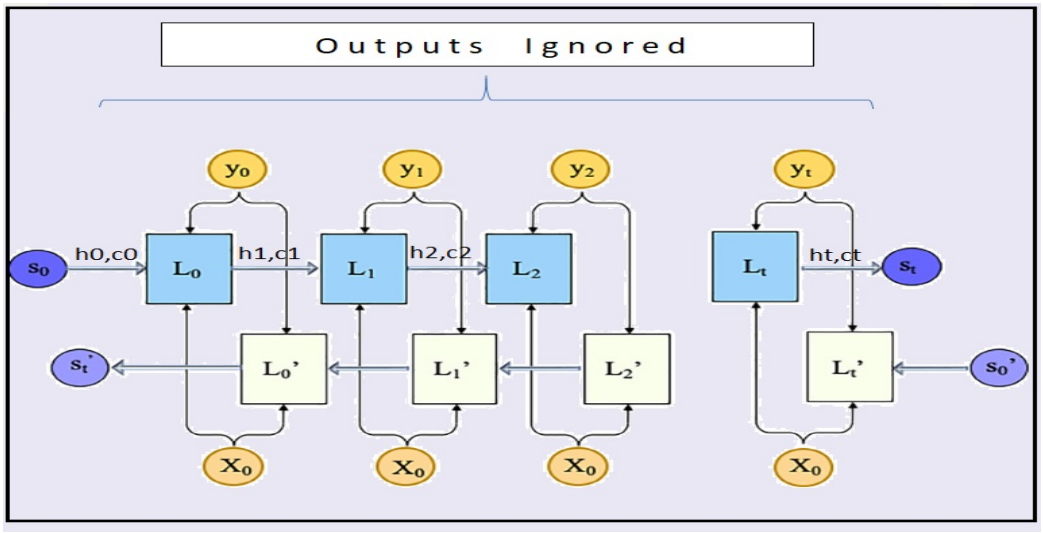


Figure 4.4: Bidirectional LSTM based Encoder Model

directional LSTM, and s and s' is the time series information delivering in LSTM cells.

It is implemented to be responsible for encoding the input time series data into a lower-dimensional representation, which can be later decoded by the decoder network to predict the neural network architecture. The model consists of an input layer followed by three bidirectional LSTM layers, two dense layers, and an output layer (the embeddings). The input to the encoder is a time series with shape (timesteps, number of features). The output of the encoder is an embedding.

Here's a detailed explanation of the BiLSTM Based Encoder Model:

Input Layer: The input layer takes time series data with the shape (timesteps, n_features). timesteps and n_features are derived from the preprocessed data.

Bidirectional LSTM Layers: The encoder network consists of three Bidirectional LSTM layers. These layers can capture both forward and backward dependencies in the input sequence. The first Bidirectional LSTM layer has 80 units and is followed by two additional Bidirectional LSTM layers with 30 and 20 units, respectively. The

first two layers return sequences, while the last layer returns only the final hidden state.

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) architecture that can learn long-term dependencies in sequence data through the use of gating units. Bidirectional LSTMs process the input sequence in both forward and backward directions, making them more effective at capturing complex patterns.

Dense Layers: After the Bidirectional LSTM layers, the encoder has two Dense layers with 30 and 2 units, respectively. These layers have ReLU (Rectified Linear Unit) and Tanh activation functions, respectively. Dense layers, also known as fully connected layers, are used to combine the features learned by the previous layers and create a lower-dimensional representation (embedding) of the input sequence. The final Dense layer with 2 units is the output of the encoder, which serves as the input to the decoder network.

The Encoder Model takes the input sequence (Figure 4.3), processes it through the Bidirectional LSTM layers, and summarizes the information in something called the internal state or context vector (in case of LSTM these are called the hidden state and cell state vectors). We discard the outputs of the encoder and only preserve the internal states. This context vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions and generate an embedding that represents the input sequence in a lower-dimensional space.

This embedding is then used as input to the Decoder Model to predict the neural network architecture.

The hidden states h_i are computed using the formula:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t) \tag{4.1}$$

The LSTM reads the data, one item after the other. Thus if the input is a sequence of length 't', we say that LSTM reads it in 't' time steps.

1. X_i = Input sequence at time step i.
2. h_i and c_i = LSTM maintains two states ('h' for hidden state and 'c' for cell state) at each time step. Combined together these are internal state of the LSTM at time step i.
3. Y_i = Output sequence at time step i. Y_i is actually a probability distribution over the entire data sequence which is generated by using a softmax activation. Thus each Y_i is a vector of size "sequence_size" representing a probability distribution.

b. Decoder hidden model: This model takes the embeddings from the encoder as input and produces hidden states for the decoder LSTM. It has two dense layers and is compiled with the specified optimizer, loss function, and metric. The decoder predicts the neural network architecture very close to the one which was send to the embedding model with 98.9% accuracy.

c. Decoder sequence model: This model generates the output sequence from the decoder. It has an LSTM layer followed by two dense layers and an output layer. The model is compiled with the specified optimizer, loss function, and metric.

Here the decoder is an LSTM whose initial states are initialized to the final states of the Encoder LSTM, i.e. the context vector of the encoder's final cell is input to the first cell of the decoder network. Using these initial states, the decoder starts generating the output sequence, and these outputs are also taken into consideration for future outputs. A stack of several LSTM units where each predicts an output y_t at a time step t.

Each recurrent unit accepts a hidden state from the previous iteration and produces and output as well as its own hidden state.
Any hidden state h_i is computed using the formula:

$$h_t = f(W^{(hh)}h_{t-1}) \tag{4.2}$$

The output y_t at time step t is computed using the formula:

$$y_t = softmax(W^{(s)}h_t) \tag{4.3}$$

d. Decoder depth model: This model predicts the depth (number of layers) of the neural network architecture. It has two dense layers and an output layer. The model is compiled with the specified optimizer, loss function, and metric.

We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help us determine the final output.

We will add two tokens in the output sequence as follows (also shown in Figure. 4.3):

"START_Indicator" = [0,1]

"END_Of_Seq_Indicator" = [0.0, -1.0]

The most important point is that the initial states (h0, c0) of the decoder are set to the final states of the encoder. This intuitively means that the decoder is trained to start generating the output sequence depending on the information encoded by the encoder.

Finally, the loss is calculated on the predicted outputs from each time step and the errors are backpropagated through time in order to update the parameters of the network. Training the network over a longer periods with a sufficiently large amount of data results in pretty good predictions.

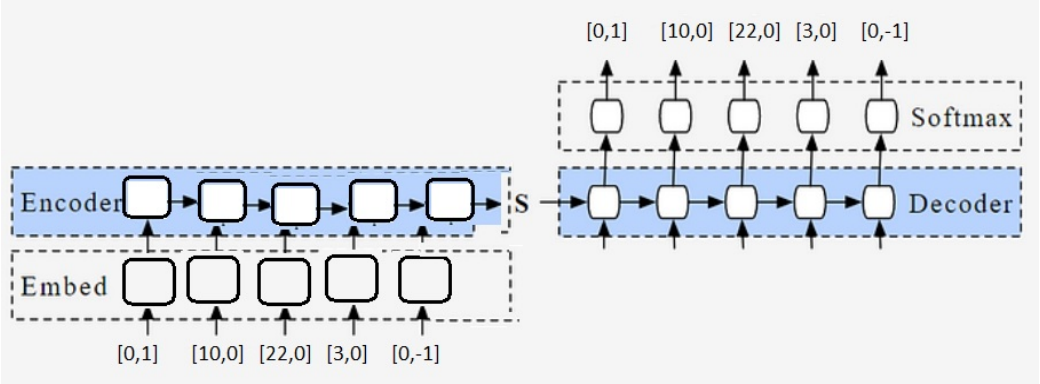


Figure 4.5: Seq-Seq-Encoder-Decoder Architecture

In Summary, as shown in Figure 4.5:

- During inference, we generate one item of the sequence (word) at a time.
- The initial states of the decoder are set to the final states of the encoder.
- The initial input to the decoder is always the START token.
- At each time step, we preserve the states of the decoder and set them as initial states for the next time step.
- At each time step, the predicted output is fed as input in the next time step.
- We break the loop when the decoder predicts the END token.

e. Main model: The main model combines the encoder, decoder hidden model, decoder seq model, and decoder depth model. It has two outputs: one for the depth prediction and one for the output sequence. The model is compiled with the specified optimizer, loss function, and metric.

Model training: The main model is trained on the preprocessed data using the specified batch size, and a checkpoint callback is used to save the best model during training. Adam optimizer with a learning rate of 0.001 is used. Mean Squared Error loss function is also effectively used during training.

The training data is split into training and validation sets with a 90/10 ratio. The model is trained using a batch size of 500 over 10000 epochs with both legal and illegal data sets and a 'Model Checkpoint' callback is used to save the best model based on validation accuracy.

The model employs bidirectional LSTM layers in the encoder and standard LSTM layers in the decoder, with attention mechanisms to improve the model's ability to capture long-range dependencies in the input data. The trained model can be used to encode the network in the embedding space and later predict (decode) the architecture of a neural network, which can potentially help in future RL training and Policy generation tasks.

The Encoder-Decoder model is pre-trained first to get best performance and then we freeze their weights to train it with the Accuracy model and finally train them all together to yield the best performance of the combined network.

4.2.3 Implement and Initialize Accuracy Network

The accuracy model that takes an 8-dimensional input, splits it into two parts, applies several dense layers with a leaky ReLU activation function to each part, and then combines them into an output with a sigmoid activation function. The output is then split into two parts again, one for accuracy and one for legality of the given network. The accuracy model is compiled with a custom loss function and several metrics.

Since we are using a Siamese Network to co-locate legal networks therefore in the end we concatenate the output of the two encoder models, calculates the Euclidean distance between the embeddings using a lambda layer, and concatenates the results with the final predictions from the two decoder models. The final Encoder-Decoder-Accuracy model is compiled together with a custom loss func-

tions and several metrics. The model is then trained on a training set and validated on a validation set using an optimizer and callbacks to save the best weights.

The layer diagram for the Accuracy network is shown in Figure 6.8

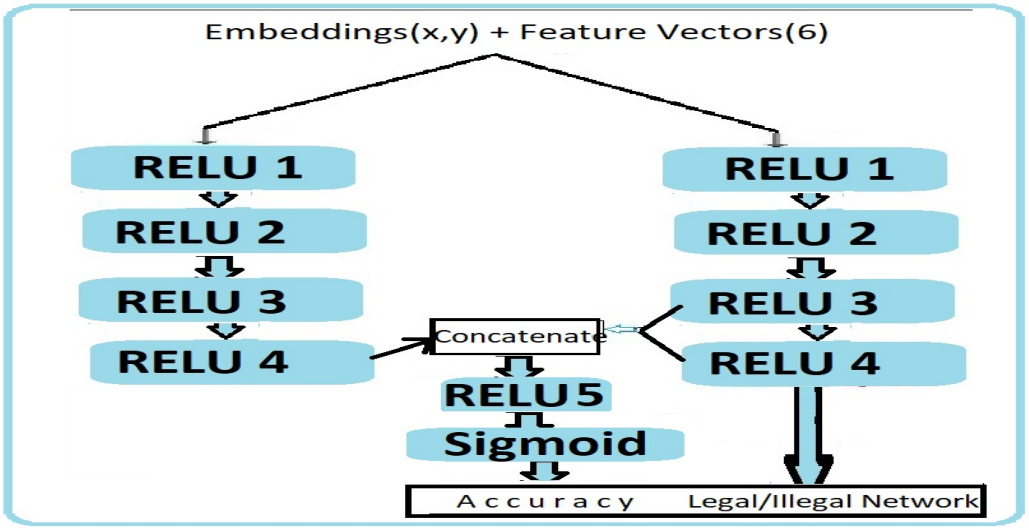


Figure 4.6: Accuracy Network Layers

4.2.4 Implement and Initialize Actor Network

After a few experiments, the Actor Network was commissioned with four dense layers and one lambda_function layer. It uses sigmoid as an activation function to also output distance of the directional vector. The Input layer takes the state vector, an output from the first Concatenation layer. This is a standard setup for the Actor network where we have also used a Lambda function to normalize our two-dimension vector and the third dense layer returns the length of that vector (Figure 4.7).

The Actor directly maps embedding states which includes network architecture embeddings and the Feature Vector embedding of the target problem asso-

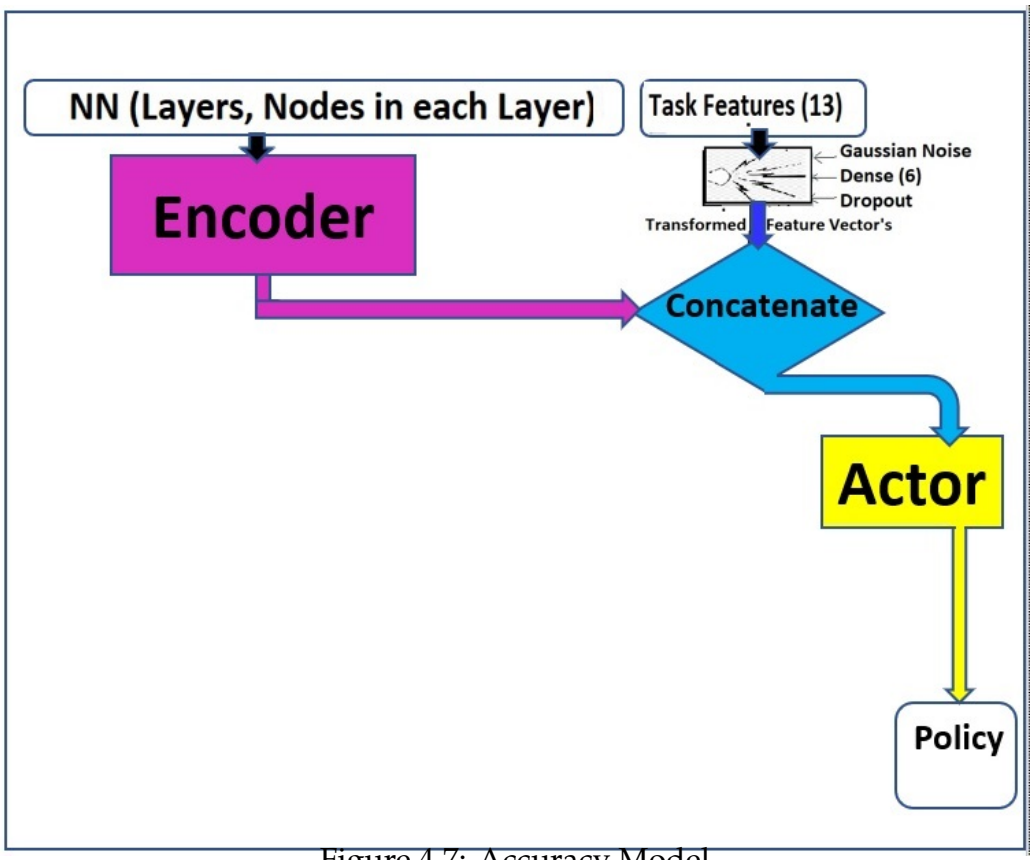


Figure 4.7: Accuracy Model

ciated with the network to continuous actions (Figure 4.7). The network modification action is chosen as the output from this network. If the Actor is in training(exploration) mode then some random normal noise is added to this action, otherwise, i.e. if the model is in test mode, the action output represents the pure deterministic output. The target value for critic loss is calculated by predicting the action for the next states using the actor_target network and then using these actions to get the next state's value in the embedding space using the critic_target network.

To make sure that the action returned by the Actor Model is within the boundary of $+/-1$, the action is clipped to make sure no illegal action is passed back to the training environment. This helps to keeps the target value close to

the original action. Also, as per TD3 (Figure 2.10), the Actor network is deployed in pairs as actor_main and actor_target. Delayed update of the actor network is provisioned, thereby only updating it every second time steps instead of after each time step, resulting in more stable and efficient training. No gradient decent is allowed on target actor network as soft update is used.

The layered graph for the Actor Network is shown in Figure 4.8.

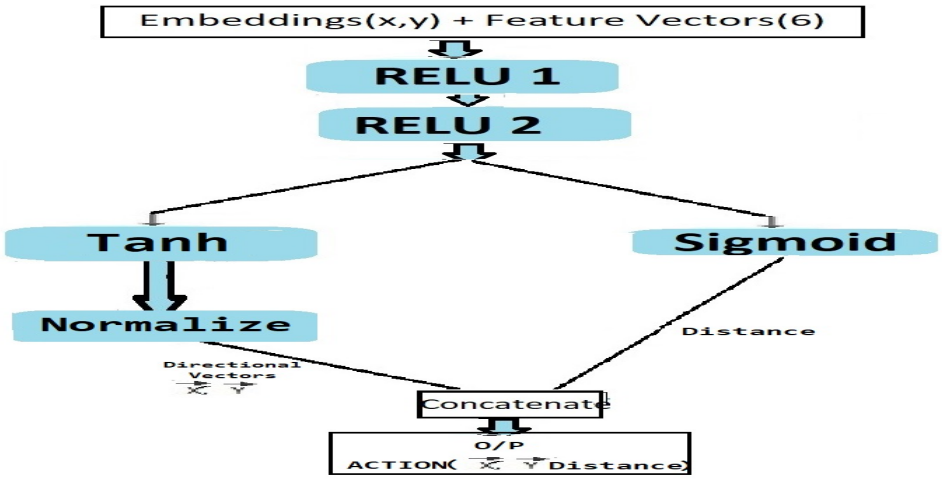


Figure 4.8: Layered Graph Actor Network

4.2.5 Implement and Initialize Critic Network

This actor_target network plays a very critical role in the loss function of the critic, along with critic_target and critic_main networks. The Critic Network is formulated with nine (9) dense layers to give two outputs using a tanh activation function. The Critic Network takes the state vector and the output of the actor network (Figure 4.9) as its input. The Critic Network does forward propagation operation and outputs Value-On-Policy (V) from input state and Action-On-Policy (A) from state and action.

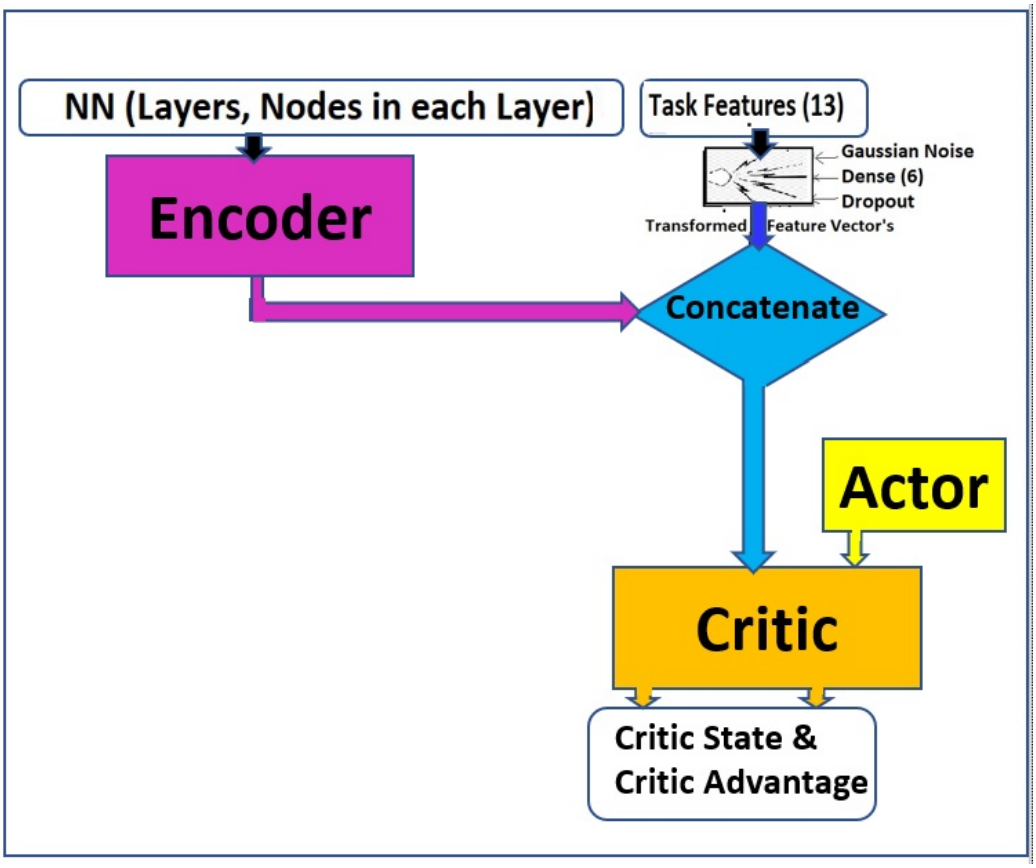


Figure 4.9: Accuracy Model

Note, critic network has almost twice dense layers as compared to Actor network. Inspired by TD3 (Figure 2.10), Critic network is deployed in two pair. This was inspired by the technique seen in Deep Reinforcement Learning with Double Q-learning [75] which involved estimating the current Q value using a separate target value function, thus reducing the bias. It uses clipped double Q learning where it takes the smallest value of the two critic networks. This method favours underestimation of Q values. This underestimation bias isn't a problem as the low values will not be propagated through the algorithm, unlike overestimate values. This provides a more stable approximation, thus improving the stability of the

entire algorithm.

The critic-pair networks run in parallel to each other and results in two separate outputs. They are initialized as separate main and target networks, also referred to as: critic_main and critic_main2, critic_target and critic_target2. The critic_target network is effectively used to create targets for training using our critic loss and to find the next state value by using the smallest value of the two critic-pair networks when forming the targets.

In value-based reinforcement learning methods such as deep Q-learning, function approximation errors are known to lead to overestimated value estimates and suboptimal policies. This problem persists in an actor-critic setting therefore a mechanism is needed to minimize its effects on both the actor and the critic. Connection can be drawn between target networks and overestimation bias, therefore it is suggested that delayed policy updates reduce per-update error leading to improved performance. Hence delayed update of the actor network is provisioned, only updating it every 2 time-steps instead of after each time step, resulting in more stable and efficient training.

Predicted (critic) values are the output of the main critic network which takes states and actions from the buffer sample as an input. Critic value is the value of the current state with respect to original state and the actions the agent actually took during the course of the episode.

The layered graph for the Critic network is shown in Figure 4.10.

When creating the deep neural networks and configuring actor or critic, the following approach is considered as a starting point.

Start with the smallest possible network and a high learning rate (0.01). Train this initial network to see if the agent converges quickly to a poor policy or acts in

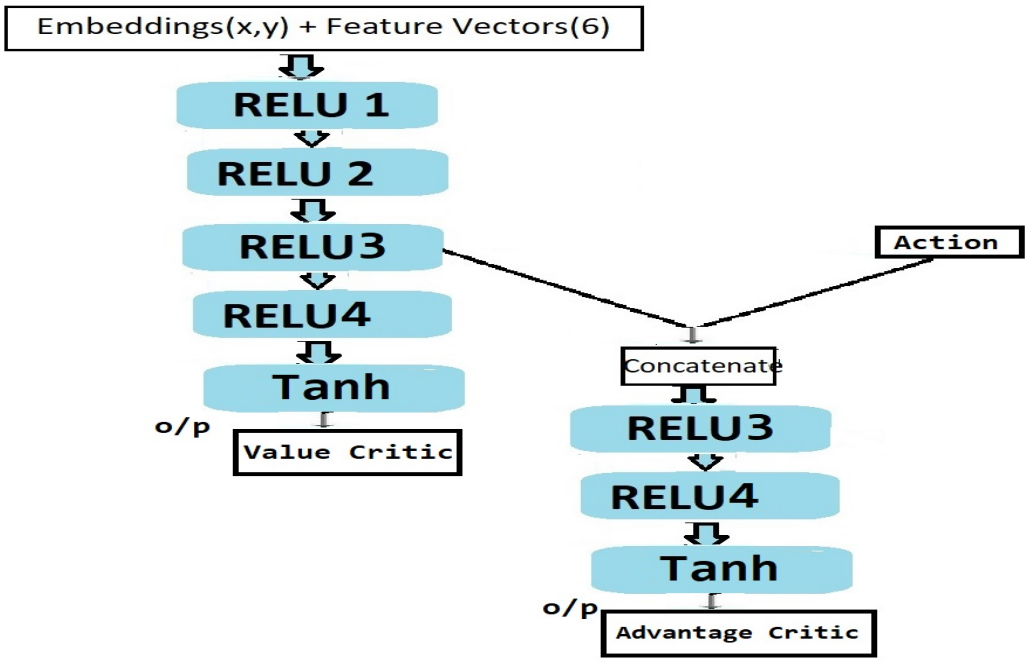


Figure 4.10: Layered Graph Critic Network

a random manner. If either of these issues occur, rescale the network by adding more layers or more outputs on each layer. The goal is to find a network structure that is just big enough, and does not learn too fast, and shows signs of learning (an improving trajectory of the reward graph) after an initial training period.

Once a good network architecture is settled on, a low initial learning rate (0.001) can allow to see if the agent is on the right track, and confirm that the network architecture is satisfactory for the problem. A low learning rate makes tuning parameters easier, especially for difficult problems.

One has to be patient with the TD3 agents, since they might not learn anything for some time during the early episodes, and they typically show a dip in cumulative reward early in the training process. Eventually, they can show signs of learning after the first few thousand episodes. Therefore for these agents, promoting exploration of the agent is critical. For agents with both actor and critic

networks, set the initial learning rates of both actor and critic to the same value such as Adam(0.001). However, for some problems, setting the critic learning rate to a higher value than that of the actor can improve learning results.

Depending on the learning algorithm, an agent maintains one or more parameterized function approximators for training the policy. Approximators can be used in two ways.

1) Critics — For a given observation and action, a critic returns the predicted discounted value of the cumulative long-term reward.

2) Actor — For a given observation, an actor returns as output the action that (often) maximizes the predicted discounted cumulative long-term reward.

Agents that use both an actor and a critic are referred to as actor-critic agents. In these agents, during training, the actor learns the best action to take using feedback from the critic (instead of using the reward directly). At the same time, the critic learns the value function from the rewards so that it can properly critique the actor. In general, these agents can handle both discrete and continuous action spaces.

4.2.6 Implement And Initialize Reinforcement Learning Agents

The goal of reinforcement learning is to train an agent to complete a task within an uncertain environment. At each time interval, the agent receives observations and a reward from the environment and sends an action to the environment. The reward is a measure of how successful the previous action (taken from the previous state) was with respect to completing the task goal.

The agent contains two components: a policy and a learning algorithm. During the policy learning it maps the current environment observation to a probability distribution of the actions to be taken. The policy is implemented within an agent,

by a function approximator with tunable parameters and a specific approximation model, such as a deep neural network.

The learning algorithm continuously updates the policy parameters based on the actions, observations, and rewards. The goal of the learning algorithm is to find an optimal policy that maximizes the expected cumulative long-term reward received during the task.

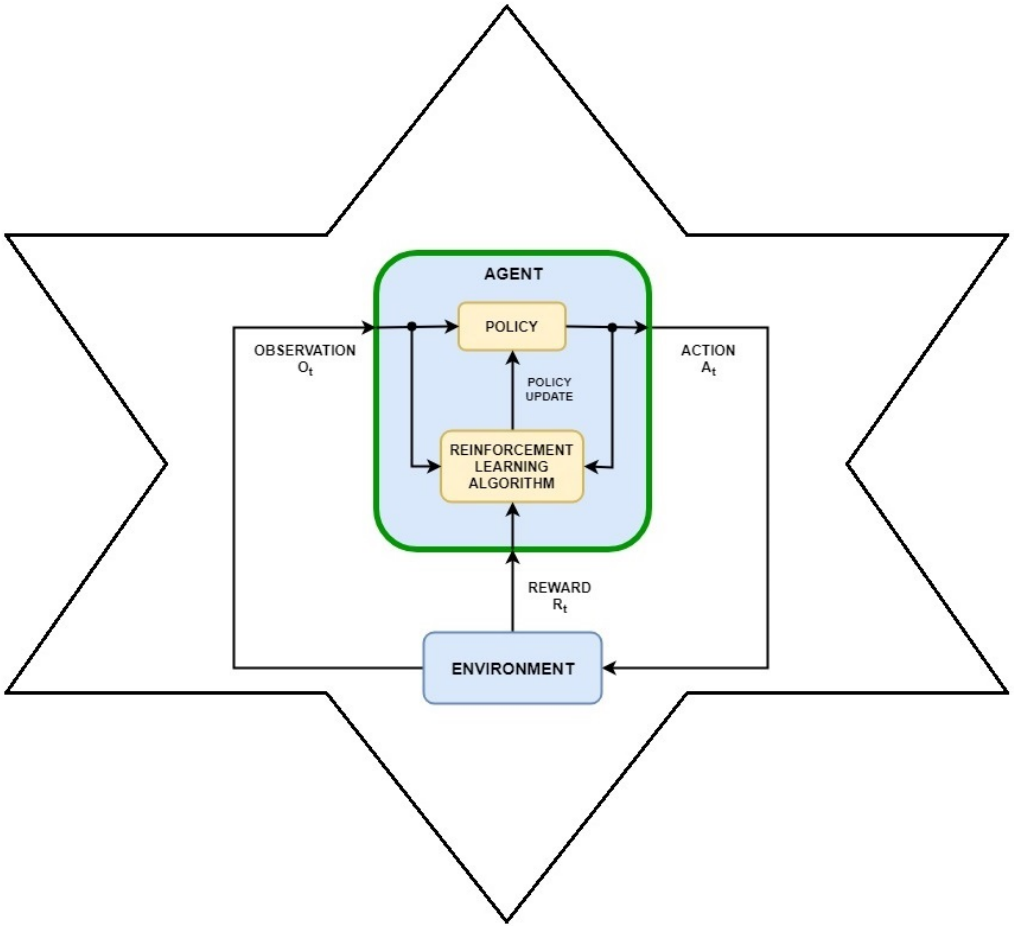


Figure 4.11: Layered Graph Critic Network

For the problems to train, a TD3 agent was used which is a type of Reinforcement Learning Agent. The TD3 agents is trained in environments with continuous

observations and actions and uses two value function critic, each estimating a V and an A function, and deterministic policy actor π(S)

The Agent class is initialized to setup the training environment for exploration for the max/min actions as the noise will be added to the output of our deep NN for some exploration. It will create default actor and critics based on the observation and action specifications from the environment. For training, batch_size is initialized to 300. Optimizer Adam is initialized to (0.001) for all networks. Gamma (γ) is needed as a discount factor for updating target for the terminal new state which is nothing but just the reward for every other state computed as reward + discount factor times the value of the resulting state. Note there is no need to do any gradient descent on both actor and critic target networks. Only soft network updates are done on these target networks to slowly move them towards the function learned by the main networks.

To create an agent, the following steps are performed:

1. Create observation specifications for training environment.
2. Agent learns two pairs of state value (V) and advantage (A) value functions and uses the minimum value function estimate during policy updates.
3. Agent updates the policy and targets less frequently than the value functions.
4. When updating the policy, agent adds noise to the target action, which makes the policy less likely to exploit actions with high value estimates.
5. Train the agent with two pairs of value functions.

4.2.6.1 Agent Learning Algorithm

Agents use the following training algorithm, in which it updates actor and critic models at each time step. To configure the training algorithm, K = 2 is used as number of critics where k is the critic index.

1. Initialize each critic $V_k(S;\phi_k), A_k(S,A;\phi_k)$ with random parameter values ϕ_k, and

initialize each target critic with the same random parameter values: $\phi_{tk} = \phi_k$

2. Initialize the actor $\pi(S\ \theta)$ with random parameter values θ_t, and initialize the target actor with the same parameter values: $\theta_t = \theta$

3. For each training time step:

a. For the current observation S, select action a = $\pi(S\ \theta)$ + N, where N is stochastic noise from the noise model.

b. Execute action a . Observe the reward R and next observation S′.

c. Store the experience (S,A,R,S′) in the experience buffer.

d. Sample a random mini-batch of M experiences (S_i, A_i, R_i, S_i') from the experience buffer.

e. If (S_i') is a terminal state, set the value function target y_i to R_i. Otherwise, set it to

$$y_i = R_i + \gamma * min(V_{tk}(S_i', clip(\pi_t(S_i';\theta_t) + \varepsilon);\phi_{tk})) \tag{4.4}$$

The value function target is the sum of the experience reward R_i and the minimum discounted future reward from the critics.

To specify the discount factor use γ.

To compute the cumulative reward, the agent first computes a next action by passing the next observation S_i' from the sampled

experience to the target actor. Then, the agent adds noise ε to the computed action using the target policy smoothing, and clips the action based on the upper and lower noise limits. The agent finds the cumulative rewards by passing the next action to the target critics.

f. At every time training step, update the parameters of each critic by minimizing the loss L_k across all sampled experiences.

$$L_k = \frac{1}{2M} \sum_{i=1}^{M} (y_i - V_k(S_i, A_i, \phi_k))^2 \tag{4.5}$$

g. Every D_1 steps, update the actor parameters using the following sampled policy gradient to maximize the expected discounted reward. To set D_1 use 2 as we only update every 2 time-steps instead of after each time step.

$$\Delta_\theta J \approx \frac{1}{2M} \sum_{i=1}^{M} G_{ai} G_{\pi i} \tag{4.6}$$

Here, G_{ai} is the gradient of the minimum critic advantage value output with respect to the action computed by the actor network, and $G_{\pi i}$ is the gradient of the actor output with respect to the actor parameters. Both gradients are evaluated for observation S_i.

h. Every D_2 steps, update the target actor and critics depending on the target update method. To specify D_2, use 2 as we only update every 2 time-steps instead of after each time step.

For simplicity, the actor and critic updates in this algorithm show a gradient update using basic stochastic gradient descent. The actual gradient update method depends on the optimizer specified as Adam which is initialized to (0.001)

One can extract a policy object from an agent to generate deterministic or stochastic actions from the policy, given an input observation which is the network architecture embedding and the embedded Feature Vector, i.e the current state of the environment. Working with policy objects can be useful for application deployment or custom training purposes.

4.2.7 Implement and Initialize replay buffer (RBuffer)

A standard replay buffer is implemented and used to store the states, actions, rewards, new states & terminal flags.

The replay buffer is effectively used to store experiences. The main three parameters for our RBuffer are:

- maxsize - max size of memory to bound it
- statedim - input shape from our environment
- naction - number of action dimensions (3) for the continuous action space. Here the dimensions represent the sine and cosine of the direction and the step size of the displacement in the embedding space.
- observation space - observation specifications for your environment
- action specifications - action specifications for the environment
- action_memory - memory size by number of actions
- reward_memory - memory size by number of rewards
- accuracy_memory - memory size by number of accuracy

4.3 Summary

The implementation of the architecture and its key components of our architecture is a complex and iterative process that requires careful consideration

and experimentation to achieve optimal results. Successful implementation of all models involves choosing appropriate architectures, optimizers, loss functions, and metrics, as well as incorporating techniques such as regularization and feature engineering. Ultimately, the goal is to create models that can learn from data and generalize well to new, unseen data, enabling them to be applied effectively in a wide range of applications.

CHAPTER 5

Training and Experiments

5.1 Introduction

Training and experiments of various components of the proposed Architecture are the cornerstone of building and evaluating the models component and collective performance. Training of these models involves feeding data into an algorithm. The data used for training is typically split into training and validation sets, with the training set used to update the model's parameters and the validation set used to evaluate its performance.

Most of my experiments involved running different iterations of training on different model components, sometimes jointly, with different combinations of hyperparameters, such as learning rate and regularization strength, to determine the best performing model/models for a given task. These experiments help to identify the optimal configuration of the model, allowing for greater accuracy and better generalization to new data. In conducting training experiments, it is important to use appropriate statistical techniques for model evaluation, to ensure that the results are robust and not overly influenced by chance and overfit in any way. Additionally, techniques such as early stopping and model checkpointing can be used to prevent overfitting and improve training efficiency.

Data preprocessing and feature engineering are also important aspects of our iterative training and experimentation, as they can have a significant impact on the performance of the model. Preprocessing involves transforming raw data into a format that can be fed into the model, such as scaling or normalization. Feature

engineering involved selecting or creating features (Figure 5.2 that are relevant to the task at hand, which can help to improve the model's accuracy.

Much like Luo[1],the initial training here was focused on fully connected networks up to a certain size, along with some performance data (network accuracy) to help construct a simpler modification space. Later, to cater more complex and variable layer Network's we updated our encoder-decoder model to ingest time series data. This resulted in defining the network space as complex networks with more variable numbers of layers with an tanh activation function and softmax outputs (Figure 4.3).

Classification problems from the UCI data set [74] were used with a default reward function in the form of the improvement in performance (network accuracy) due to the change in the network in embedding space. Each dataset will represent separate entities or problems and will help us to run trajectories to optimize one problem at a time. For our training we have used a random initial policy to build trajectories in embedding space. To study the performance and operation of the different network components, we chose here to go through a sequence of pre-training steps which focus on different components and then studied the results before training the next component.

Training and experimentation in this research required a great deal of computational resources, making it important to leverage good computing platforms and other distributed computing systems to accelerate the training process. Additionally, the use of GPUs or TPUs can greatly speed up training and allow for larger models to be used.

5.2 Training DataSet

Without high-quality training data, even the most efficient machine learning algorithms will fail to perform. For effective training of Encoder-Decoder,Accuracy,Actor and Critic models, quality, accurate, complete, and relevant data is needed. After careful search and consideration, it was decided to get some training datasets from the UCI Machine Learning Repository [74]. The UCI repository is a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms.

Data Set Characteristics:	Multivariate, Sequential,Time_series, Univariate, Domain-Theory	Number of Instances:	NUMBER	Area:	Computer, Life,Game, Business
Attribute Characteristics:	Real, Integer, Real-Integer	Number of Attributes:	NUMBER	Date Donated	Date
Associated Tasks:	Classification, Regression	Missing Values?	N/A	Number of Web Hits:	NUMBER

Figure 5.1: Problem Selection Attributes

Comment:
This table is close to unreadable on my computer

Dataset selection was primarily focused on dataset and attribute characteristics (Figure 5.1). Also, to uniquely represent each problem in an embedding space, a set of 13 problem-specific attributes that relate to the complexity of the learning problem and yet easy to compute were designed and computed to represent each problem in the training dataset. Apart from 'Number of Classes', 'Data Set Size', 'Attribute Type', we also computed Entropy on dataset labels, average Entropy and average Entropy between data set features. To give a good spread and variation;

training and test accuracy was also computed on three baseline Neural Networks with different numbers of layers and nodes in each layers (Figure 5.2).

Problems (From UCI DATASET)	FEATURE VECTORS (Normalized)												
Features ⇒ Problems ⇓	No Of Attributes	No Of Classes	Data Set Size	Attribute Type	Entropy Label	Avg Entropy Features	Avg Correlation Between Features	2_6_8 Training Accuracy	2_6_8 Test Accuracy	1_5_0 Training Accuracy	1_5_0 Test Accuracy	2_6_20 Training Accuracy	2_6_20_ Test Accuracy
FV1 (AreM)	0.75	0.62	0.56	0	0.93	0.43	0.50	0.72	0.72	0.69	0.69	0.73	0.75
FV2 (Somerville Happiness)	0.75	0	0.17×10^{-2}	1	0.99	0	0.73	0.85	0.45	0.72	0.65	0.96	0.75
FV3 (Activity Recognition Healthy older People)	1	0.25	1	0	0	0.16	0.21	0.97	0.97	0.97	0.96	0.97	1
FV4 (Banknote Authentication)	0.25	0	0.02	0	0.98	0.44	0	1	1	0.99	0.98	1	0.25
FV5 (Basketball)	1	0.37	0.70	0	0.91	0.58	0.28	0.6	0.69	0.56	0.57	0.75	1
FV6 (BloodTransfusion)	0.25	0	0.97×10^{-2}	0	0.43	1	1	0.82	0.80	0.79	0.78	0.79	0.25
FV7(Caesarian)	0.5	0	0.08×10^{-2}	1	0.72	0.15	0.48	0.89	0.75	0.79	0.44	0.91	0.5
FV8(Container CraneController)	0	1	0	0	0.93	0.34	0.29	0.90	0	0.90	0	0.28	0
FV9(Cryotherapy)	0.75	0	0.09×10^{-2}	0.5	1.00	0.31	0.54	1	0.94	1	0.94	1	0.75
FV10 (Test 1)-(Ecoli)	1	0.75	0.42×10^{-2}	0	0.27	0.36	0.61	0.91	0.80	0.92	0.76	0.92	1
FV11 (Test 2)-(Car)	0.75	0.25	0.02×10^{-2}	1	1.01	0.16	0.29	0.90	0.88	0.85	0.81	0.95	0.75
FV12 (Test 3)-(VertebralColumn)	0.75	0	0.39×10^{-2}	0	0.78	0.82	0.93	0.86	0.89	0.84	0.90	0.86	0.75

Figure 5.2: Problem Selection Attributes

5.3 Incremental and Supervised Learning for all models

Incremental learning refers to the process of learning continuously over time by updating and improving a model as we connect one module to other modules in the Architecture. This type of learning is particularly useful in this situation where multiple components are developed simultaneously and it s important to be able to test some first as failure to learn for one will cause others to not learn and thus as the cost of retraining a model from scratch is expensive and may not lead to desired results. Incremental and step-by-step learning was used to make sure

Accuracy, Autoencoder, Actor and Critic models get fine tuned and work together to generate a policy of our interest.

Supervised learning was initially used to train Encoder-Decoder and Accuracy models using labeled data, where the model tries to learn the relationship between the input features and the output values, so that it can predict the output for new, unseen data. Critic and Actor were also initially pre-trained using supervised learningbut not for the final function as that is not known, but instead for stand-in functions with the goal of start actor-critic training from a state where actor and critic are consistent. Both incremental and supervised learning was applied to various models and by combining these two approaches, we were able to develop models that not only learned from labeled data but also continuously improved their performance as they were integrated with other models, resulting in more accurate and reliable predictions. This is particularly beneficial in this dynamic and complex RL environment where the data is acquired following a learned policy and thus the data distribution is constantly changing.

5.3.1 Encoder-Decoder Pre-Training

Pre-training is a technique used in deep learning to initialize a neural network with a set of weights that have been trained on a large dataset. This can speed up the learning process when the model is fine-tuned on a smaller dataset for a specific task. A pre-training approach was used to train the encoder-decoder architecture for which it is possible to collect a priori data by training a range of randomly selected neural networks on the training problems. Using this, the encoder learns to compress the input data into a lower-dimensional representation, and the decoder learns to reconstruct the original input from the compressed representation.

With the help of supervised learning, the encoder-decoder model was pre-trained on a UCI dataset with labeled data and network accuracy. For the first experiments, the encoder-decoder with simple fully connect networks. Once these experiments were completed successfully, a new experiment extended the network space to more variable sized networks and the encoder was trained to encode time series network data with only limited constrains on length of layers and nodes. In both cases, the encoder mapped the network architectures into a lower-dimensional representation, and the decoder was trained to generate a target sentence from the encoded representation. This pre-trained model was retrained and fine-tuned by combining it with the accuracy network for a more predictable mapping in the 2-Dimensional embedding space.

We found that pre-training the encoder-decoder model using supervised learning resulted in a good initialization of the model weights, which improves its performance when fine-tuned later. Most importantly, however, it allows to verify network sizes prior to training the actor-critic component, and thus to narrow down potential failure sites in case complete training does not achieve the desired performance.

To validate the encoding space, the encoder-decoder network was first trained to obtain the desired accuracy from the decoder network. For this, we first generated an encoder-decoder training set by generating a set of random fully connected networks and training these networks on the chosen UCI classification data sets (Figure 5.1) to obtain their accuracies as target values for the accuracy network during fine-tuning.

During Phase-I training, (50x9) random fully connected legal and (50x9) illegal networks along with their accuracies are generated for the data sets shown in Figure 5.3 as we were only interested in the ability of the model to learn a policy

Number	File Name	No Of Attributes	No Of Classes	Data Points	Training	Testing	ONE HOT VECTOR F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
1	ContainerCraneControllerDataSet.csv	3	10	15	12	3	0	0	0	0	0	0	0	1	0	0
2	AreM_dataset.csv	6	7	42239	33791	8448	0	1	0	0	0	0	0	0	0	0
3	SomervilleHainessSurvey2015.csv	6	2	143	115	28	1	0	0	0	0	0	0	0	0	0
4	ActivityRecognitionHealthyOlderPeole.csv	8	4	75128	60103	15025	0	0	1	0	0	0	0	0	0	0
5	BanknoteAuthentication.csv	4	2	1372	1098	274	0	0	0	1	0	0	0	0	0	0
6	BasketballDataset.csv	7	5	5213	42331	10583	0	0	0	0	1	0	0	0	0	0
7	BloodTransfusion.csv	4	2	748	598	150	0	0	0	0	0	1	0	0	0	0
8	Caesarian.csv	5	2	80	64	16	0	0	0	0	0	0	1	0	0	0
9	CryotheraphyDataset.csv	6	2	90	72	18	0	0	0	0	0	0	0	0	1	0
10	Ecoli.csv	7	8	336	269	67	0	0	0	0	0	0	0	0	0	1

Figure 5.3: Master Dataset File (Problem Description)

for these specific problems, a One-Hot Encoding (label encoding) F1, F2, F3, F4, F5, F6, F7, F8, F9, F10 was used as a target problem feature vector. While this vector will not allow the system to learn a transferable policy, it allows to train a policy that can solve these specific problems. Later this One-Hot Encoding will be replaced during Phase-II training by a set of 13 problem attributes as discussed earlier (Figure 5.2) to evaluate transfer to novel problems. In Phase-III training a more comprehensive set of deeper (up to 5 hidden layers) fully connected legal and illegal networks was generated and represented as time series as shown in Figures 4.3 with the start of the sequence encoded as [0.0,1.0] and the end of the sequence represented by [0.0,-1.0]. An example for a 5 hidden layer network with 23, 20, 12, 11, and 6 hidden units in the layers, respectively, is shown below:

"[0.0, 1.0], [23.0, 0], [20.0, 0], [12.0, 0], [11.0, 0], [6.0, 0], [0.0, -1.0]"

Finally all these randomly generated legal networks for all problems with their accuracies are saved along with illegal networks (accuracy 0) into a training file. Using this training file the encoder-decoder networks are pre-trained using reconstruction loss (MSE) multiplied by legal value to see if the networks were able to form an effective embedding space for the fully connected legal network architectures. Multiplying the loss by the legal status prevents the network from spending effort on trying to reconstruct illegal network presentations, and trading this off against the quality of reconstruction of legal network configurations. The layered diagram for the encoder-decoder network evolved from one for simple fully connected networks (Figure 5.4) to the one for more complex variable length networks using sequence-sequence time series input data (Figure 5.5).

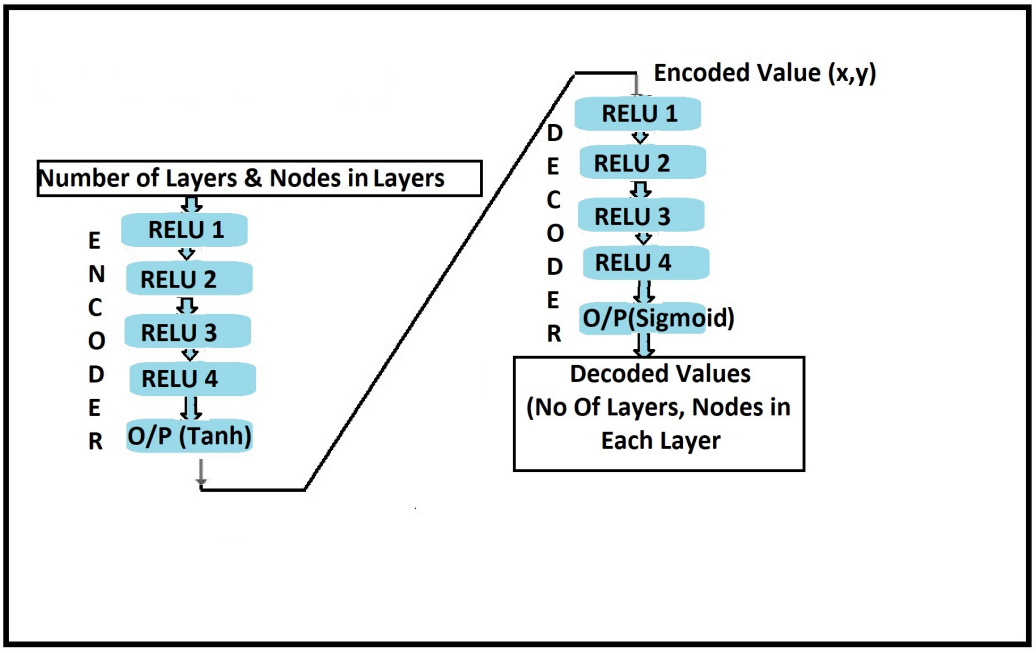


Figure 5.4: Encoder-Decoder-Layer-Diagram

A decoder_loss function was implemented to calculate the difference between the true and predicted values for two sets of variables introduced through the use

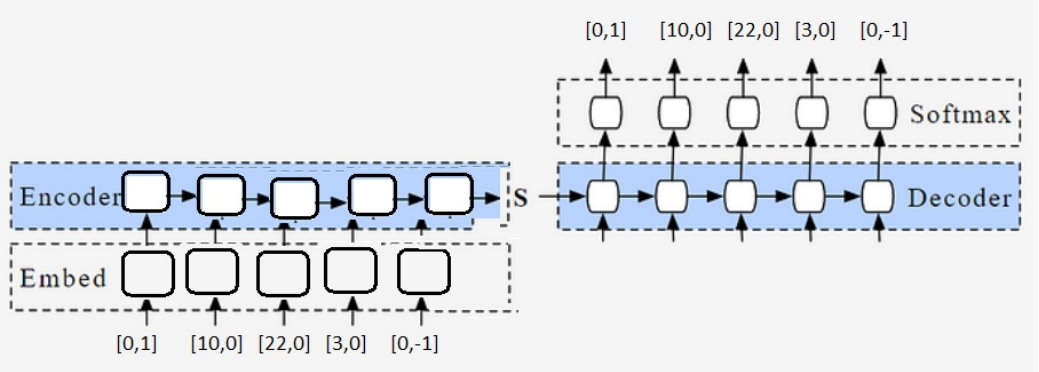


Figure 5.5: Encoder-Decoder-Layer-Diagram

of Siamese networks (and thus the presentation of a pair of network architectures in our encoder-decoder network model.

Below is a step-by-step explanation of our normalization and decoder loss function for the Phase III training:

1. Create a constant matrix_multiply with a shape of (1,3) and values [5, 24, 24, 24, 24, 24] which indicates that we allow max 5 layered network with max nodes 24 in each layer.
2. Multiply the true values y_true[:,2:5] and y_true[:,7:10] by matrix_multiply to obtain denormalized y_true and denormalized1_y_true, respectively. This operation "denormalizes" the true values to their original scale.
3. Multiply the predicted values y_pred[:,2:5] and y_pred[:,7:10] by matrix_multiply to obtain denormalized y_pred and denormalized1 y_pred, respectively. This operation "denormalizes" the predicted values to their original scale.
4. Compute loss1 by taking the element-wise difference between denormalized y_true and denormalized y_pred, squaring the result, and multiplying it by y_true[:,1:2]. Then, calculate the mean of the squared weighted differences along the last axis.

5. Compute loss2 by taking the element-wise difference between denormalized1 y_true and denormalized1 y_pred, squaring the result, and multiplying it by y_true[:,6:7]. Then, calculate the mean of the squared weighted differences along the last axis.

Compute the final loss by adding loss1 and loss2.

The decoder_loss function returns the final loss value, which is used to evaluate the performance of the model. The encoder-decoder model will try to minimize this loss during the training process, improving its predictions on the denormalized variables.

We also calculate the decoder_accuracy_legal matrix for the decoder model to check the performance of the decoder network for all legal decoded networks.

The decoder_accuracy_legal matrix function calculates the average accuracy of a model's reconstruction for two sets of variables. It takes into account only "legal" cases, as indicated by specific columns of the true values.

Below is a step-by-step explanation :

1. Create a constant matrix_multiply with a shape of (1,6) and values [5, 24, 24, 24, 24, 24].
2. Denormalize the true and predicted values for the first set of variables y_true[:,2:5] and y_pred[:,2:5] by multiplying them with matrix_multiply, and round the results to obtain denorm_y_true and denorm_y_pred.
3. Compute the element-wise difference between denorm_y_true and denorm_y_pred, and round the result.
4. Calculate the mean of the differences result along the last axis.
5. Check if the mean differences are equal to 0 , which indicates a correct prediction. Cast the result to float .

6. Multiply the cast result by the mean of y_true[:,1:2] along the last axis to consider only the "legal" cases. Multiply the result by 2 to obtain decoder_accuracy_legal.
7. Repeat steps 2 to 6 for the second set of variables y_true[:,7:10] and y_pred[:,7:10], considering y_true[:,6:7] for the "legal" cases, and obtain decoder_accuracy_legal1.
8. Calculate the returnValue as the average of decoder_accuracy_legal and decoder_accuracy_legal1.

The function returns the returnValue, which represents the average accuracy of the model's predictions for the two sets of variables, considering only the "legal" cases. The higher the returnValue, the better the model's performance.

The training losses, as well as a manual investigation and testing of the embedding space and the accuracy prediction function indicated that the system was able to learn an efficient embedding that was able to reconstruct validation networks with an accuracy above 98.9% and that could predict accuracy with a squared error below 10^{-5}.

5.3.1.1 Siamese Networks

As explained in Chapter 2 that Siamese Networks are a type of neural network architecture specifically designed to learn similarity measures between pairs of inputs. Specially in our case they are particularly useful as there is limited labeled data, as they leverage the information contained within the relationships between the input pairs. We have introduced Siamese Networks in Phase-II to co-locate similar legal networks so that we can learn about a policy while training Actor-Critic networks. Siamese Networks were trained along with encoder-decoder and accuracy networks as explained in the following steps:

1. Prepare training data: generate a dataset of legal/illegal networks in pairs, where each network can be represented as a graph or a set of features. For each pair of legal networks, we also have a binary label (1) indicating whether the pair is similar or not.
2. Siamese Network architecture: as explained in Figure 2.2 -Chapter 2, it consists of two identical subnetworks, often called "sister networks," that share the same weights. Each sister network processes one of the elements n the input pairs, and their outputs are combined to produce a similarity score. During design of the Siamese Networks architecture for our sister networks we included encoder, decoder and accuracy networks while considering the nature of our legal network data.
3. Contrastive(distance) loss function: encourages the model to learn representations that bring similar pairs closer together and push dissimilar pairs further apart. This model is trained using our paired training dataset which includes legal_networks_1 and legal_networks_2 as ~~an~~ arrays containing the feature representations of our legal networks, where the ith element in both arrays forms a pair.
   Note, the labels array should contain the binary labels (legal(1), illegal(0)) corresponding to each pair.
4. After successful training: Siamese Networks can compare new pairs of legal networks and determine their similarity. The model will co-locate similar legal networks by assigning them similar feature representations in the learned embedding space. To do this, we used the sister_network submodel to obtain the embeddings for any new legal networks and calculate the

distance between the embeddings.

By comparing the distances for different pairs of legal networks, we can identify which legal networks are closer together in the learned embedding space. This enabled us to group similar legal networks and analyze their relationships more effectively as shown in Figure. 5.6

One can also use other distance metrics, such as the cosine similarity, depending on the characteristics of the legal network data and the desired properties of the similarity measure. Furthermore, one may consider using more advanced graph neural network architectures for the sister networks if the legal networks are represented as graphs.

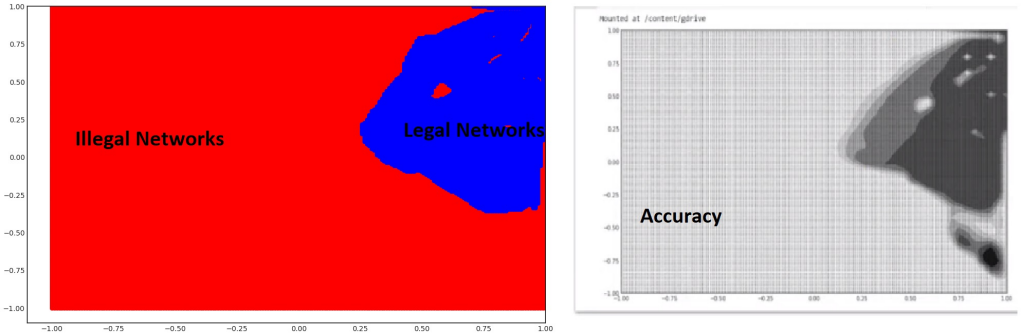


Figure 5.6: Siamese and Accuracy Diagram to show legal and illegal networks

5.3.2 Pre-Train Accuracy Network

To have more effective Accuracy training we add Gaussian Noise and transform all 13 Feature Vectors to a dense set of length 6 Feature Vector effectively learning an embedding of the target problem feature vector. This is done here as we know that deep learning neural networks are likely to quickly overfit a training dataset with few examples, and thus lose the ability too generalize to new

problems. Therefore we add a dropout layer as dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. The Feature_Conversion_model is designed to transform Feature Vectors from size 13 to 6 before we introduce them to the Accuracy Network. The feature vector embedding architecture is shown in Figure 6.5.

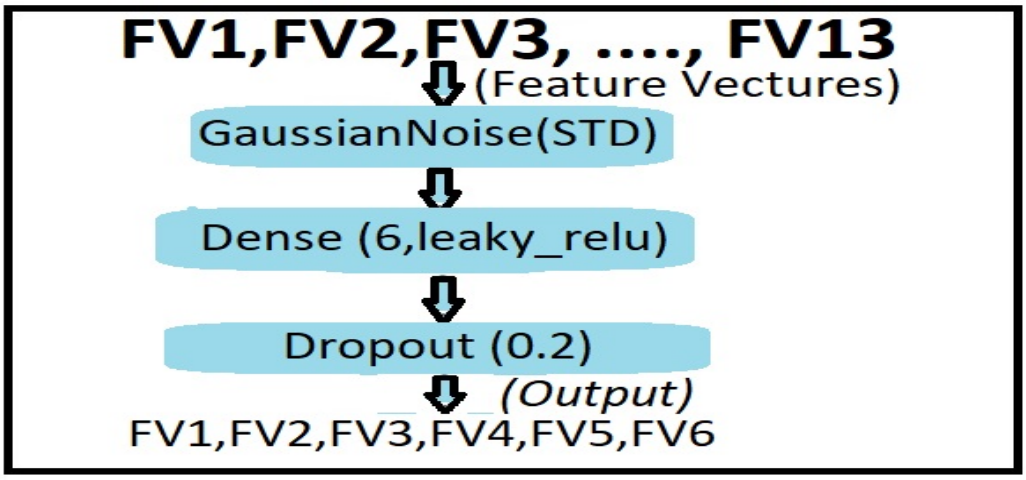


Figure 5.7: Feature Conversion Model Layer Diagram

Next we freeze the weights of encoder-decoder model and train them together with the accuracy network. A custom loss function for the accuracy network was designed which uses binary_cross_entropy to calculate legal prediction loss and MSE to computer accuracy loss, reaching a MSE of 0.00257641.

5.3.3 Training Encoder Decoder and Accuracy Network all together

Once we have the encoder-decoder and accuracy network pre-trained individually, we now train them together for 5000 episodes. We must clip the optimizer for good results as: *optimizer = Adam(clipnorm=1.0)* . Before we got to the final

training we came up with a custom_loss function for this joint model which is shown in Figure 5.8.

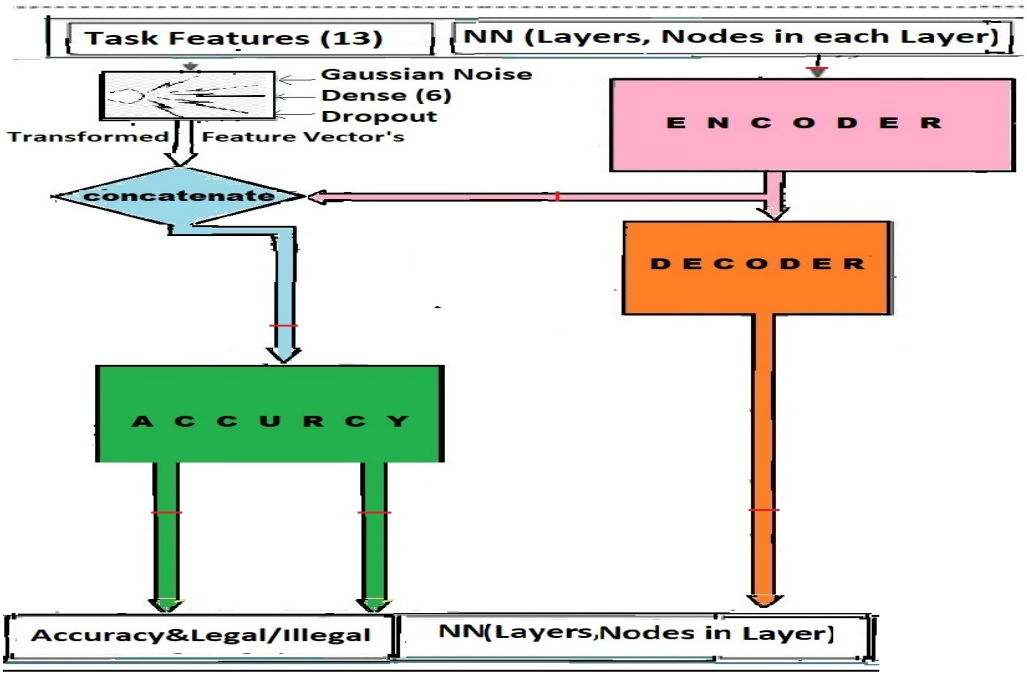


Figure 5.8: Encoder-Decoder-Accuracy-Diagram

5.3.3.1 Custom Loss Function

The Custom loss function utilized is consists of decoder_loss, accuracy_loss, legal_loss and distance_loss. By trial and error we came up with their weighting to add up for Custom loss function as shown below:

$weight_A = accuracy\_loss\_weight(ratio) = 1.0$

$weight_L = legal\_loss\_weight(ratio) = 1.2$

$weight_D = decoder\_loss\_weight(ratio) = 1.08$

$weight_{Dist} = distance\_loss\_weight(ratio) = 0.325$

This custom loss function combines four different loss components to evaluate the performance of a neural network model.

The loss components are:

1. **Decoder loss (decoder_loss):** This measures the squared difference between the true and predicted values for two sets of variables (denorm_y_true, denorm1_y_true) after multiplying them by a certain weight (y_true[:,1:2], y_true[:,6:7]). The differences are averaged, and the decoder_loss is obtained by summing up the two averaged differences.
2. **Accuracy loss (acc_loss):** This calculates the squared difference between the true and predicted values for two different variables (y_true[:,0:1], y_true[:,5:6]).
3. **Legal loss (legal_loss):** This computes the binary cross-entropy between the true and predicted values for two different binary variables (y_true[:,1:2], y_true[:,6:7]). Binary cross-entropy is a common loss function used for binary classification problems.
4. **Distance loss (distance_loss):** This calculates a weighted combination of the true values of two variables (y_true[:,1:2], y_true[:,6:7]) and a predicted variable (y_pred[:,10:11]). The weights are derived from the true values, and their combination is multiplied by the predicted variable.

The custom_loss function combines these four loss components using different weights ($weight_A$, $weight_L$, $weight_D$, and $weight_{Dist}$) to calculate the final loss value. During the training process, the model tries to minimize this custom loss, improving its predictions across various aspects.

$$\textbf{\textit{custom\_loss}} = weight_D * \textbf{(decoder\_loss)} + weight_A * \textbf{(accuracy\_loss)} + weight_L * \textbf{legal\_loss} + weight_{Dist} * \textbf{distance\_loss}$$

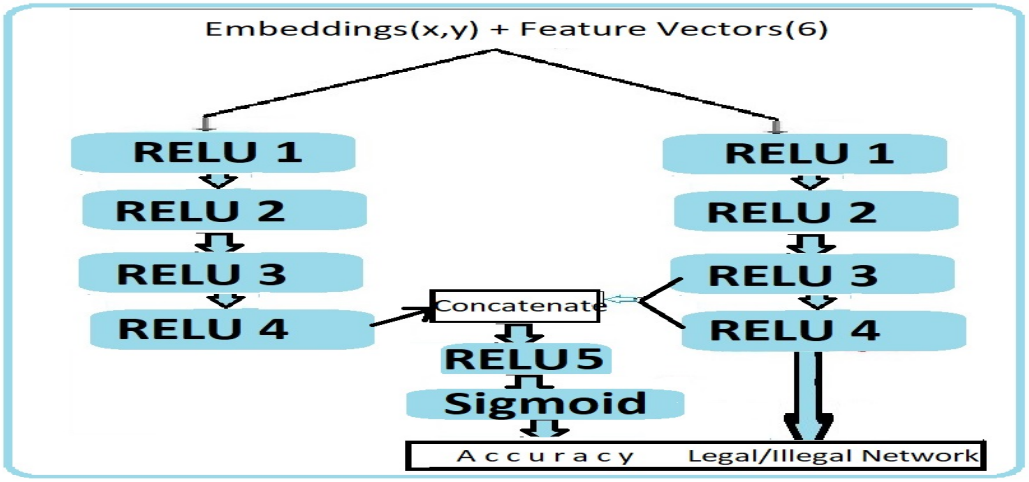


Figure 5.9: Encoder-Decoder-Accuracy-Diagram

The distance loss was introduced by the Siamese network to help embed all legal networks in proximity so that an efficient trajectory exists between legal networks and thus optimization strategies do not have to cross large areas of illegal networks in the embedding space.

To evaluate performance we calculated 9 metrics to make sure we get a well trained encoder-decoder-accuracy model to help us obtain an effective policy when we perform Reinforcement training with Actor and Critic Models. The layered diagram of the Accuracy Network is shown in Figure 5.9.

After training this combined encoder-decoder-accuracy model for 5000 episodes we obtained following metrics values to evaluate our training and gage model performance :

1. training_loss(0.0208641),
2. validation_loss(0.13546667),
3. decoder_accuracy_legal (0.9905621409416199),
4. validation_decoder_accuracy_legal(1.0110701322555542),
5. decoder_loss(0.007852517999708652),

6. validation_decoder_loss(0.0025045871734),

7. custom_loss(0.0209068),

8. validation_custom_loss(0.10064433515071869),

9. mean_sqe_pred(5.9493566368473694e-05),

10. validation_mean_sqe_pred(4.512822124524973e-05),

11. binaryCrossEntrphy(0.010269965045154095),

12. validation_binaryCrossEntrphy(0.11059717833995819),

13. legal_Network_Pred(0.9971275925636292),

14. validation_legal_Network_Pred (0.9778597950935364)

5.3.4 Training Data For Actor and Critic models

Once we have a high performing autoencoder and accuracy model we need to train Actor and Critic Networks to evaluate whether Reinforcement learning can actually produce a policy that will optimize the network architecture for the UCI dataset used.

We generate training data for pre-training Actor and Critic networks by utilizing a given dataset. This input data set consist of embeddings for both legal and illegal networks along with values for all 13 feature vectors as shown in Figure 5.10. Note, with the help of Siamese Networks we have co-located all legal and illegal networks and tanh Activation sets the boundary of my embeddings in two Dimensional space (grid), which is X (+1,-1) and Y (+1,-1).

We also define the list of action directions starting from 0 to 360 degrees at intervals of 10 degrees apart. We evenly explore the embedding space by covering the distance from 0 to 1.0 in all 360 directions to generate the training data as shown as Figure 5.11:

listAngles = [0, 10,20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180,190, 200, 210, 220, 230, 240, 250, 260, 270, 280, 290,300,310, 320,330, 340, 350, 360]

listDistance = [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

Problems (From UCI DATASET)	FEATURE VECTORS (Normalized)													Network Accuracy & Legal /Illegal		
Features ⟶ Embeddings(x,y) ⟶	No Of Attributes	No Of Classes	Data Set Size	Attribute Type	Entropy Label	Avg Entropy Features	Avg Correlation Between Features	2_6_8 Training Accuracy	2_6_8 Test Accuracy	1_5_0 Training Accuracy	1_5_0 Test Accuracy	2_6_20 Training Accuracy	2_6_20 Test Accuracy	Network Accuracy	Legal
0.16878778,0.7779948	0.75	0.62	0.56	0	0.93	0.43	0.50	0.72	0.72	0.69	0.69	0.73	0.75	0.74	1
0.80934095,0.9981187	0.75	0	0.17×10^{-2}	1	0.99	0	0.73	0.85	0.45	0.72	0.65	0.96	0.75	$6.41e^{-37}$	0
0.5502441,0.4602287	1	0.25	1	0	0	0.16	0.21	0.97	0.97	0.97	0.96	0.97	1	0.96	1
0.7842969,0.9980797	0.25	0	0.02	0	0.98	0.44	0	1	1	0.99	0.98	1	0.25	$1.24e^{-26}$	0
0.16346914,0.3396268	1	0.37	0.70	0	0.91	0.58	0.28	0.6	0.69	0.56	0.57	0.75	1	0.75	1
0.8312806,0.9980330	0.25	0	0.97×10^{-2}	0	0.43	1	1	0.82	0.80	0.79	0.78	0.79	0.25	0.79	1
0.932896,0.99238926	0.5	0	0.08×10^{-2}	1	0.72	0.15	0.48	0.89	0.75	0.79	0.44	0.91	0.5	$8.39e^{-29}$	0
0.8954702,0.92344075	0	1	0	0	0.93	0.34	0.29	0.90	0	0.90	0	0.28	0	$1.58e^{-27}$	0
0.6008147,-0.252559	0.75	0	0.09×10^{-2}	0.5	1.00	0.31	0.54	1	0.94	1	0.94	1	0.75	0.98	1
0.32733244,0.404954	1	0.75	0.42×10^{-2}	0	0.27	0.36	0.61	0.91	0.80	0.92	0.76	0.92	1	0.91	1
0.017314577,0.459166	0.75	0.25	0.02×10^{-2}	1	1.01	0.16	0.29	0.90	0.88	0.85	0.81	0.95	0.75	$1.62e^{-29}$	0
0.88320404,0.995748	0.75	0	0.39×10^{-2}	0	0.78	0.82	0.93	0.86	0.89	0.84	0.90	0.86	0.75	0.96	1
...,...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 5.10: Input data with embeddings and Feature Vector's

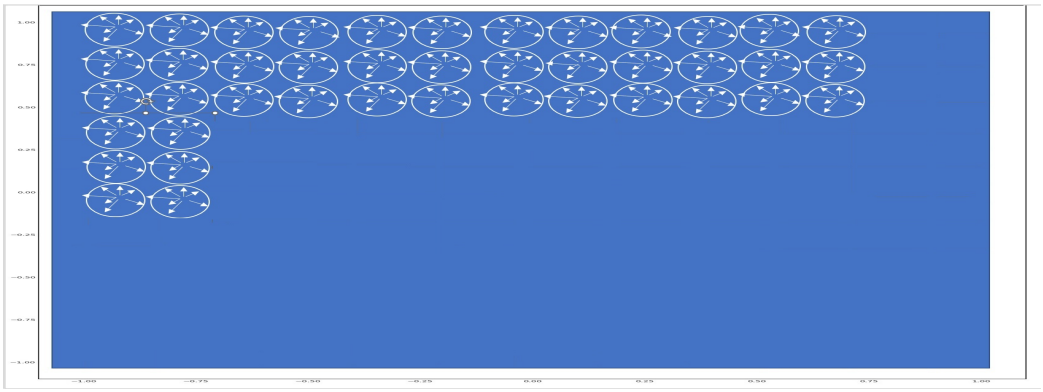


Figure 5.11: Exploring reward and accuracy in 360 Degree at various length

After we read the input data (Figure 5.10) we generate 10 additional vectors with noise for each row to avoid overfitting. Then we compute the next state, accuracy, legality and reward for each action in the action space using the getNextSteps() function. Finally, we calculate total reward, average reward (Reward_V), and Advantage Reward (Reward_A) which is nothing but subtracting average reward from reward at each step in the neighborhood as shown in the Figure 5.11This results in data items as shown in Figure 5.12.

Training Data To Pre-Train Critic Network

Current State Embeddings (x,y)	Next State Embeddings (x,y)	Action Array	Reward	Current Accuracy	Next Accuracy	Current Legal	Next Legal	Feature Vector's (FV1,....FV13)	Reward_V	Reward_A
0.1687,0.777	0.1697,0.788	[0.98,0.17, 0.1]	-0.038	0.64	0.60	1	1	.....	0.003	-0.0483
...,...	...,...	...	...	...	...	...	...	...	...	...

Figure 5.12: Training Data Set To Pre-Train Critic

The getNextSteps() function is implemented to help compute the next state, accuracy, legality, and reward for each action in the given action space using an input data set and an accuracy_model. The function takes a input dataset containing information about the current state, and uses accuracy_model to predict accuracy and legality for the next states. Further, with the help of change in accuracy it computes the reward at the next state based on the current and next state legality and accuracy.

To avoid the model to be too shallow, we scale reward by 2.5.

In case our next state is illegal while the current state is legal we punish by assigning -1 to the reward. Similarly, if our next state is illegal while the current state is also illegal we assigning 0 to the reward. Such as: illegal_state to illegal_state:

Reward = 0. The goal here is to prevent the policy to move towards illegal networks as we have not trained accuracy predictions for those areas, which could therefore be misleading.

Once we have generated training data we will pre-train our Critic Network so that it adapts for future training with the Actor Network.

5.3.5 Pre-train Critic models

To effectively train actor and critic using RL we first attempt to create consistent actor and critic networks through a sequence of pre-training steps. We first pre-train our critic network for a uniform policy and a discount factor of $\gamma = 0$, i.e. to reflect the reward function.

This basically initializes the critic network with the average reward values and serves the purpose to assess whether the network is sufficiently complex to represent this function over the embedding space. During training we use the mean squared error (MSE) loss between the true and predicted values. We run two training sessions for critic pre-training as we will later need two distinct critic networks in the TD3 approach.

Once our critic network is trained we predict critic values for the given embeddings and found that the plot of critic_A against embeddings is very similar to the embeddings plotted against Rewards. This proves that critic was pre-trained successfully.

5.3.6 Pre-train Actor model with Critic Weights Frozen

Through a series of experiments and iterative refinements, it has been determined that pre-training the actor network in conjunction with the critic network improves their mutual compatibility. This step-by-step process currently involves

pre-training the actor network with the pre-trained critic while maintaining the fixed weights of the previously trained critic. The discount factor γ is kept at 0, and τ(tau) is set to 0.00005. τ represents the discount factor, with a high τ rendering all actions equiprobable, while a low τ biases the probability towards a greedy policy. Over the course of training, τ can be annealed, facilitating greater exploration at the beginning of training and increased exploitation towards the end.

This evaluation serves to determine if the actor network's complexity is adequate to capture the intricacies of a policy within the embedding space. It is important to note that we blend the weights for the critic network prior to initiating actor training. This entails obtaining critic main1 and critic target1, as well as critic main2 and critic target2 weights from two independently trained critics, as depicted in Figure 5.13.

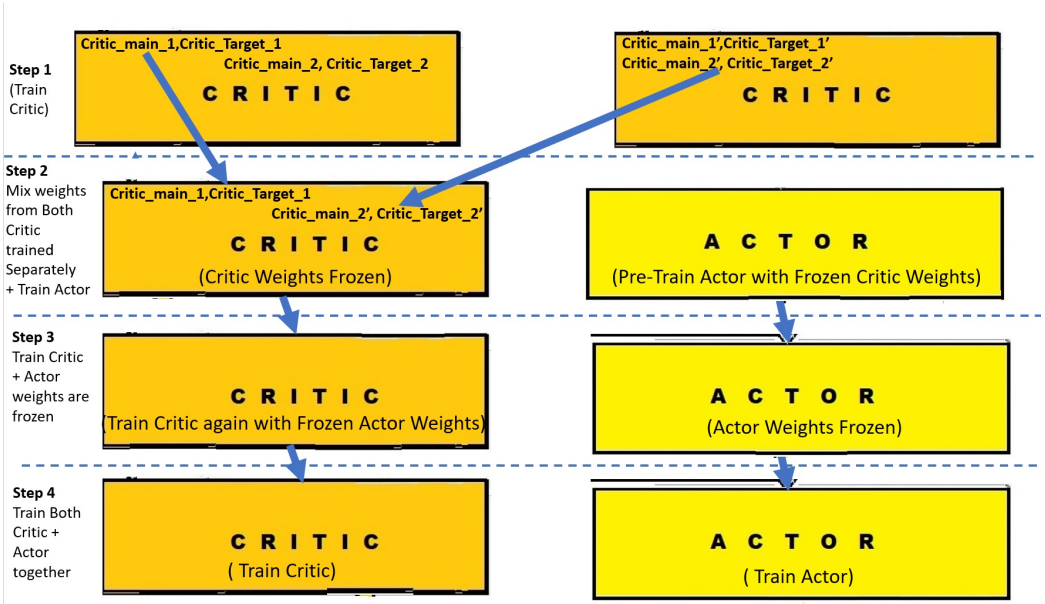


Figure 5.13: Actor-Critic-Training Flow

5.3.7 Pre-train Critic model with Actor Weights Frozen

After training the actor with the pre-trained critic, we proceed to train the critic network again while keeping the actor's weights fixed. We utilize the pre-trained policy (Actor) to generate embedding trajectories, employing a final discount factor of $\gamma = 0.9$ and τ (tau) = 0.00005. This ensures that our pre-trained critic is consistent with the policy before commencing the comprehensive Actor-Critic (TD3 RL) training for both the critic and the policy. Figure 5.13 illustrates the step-by-step Actor-Critic training flow and its dependencies.

5.3.8 Training Actor and Critic models together

Following the completion of the pre-training process for network models using active learning and a pre-trained acquisition function to optimize performance, both the actor and critic are trained concurrently for several iterations. This joint training allows them to synchronize and fine-tune their exploration and training capabilities for the final policy. The trajectory data is generated on-the-fly based on the current policy at the respective training stages, as produced by the actor network.

It is crucial to note that a well-trained, high-performing actor network enables the construction of a policy that can effectively identify an optimal network for the original classification problem.

5.4 Explore and carry out action with exploration noise

5.4.1 Background

For this training, we employ the TD3 algorithm [71] along with a large replay buffer to store the states, actions, rewards, and new states generated by executing

trajectories in accordance with the actor's current policy. In our continuous embedding action space, we explore and execute actions with exploration noise for target policy smoothing. To regularize action noise, a small amount of random noise is added to the target, averaging over multiple mini-batches. As a result, policies tend to be more stable since the target value returns higher values for actions that are more robust to noise and interference.

When calculating the targets, clipped noise is added to the selected action to obtain higher values for more robust actions. These transitions are saved and used to update the critic, followed by the actor and target networks. We continue this process until stable results are achieved.

5.4.2 Explore

Exploration in discrete action spaces is typically achieved through probabilistic selection of random actions, such as Boltzmann exploration. However, in continuous embedding action spaces, exploration is performed by adding noise directly to the action itself. Since the critic-pair network has twice the number of layers as the actor network, these critic-pair layers operate in parallel, resulting in two separate outputs, as illustrated in Figure 4.10.

The learning function (train) drives exploration, encompassing the majority of the functionality. It is essential to ensure that the memory is filled to at least the batch size, as learning should not occur for less than the batch size. Exploration commences by allowing the agent to select an action with added exploration noise, a standard step in the Markov Decision Process (MDP) for the embedding environment.

Gradient tape is utilized for gradient calculations, as it facilitates the addition of operations to the computational graph for gradient computations. Consequently,

when the action act() function is called on the agent network, those operations used for gradient calculations are not stored anywhere and are effectively detached from the graph. This is where the update rule is applied, as it is the only element within this context manager used for gradient calculation.

To obtain optimized results, the actor-critic network was trained for 5,000 episodes with a maximum of 300 steps per episode. For each step, the step size is calculated by adding alpha (0.005) to the 2D embedding space as follows:

$$next\_state = current\_state + alpha * action \tag{5.1}$$

The next_accuracy value is calculate by passing this next_state value to the Accuracy Model. The reward is then computed as the difference between the accuracy of the next state and the current state, ultimately determining the utility value of this one-step exploration.

Crucial exploration results, including state, next_state, action, reward, and next_accuracy, are stored in the first replay buffer (RBuffer) for future training.

Scatter plot of trajectories (Fig. 5.14) revealed that certain parts of the trajectories did not connect to their intended goals. This also suggests that during the exploration of the embedding space, the training data points did not traverse those areas. One possible explanation (based on observation) is that during Phase I of the initial training, we took a fixed number of steps in the embedding space, resulting in our trajectories not fully exploring all embedding points towards the goal. This implies that the final reward is not being adequately propagated. The challenge faced here is the finite horizon; we do not want to run our trajectories indefinitely, and reinforcement learning may not be highly effective if we cannot access the full

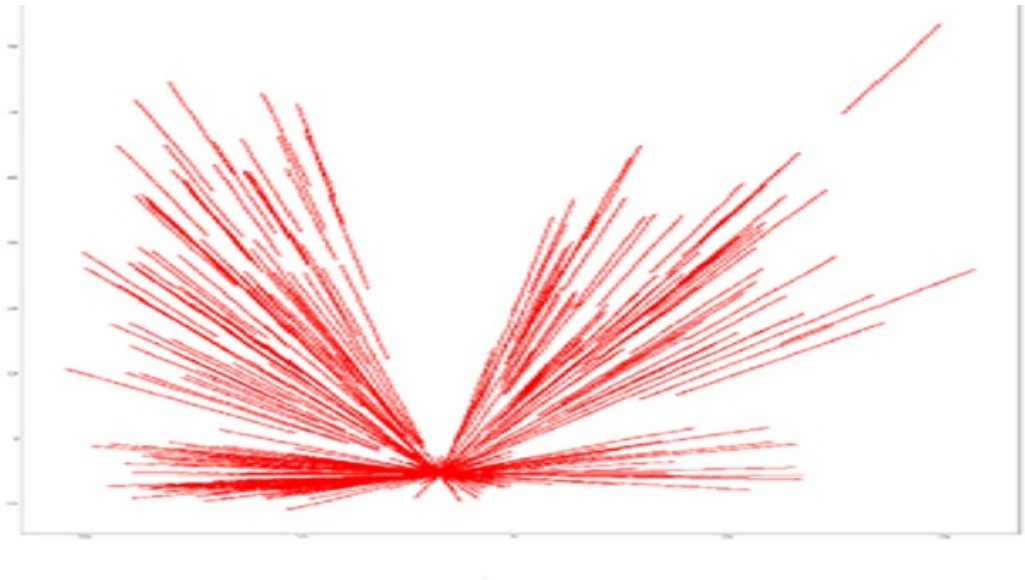


Figure 5.14: Scatter Plot After Initial Exploration of Embedding Space

reward and properly propagate the value function. However, the Forward Lookup Tree Search did assist in eliminating gaps while exploring the embedding space.

5.4.2.1 Forward Lookup Tree Search

To minimize fluctuations in results while exploring the embedding space, additional exploration was necessary. A forward lookup strategy was employed by constructing a tree with a depth of 3 and 5. The algorithm's flow for the forward lookup tree is illustrated in Figure 5.15:

The high level flow for forward lookup tree search is shown as:

1. Traverse the tree depth-first, and at each node along the way, compute the reward, next_accuracy, and next_State using the Accuracy Model, similar to a one-step exploration.
2. Calculate the utility value of current node.
3. Select the node with max utility among all leaf nodes.

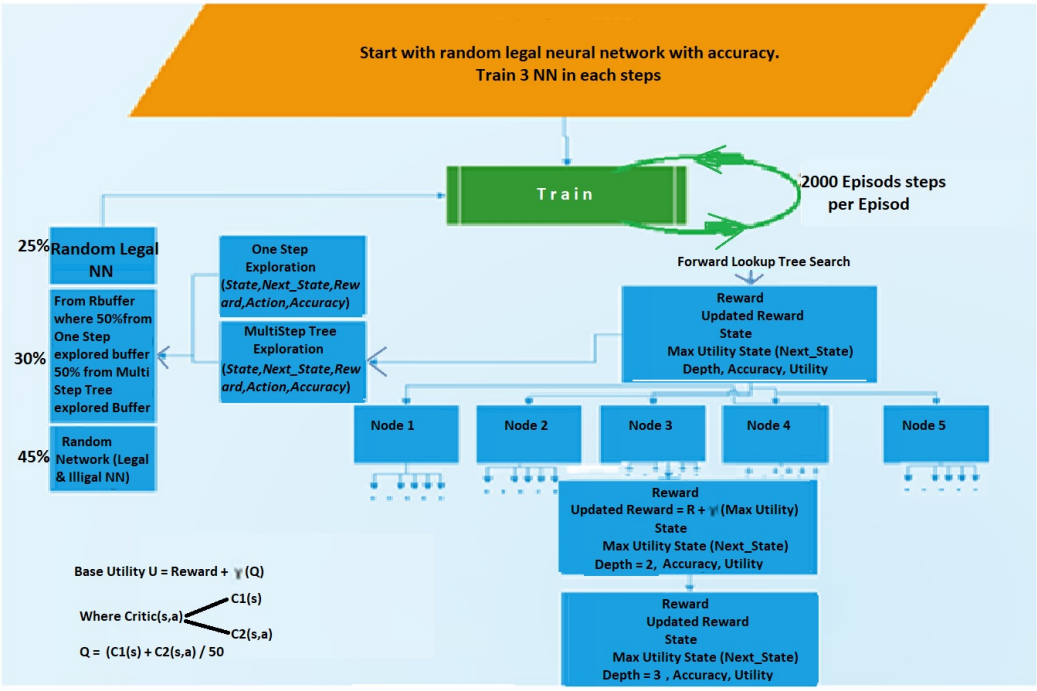


Figure 5.15: Forward Lookup Tree Search

4. Repeat step 1,2,3 by traversing this tree of depth 3 with 5 leaves at each node

Exploring the node with the highest utility is also known as 'multistep forward tree exploration'. Key results obtained here, such as state, next_state, action, reward, and next_accuracy, are stored in the second replay buffer (RBuffer) for future training. It should be noted that networks from both replay buffers (one-step and multistep) are utilized equally (50% each) for future training steps to update the networks.

5.4.3 Update Critic

This step is a critical part of the algorithm, encompassing most of the additional features of TD3. The first task is to sample a mini-batch of stored transitions

from the replay buffer, i.e., random samples with a batch size of 300. The key stored transactions include states, next_states,rewards,actions,accuracy & depth.

Next, select an action for each state retrieved from the mini-batch and apply target policy smoothing. this involves choosing an action for the target actor network. Noise is added to this action and clipped to ensure that the noisy action does not deviate significantly from the original action value.

Once the target_actions are computed, the target values for the critic are determined. This is where the double critic networks (critic-pair) come into play. V and A values for each target critic are calculated, and the smallest of the two (critic_target and critic_target2) is chosen for the target value.

The critic value for the new states is computed through the target critic evaluation of the next states and target actions, squeezed along the first dimension. This process performs a forward pass to obtain the value of the successor state for the best action.

Note that the predicted (critic) values are the output of the main critic network, which takes states and actions from the buffer sample. The critic value represents the value of the action in a given state, with respect to the original state and actions the agent actually took during the course of the episode.

The target value for the terminal new state is simply the reward for every other state. According to the Bellman equation, the target critic network's value is the reward plus the discounted value of the resulting state. If there is no successor state, the target value is the reward itself.

In addition, the target value for critic loss is calculated by predicting the action for the next states using the actor's target network. Then, using these actions, one can obtain the next state's values using the critic's target network. These derived target actions are essentially the target actor's indication of what steps one should

take for the new states. To avoid looping issues, the actor target is used instead of the actor main.

Finally, the loss for the two current critic networks is calculated. The loss function is computed using three networks: actor target, critic target, and critic main. Critic loss is determined as the mean squared error (MSE) of target values and predicted values. The Boltzmann error is the difference (MSE) between the value of the action in the current state and the sum of the reward and the gamma times the maximum of all possible action values in the successor state. This is essentially the MSE of each current critic and the target values calculated earlier.

If the rewards are very small, the values are boosted by a boosting factor (2.5), which was used earlier to amplify the rewards.

Thus, the critic's optimization is carried out by updating the critic and minimizing the loss. This is done by applying the gradients to the same critic main trainable variables, i.e., all the weights of the three layers in the main critic network. The gradient tape is utilized to load up operations to the computational graph for gradient calculations.

It is important to note that the gradient involves all the weights of all three networks, but weights are applied only to the main critic networks. Gradients are applied to the same critic main trainable variables, using all the weights of the three layers in the main critic network.

5.4.4 Update Actor

Compared to the critic, updating the actor is much simpler. It is essential to ensure that the actor is updated less frequently than the critic, which means updating the actor every second time step. The actions from the actor are selected based on its current set of weights, not based on the weights it had at the time the

data was stored in the agent's memory (RBuffer).

$$new\_policy\_actions = actor\_main(states)$$

The actor's loss function calculates the mean of the negative Q values from the critic network, with the actor determining which action to take given the mini-batch of states. The actor loss is represented as negative ('-') because gradient ascent is being used here. As explained in policy gradient methods, gradient descent is not used because it would minimize the total score over time. To maximize the total score over time, gradient ascent is required, which is simply the negative of gradient descent.

The actor network is optimized using backpropagation, as the actor loss involves both the actor main and critic main networks. Actor loss is calculated as the negative of critic main values, with inputs being the main actor's predicted actions.

Thus, the actor is updated using a combination of the deterministic policy gradient and delayed updates. The actor is updated as follows:

Delayed policy updates: The actor is updated less frequently than the critic networks, typically after every second update of critic updates. This delay helps to reduce the overestimation bias and stabilize the training process.

Target policy smoothing: Which adds noise to the target actions before passing them to the target critic networks. This technique encourages the actor to explore a wider range of actions and helps to mitigate the overestimation bias.

Deterministic policy gradient: The actor's loss function is calculated using the deterministic policy gradient, which is the mean of the negative advantage values from one of the critic networks. The actor chooses the action to take, given the mini-batch of states, and the gradient ascent is used to maximize the expected return.

Backpropagation: The actor network is optimized using backpropagation, as the actor loss involves both the actor main and one of the critic main networks. Actor loss is calculated as the negative of the critic main values with inputs being the main actor's predicted actions.

Soft target updates: After updating the actor network, the target actor network is updated using a soft update strategy. This strategy involves slowly blending the weights of the main actor network with the weights of the target actor network, ensuring a smoother and more stable learning process.

With the help of TD3 algorithm the actor's update process is tremendously improved by using two critic networks, delayed policy updates, target policy smoothing, and soft target updates. These enhancements help to reduce over-estimation bias and improve the stability and performance of the reinforcement learning agent.

5.4.5 Update Target Networks

Target networks are time-delayed copies of their original networks, and they slowly track the original networks. To avoid looping issues, the actor_target is used and actor_main is kept stable and a copy of the initial weights of both actor_main, critic_main network is done is transferred to actor_target & critic_target network.

In the end, frozen target networks are updated using a soft update alongside the actor update. Actor_target and critic_target networks are slowly moved using tau(τ) towards the trained actor_main and critic_main network via "soft updates".

Updating the target networks is a crucial aspect of the learning process. The target networks are essential in reducing the overestimation bias and stabilizing the training. We maintain two sets of networks, a main network (actor and two critics) and a target network (target actor and two target critics). The target networks

are updated using a soft update strategy, which involves a slow blending of the weights of the main networks with those of the target networks.

The soft update strategy is implemented as follows:

Polyak (τ) averaging: A hyperparameter called the Polyak factor (denoted as 'τ') is introduced, which is typically set to a small value (e.g., 0.005). The τ factor determines the rate at which the target networks are updated.

Updating the target critic networks: The weights of both target critic networks are updated by blending the weights of the main critic networks with the corresponding target critic networks.

The update is performed using the following equation:

$$target\_critic\_weights = (1 - \tau) * target\_critic\_weights + \tau * main\_critic\_weights$$

This equation is applied to both target critic networks.

Updating the target actor network: Similarly, the target actor network is updated by blending the weights of the main actor network with the target actor network using the same τ factor:

$$target\_actor\_weights = (1 - \tau) * target\_actor\_weights + \tau * main\_actor\_weights$$

By using the soft update strategy, the target networks are gradually updated to reflect the main networks' learning without drastically changing their outputs. This approach helps to maintain stability in the learning process and reduce the overestimation bias, leading to improved performance in TD3-based reinforcement learning.

The RL steps are repeated until the policy converged.

CHAPTER 6

Results and Performance

6.1 Introduction

In this chapter, we will discuss the results and performance of my experiments as discussed in Chapter 5.

6.2 Identifying mapping for Embedding Space

We begin by considering how to represent our networks and approximate connectivity in the embedding space. Our criteria for useful embedding space is that every state in an embedding space region is reachable from every other state by means of a single control RL generated policy. Much like NOA architecture by Luo[1], the initial training here was focused on fully connected networks up to a certain size, along with some performance data (network accuracy) and network status (legal/illegal) to help construct a simpler modification space. This resulted in defining the network space as fully connected networks of limited size as well as classification problems (Figure 6.1) from the UCI data set [74]. This permits the input network specification to be supplied in terms of layers and unit numbers per layer and provides an easy means of generating initial training data by training 300 random networks [60], each on, one of the 12 target problems.

6.3 Random Network generation and Training for Embeddings

For effective training of all models, quality, accurate, complete, and relevant data is needed to start early on in the training process. Dataset selection was pri-

Problem Dataset	No Of Attributes	No Of Classes	Data Set Size	Attribute Type	Entropy Label	Avg Entropy Features	Avg Correlation Bet Features	2_6_8 Training Accuracy	2_6_8 Test Accuracy	1_5_0 Training Accuracy	1_5_0 Test Accuracy	2_6_20 Training Accuracy	2_6_20 Test Accuracy
AreM	0.75	0.63	0.562	0	0.93	0.43	0.49	0.72	0.72	0.68	0.69	0.74	0.72
Somerville Happiness Survey	0.75	0	0.17x1	1	0.99	0	0.73	0.85	0.45	0.72	0.65	0.96	0.62
Activity Recognition Healthy_older People	1	0.25	1	0	0	0.16	0.21	0.97	0.97	0.96	0.96	0.97	0.97
Banknote Authentication	0.25	0	0.28x1	0	0.98	0.44	0	1	1	0.99	0.98	1	1
Basketball	1	0.38	0.70	0	0.92	0.58	0.28	0.69	0.69	0.56	0.57	0.75	0.75
BloodTransfusion	0.25	0	0.98x1	0	0.44	1	1	0.81	0.8	0.79	0.78	0.79	0.78
caesarian	0.5	0	0.08x1	1	0.72	0.15	0.48	0.89	0.75	0.79	0.44	0.91	0.62
Container Crane Controller	0	1	0	0	0.94	0.34	0.29	0.9	0	0.9	0	0.28	0
Cryotherapy	0.75	0	0.99x1	0.5	1.01	0.31	0.54	1	0.94	1	0.94	1	0.78
Ecoli	1	0.75	0.43x1	0	0.27	0.36	0.61	0.91	0.80	0.91	0.76	0.92	0.84
Car	0.75	0.25	0.23x1	1	1.01	0.16	0.28	0.90	0.88	0.85	0.81	0.94	0.92
VertebralColumn	0.75	0	0.39x1	0	0.78	0.82	0.93	0.86	0.88	0.84	0.90	0.85	0.87

Figure 6.1: Initial Training Data Set With Problem Selection Attributes(Normalized)

marily focused on dataset and attribute characteristics. Also to uniquely represent each problem in an embedding space, a set of 13 problem attributes were also selected and computed to represent each problem in the training dataset. Apart from 'No of Classes', 'Data Set Size', 'Attribute Type', we also computed Entropy on dataset labels, average Entropy and average Entropy between data set features. To give a good spread and variation, training and test accuracy were computed on three Neural Networks ([2,6,8], [1,5,0] and [2,6,20]) with different number of layers and nodes in each layers (Figure 6.1).

Classification problems from the UCI data set [7] were used with a default reward function as an improvement in performance (network accuracy) due to the change in the network in embedding space. Each dataset represents separate entities or problems and will help us to run trajectories to optimize one problem at a time. For our training we have used a random initial policy to build trajectories in embedding space. To study the performance and operation of the different

network components, we chose here to go through a sequence of pre-training steps which focus on different components and then studied the results before training the next component. Later to allow more complex networks we used time series network data without any constrains on the length of layers and nodes so that we can map these networks into a lower-dimensional representation. Legal and illegal networks represented by time series data set are shown in Figures 4.3 with start of sequence [0.0,1.0] and end of sequence [0.0,-1.0].

For example: "[0.0, 1.0], [23.0, 0], [20.0, 0], [12.0, 0], [11.0, 0], [6.0, 0], [0.0, -1.0]" shows a network with 5 layers and each layer heaving [23,20,12,11,6] nodes.

All these synthetically generated legal networks for all problems with their accuracies are saved along with illegal networks (accuracy 0) into a training file. Using this training file the encoder-decoder networks are pre-trained using reconstruction loss (MSE) multiplied by legal value and we found that networks were able to form an effective embedding space for our legal network architectures. The layered diagram for encoder-decoder network evolved for simple fully connected network (Figure 3.4) to more complex variable length networks using sequence-sequence time series input data (Figure 4.5).

Since the network architecture input formulation allows for illegal inputs, an addition 300 illegal network descriptions were generated and assigned accuracy values of 0 to allow the system to learn the demarcations of the embedding space. For this initial proof of concept, target problems are trained independently, yielding problem-specific policies.

Siamese Network architecture as explained in Figure 2.2 (Chapter 2), is trained consist of two identical subnetworks of our time series data, often called "sister networks," that share the same weights. Each sister network processes one of the input pairs, and their outputs are combined to produce a similarity score. We also trained

encoder, decoder and accuracy networks with siamese network, while considering the nature of our legal network data. The distance loss function encourages the model to learn the representations that bring similar pairs closer together and push dissimilar pairs further apart. This model is trained using our paired training dataset (Figure 6.2) which includes legal_networks_1 and legal_networks_2 as an arrays containing the feature representations of our legal networks, where the i^th element in both arrays forms a pair.

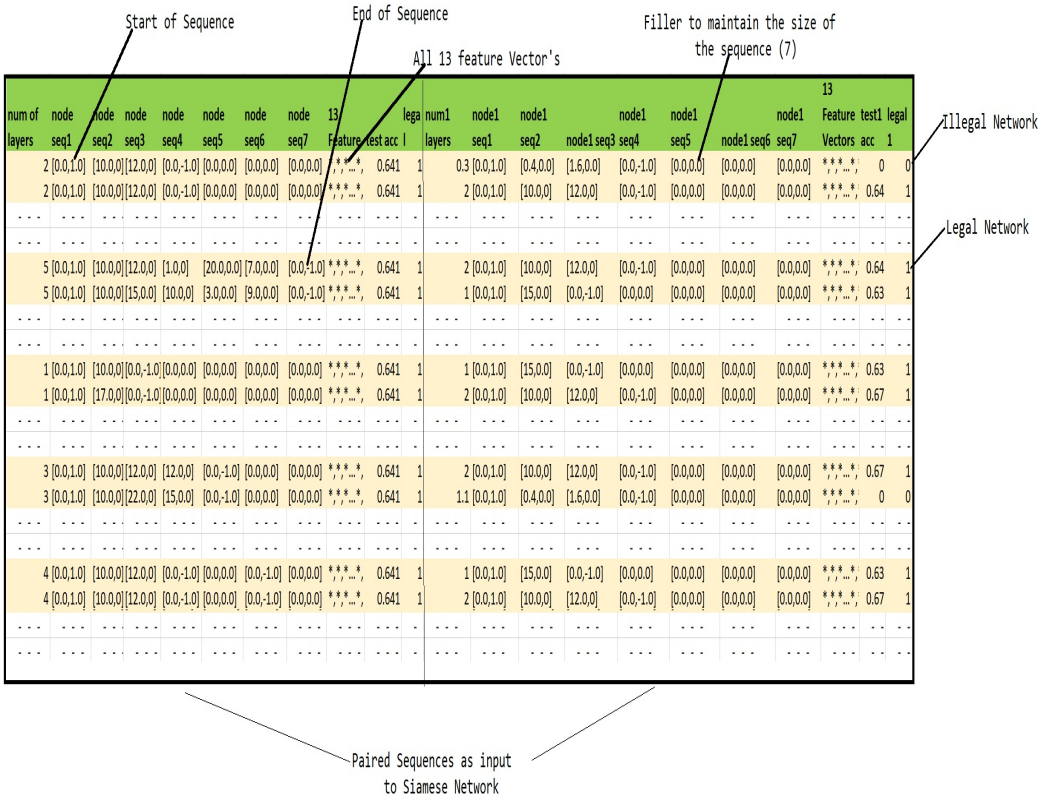


Figure 6.2: Initial Training Data Set With Problem Selection Attributes(Normalized)

Note, the labels array should contain the binary labels (legal(1), illegal(0)) corresponding to each pair.

After successful training, Siamese Network can compare new pairs of legal networks and determine their similarity. The model will co-locate similar legal networks by assigning them similar feature representations in the learned embedding space. To do this, we used the sister_network submodel to obtain the embeddings for any new legal networks and calculate the distance between the embeddings.

By comparing the distances for different pairs of legal networks, now we can identify which legal networks are closer together in the learned embedding space. This enabled us to group similar legal networks and analyze their relationships more effectively as shown in Figure 6.3

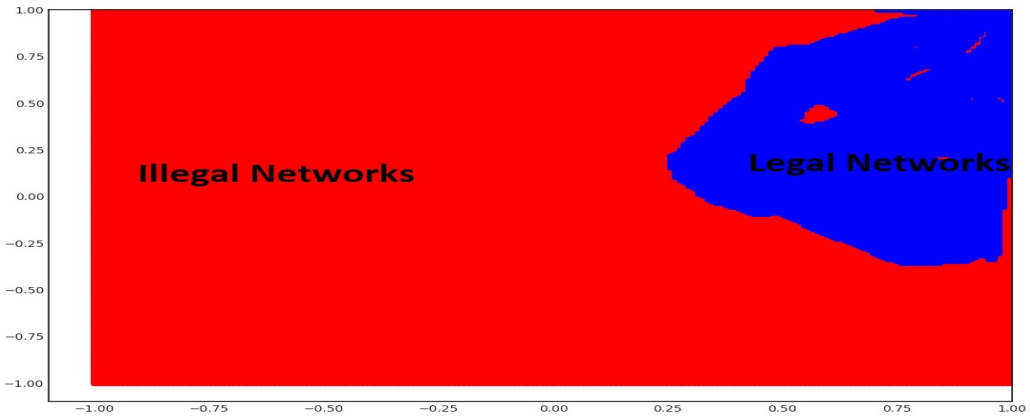


Figure 6.3: Separate Legal and Illegal Networks

The high level flow that interleaves planning and execution is shown as:

a) Initialise networks,

b) Initialise replay buffer,

c) Select and carry out action with exploration noise,

d) Store transitions,

e) Update critic,

f) Update actor,

g) Update target networks

h) Repeat until sentient.

6.4 Random Network generation and Training for Embeddings

The basic idea behind this technique is to randomly generate neural networks and train them on a given dataset to obtain embeddings. The trained embeddings can then be used for various downstream tasks, such as classification, clustering, or visualization.

Our random network generation and training process involves the following steps:

1. **Randomly generate a neural network architecture:** In this step, a neural network architecture is generated randomly using a set of pre-defined hyperparameters, such as the number of layers, the number of neurons per layer, the activation functions, and the learning rate using a sequence-to-sequence (seq2seq) model as shown in Figure 4.3.
2. **Train the network on the given dataset:** The randomly generated network is trained on a given dataset using a supervised learning approach. The input data is fed into the network, and the network learns to predict the corresponding output labels. The loss function used for training is a measure of the difference between the predicted labels and the true labels.
3. **Train Encoder and Decoder models:** Encoder and decoder were implemented and trained using a sequence-to-sequence (seq2seq) model to predict the architecture of a neural network given its time series data. Figure 4.3

shows a layout of input structure for network of max layers = 5 for our experiments. The model employs a bidirectional LSTM encoder-decoder architecture with attention mechanisms.

4. **Bidirectional LSTM-based Encoder:** The Bidirectional LSTM-based Encoder Model is a part of the autoencoder network used in the code. It is a combination of forward LSTM and backword LSTM, which can fit the data from both forward direction and backward direction, and concatenate the prediction. It is implemented to be responsible for encoding the input time series data into a lower-dimensional representation, which can be later decoded by the decoder network to predict the neural network architecture. The model consists of an input layer followed by three bidirectional LSTM layers, two dense layers, and an output layer (the embeddings). The input to the encoder is a time series with shape (timesteps, number of features). The output of the encoder is an embedding. The encoder model is compiled, and its summary is printed.

5. **LSTM-based Decoder Model:** Decoder Model takes the embeddings from the encoder as input and produces hidden states for the decoder LSTM. It has two dense layers and is compiled with the specified optimizer, loss function, and metric. The decoder predicts the neural network architecture very close to the one which was send to the embedding model with 98.9% accuracy. We also implemented 'Decoder Depth model' to predicts the depth (number of layers) of the neural network architecture. It has two dense layers and an output layer.

The model is compiled with the specified optimizer, loss function, and metric.

6. **Obtain the embeddings:** Once the embedding network is trained, the final layer of the network, which produces the output with the help of 'tanh' activation. The activations of the remaining layers are then used as embeddings for the input data. These embeddings are typically lower-dimensional than the original input data, making them easier to visualize and analyze.
7. **Evaluate the embeddings:** To validate the encoding space, encoder-decoder network are pre-trained using reconstruction loss (MSE) multiplied by legal value to see if the networks were able to form an effective embedding space for the legal and illegal network architectures.
   The trained embeddings are evaluated on by plotting a scatter plot within 2-Dimensional space with length between +1 to -1 in both x and y directions.

To validate the spatial consistency of the encoding space, the locations of legal and illegal network descriptions were evaluated and compared to the legal predictions. Figure 6.4 shows the results where blue points indicate valid networks while red dots indicate embedding points that do not correspond to legal network configurations. The shaded region indicates the area for which the system predicts legal networks.

One of the main advantages of random network generation and training for embeddings is that it allows for the discovery of novel neural network architectures that may be better suited for a particular dataset than pre-defined architectures. This technique also enables the creation of embeddings that are tailored to the specific needs of a given task, such as fine-grained classification or outlier detection.

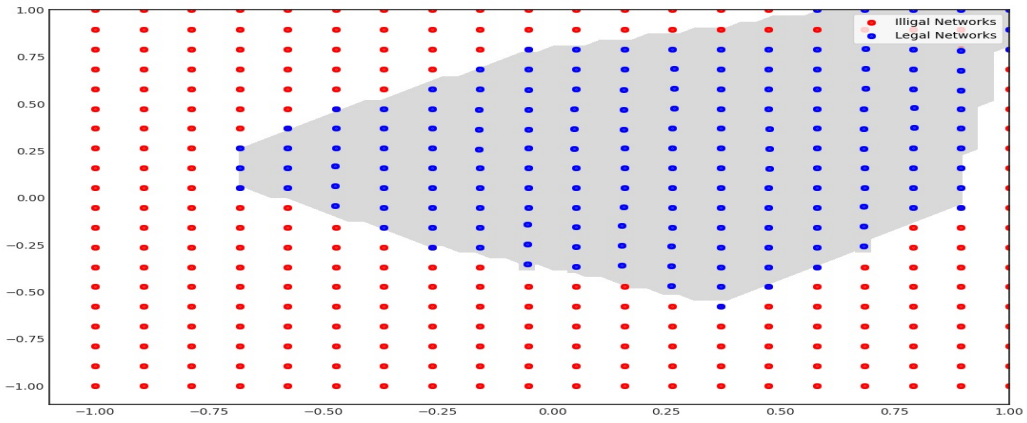


Figure 6.4: Separate Legal and Illegal Networks

However, one of the limitations of this technique is that it can be computationally expensive and time-consuming to train multiple neural networks on a given dataset. To address this, we automated the process of generating neural network architectures.

In summary, random network generation and training for embeddings will enable us the discovery of novel neural network architectures and allows for the creation of embeddings tailored to our future actor-critic training.

6.5 Results from Training Encoder-Decoder-Accuracy models

As explained above the encoder processes the input sequence and generates a fixed-length representation, often called a context vector or hidden state. The decoder then takes this representation and generates an output sequence based on it. Whereas, Accuracy Network measures how well the model's predictions match the actual target values. In the context of an Encoder-Decoder model, accuracy can be understood as the proportion of correct predictions (words or tokens) in the output sequence compared to the actual target sequence.

When training Encoder-Decoder models, several factors can influence their accuracy:

1. Model architecture: The choice of the underlying architecture, such as recurrent neural networks (RNNs), long short-term memory (LSTM) networks, or transformer-based models, can impact the model's performance. Generally, LSTMs and transformers have shown better results in sequence-to-sequence problems compared to vanilla RNNs due to their ability to capture long-range dependencies.

2. Attention mechanism: Incorporating an attention mechanism into the Encoder-Decoder model has been shown to significantly improve performance. The attention mechanism allows the decoder to focus on different parts of the input sequence when generating the output, providing a more context-aware prediction.

3. Training data: The quality and size of the training data are crucial factors that impact the model's performance. A larger and more diverse dataset helps the model generalize better to unseen data. It is also important to preprocess and clean the data, as well as handle class imbalance if it exists.

4. Hyperparameters: The choice of hyperparameters, such as the learning rate, batch size, and the number of layers and hidden units, can have a significant impact on the model's performance. It is common to use techniques like grid search or random search to find the optimal hyperparameters.

5. Regularization and optimization techniques: Techniques such as dropout, gradient clipping, and weight decay can be used to prevent overfitting and improve the model's generalization to unseen data. Additionally, advanced optimization algorithms like Adam, RMSprop, or Adagrad can help speed up training and potentially lead to better performance.

The results from training an Encoder-Decoder model can be analyzed by examining the loss curves for training and validation sets, as well as the accuracy

on these sets over time. Ideally, the loss should decrease, and the accuracy should increase during training. Monitoring these metrics helps identify potential issues such as overfitting, underfitting, or convergence problems.

In addition to accuracy, custom loss and decoder_accuracy_legal, decoder_loss ,binaryCrossEntrphy, legal_Network_Pred, custom_loss, mean_sqe_pred, acc_loss, legal_loss, distance_loss metrics are used. These metrics provide more nuanced insights into the model's performance and can helped us identify areas for improvement. The decoder_accuracy_legal matrix for decoder model specially helped us to check on performance of decoder network for all legal decoded networks. To have more effective Accuracy training we added Gaussian Noise and transform all 13 Feature Vectors to a dense set of 6 Feature Vector. A dropout layer is also added as dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This Feature Conversion model is designed to transform Feature Vectors before we introduce them to the Accuracy Network Figure 6.5

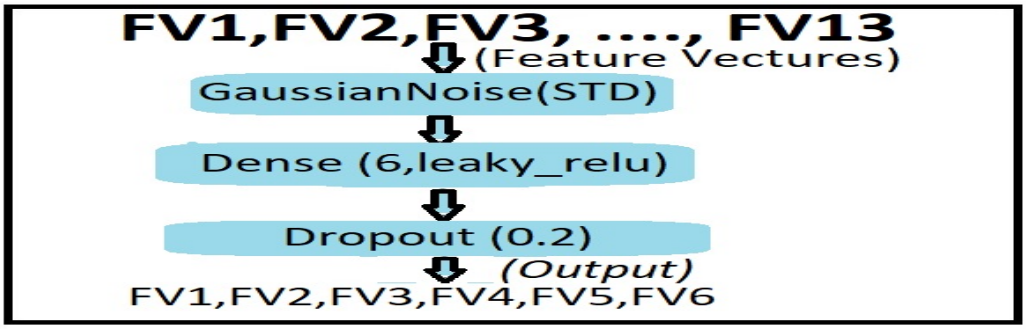


Figure 6.5: Feature Vector Transformation

During the training process, the model's weights and parameters are adjusted to minimize the error between its predictions and the actual labels. Once the

Encoder-Decoder-Accuracy models are jointly trained, it is tested on a separate dataset (the test set) to evaluate its performance for its accuracy. It is important to ensure that the model generalizes well to new data and is not overfitting the training data. Figure 6.6 shows the learned accuracy predictions for legal and illegal Networks using a specific training problem (FV1, FV2, FV3, FV4, FV5, FV6), corresponding to the embedding space.

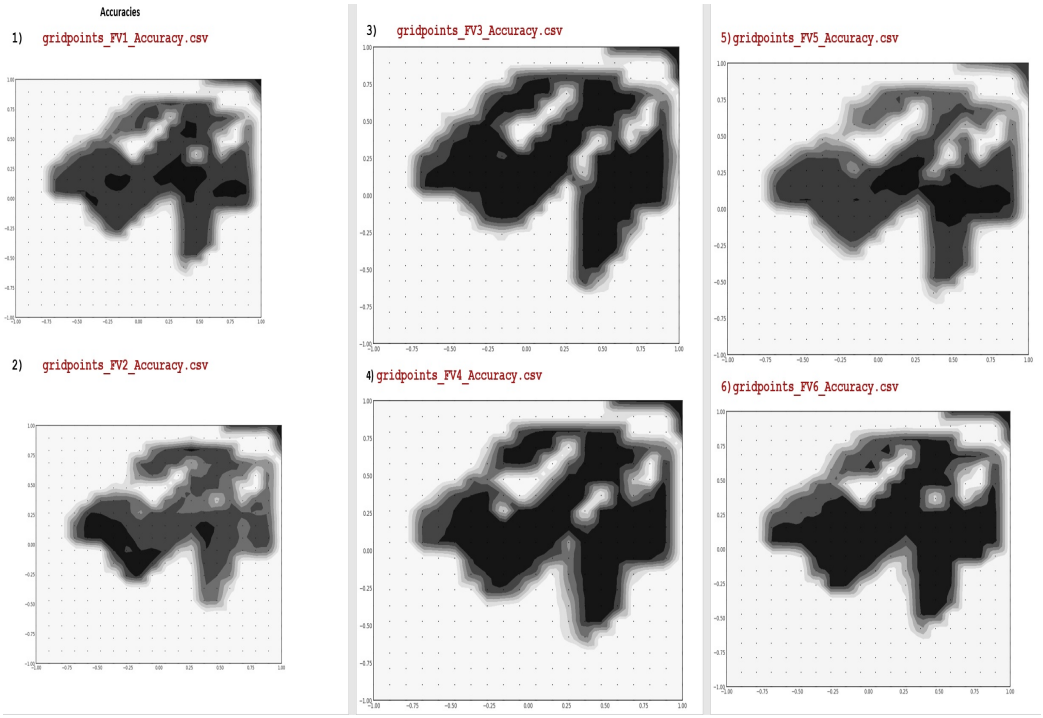


Figure 6.6: Accuracy Predictions for FV1, FV2, FV3, FV4, FV5, FV6

To evaluate the accuracy prediction, Figure 6.7 shows the learned predictions for a specific training problem (FV 7). By closely examining, we find that deep dark areas of the accuracy prediction show the highest accuracy networks.

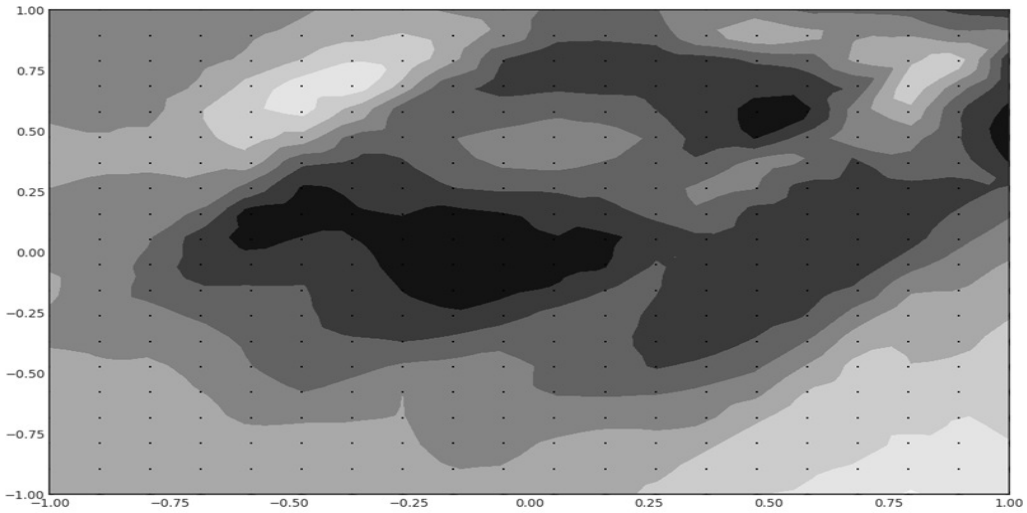


Figure 6.7: Accuracy Predictions for FV7

In summary, planning for the Accuracy model was simply through trial and error and we settled with 9 dense layers and two output layers using sigmoid activation function to output accuracy and status(legal/illegal) of the network. Input layer for Accuracy model takes the state vector, which as an output from first Concatenation layer consist of network properties and its 6 transformed Feature Vectors. We run network properties and Feature Vectors in parallel through set of dense layers to yield best results. Figure 6.8 show the layered diagram of the accuracy network and it's layer dependencies.

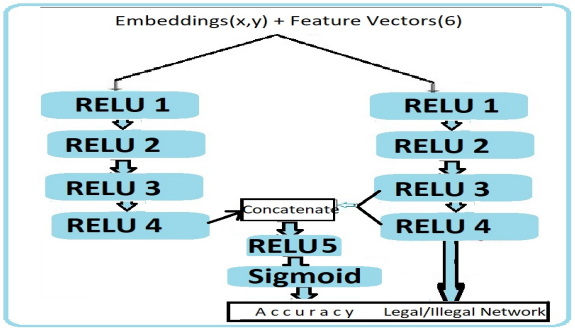


Figure 6.8: Combined Encoder-Decoder-Accuracy Training Results

The Encoder-Decoder model is pre-trained first to get best performance and then we freeze its weights and train them with the accuracy network. Finally we train them all together to yield the best performance of the combined network. A custom loss function for accuracy network was designed which uses binary cross entropy to calculate legal prediction loss and MSE to computer accuracy loss**(0.00208641)**. Custom loss function consists of decoder loss, accuracy loss, legal loss and distance loss. By trial and error we came up with their ratio to add up for component loss functions as shown below:

$$\begin{aligned} custom\_loss = {} & weightD*(decoder\_loss) + \\ & weightA*(accuracy\_loss) + \\ & weightL*legal\_loss + weightDist*distance\_loss \end{aligned}$$

Where:

$$legal\_loss = binary\_crossentropy(True\_Legal, (Predicted\_Legal))$$

$$acc\_loss = MSE(True\_Accuracy - Predicted\_Accuracy)) + legal\_loss$$

Here legal loss is the MSE of the network reconstruction, accuracy loss is the MSE of the accuracy prediction, legal loss is binary cross-entropy of legal network prediction, and distance loss is the distance error of the Siamese network. Weights were determined experimentally and the learned components were evaluated for consistency of the embedding space, accuracy of network architecture reconstruction, and precision of the accuracy and legal network predictions.

After training this combined model we obtained the following results, as shown in Figure 6.9. The training losses, as well as a manual investigation and testing of the embedding space and the accuracy prediction function shows that the system is able to organize the embedding space coherently and can relatively

accurately demarcate the area of legal network embeddings with a legal prediction accuracy of 99% on validation data.

Training loss(0.028641)	Validation loss (0.13546667)	Decoder Accuracy Legal (0.9905621)
Validation Decoder Accuracy Legal (1.0110701)	Decoder loss (0.0078525179)	Validation Decoder loss (0.002504)
Custom loss (0.0209068)	Validation Custom loss (0.100644)	Mean Sq. Predictions (5.94935663e-05)
Validation Mean Sq. Predictions (4.5128221e-05)	Binary Cross Entropy (0.01026996)	Validation Binary Cross Entropy (0.01026996)
Legal Network Prediction (0.99712759)	Validation Legal Network Prediction (0.9778597)	

Figure 6.9: Combined Encoder-Decoder-Accuracy Training Results

The accuracy prediction here achieved a Mean Square Error of $4.5 * 10^{-5}$, on validation data across all training problems, indicating its ability to learn to predict accuracy across a range of problems. Note that the system is only trained on a small region of the embedding space for illegal networks and thus accuracy predictions in illegal areas are hard to interpret.

6.6 Pre-training the Actor-Critic Model

To achieve more consistent results and facilitate further exploration of the embedding space, we employed a forward lookup strategy. This involved constructing a tree with a depth of 3 and 5 leaves at each node. To effectively train the actor and critic components for the reinforcement learning (RL) policy, a two-step pre-training process was implemented.

First, we pre-train the critic network using a uniform policy and a discount factor $\gamma = 0$. This step initializes the critic network with reward values, enabling us to evaluate whether the network's complexity is adequate to represent the function over the embedding space.

Following this, we pre-trained the actor network for the reward function while keeping the previously trained critic weights frozen. This process aims to assess if the actor network's complexity is sufficient to capture the nuances of a policy on the embedding space.

Once the actor network is pre-trained, we proceed to pre-train the critic network again. However, this time, we use the pre-trained policy to generate embedding trajectories and apply our final discount factor of $\gamma = 0.9$. This step ensures that the pre-trained critic is consistent with the policy before initiating the Twin Delayed Deep Deterministic (TD3) reinforcement learning training for both the critic and the policy components.

By employing this pre-training strategy, we can effectively mitigate fluctuations in results while exploring the embedding space, ensuring a more robust and accurate actor-critic model for reinforcement learning tasks.

To avoid fluctuations in results while exploring the embedding space, further exploration was needed. Forward lookup strategy was was used by building a tree of depth 3 with 5 leaves at each node. To effectively train actor and critic for RL policy we first pre-train our critic network for a uniform policy and a discount factor of $\gamma = 0$. This basically initializes the critic network with the reward values and serves the purpose to be able to assess whether the network is sufficiently complex to represent this function over the embedding space. Then we pre-train our actor network for the reward while previously trained critic Weights are frozen.

This again serves to assess whether the complexity of the actor network is sufficient to reflect the complexities of a policy on the embedding space.

Once the actor is trained, we again pre-train the critic, but this time using the pre-trained policy to generate embedding trajectories and using our final discount factor of $\gamma = 0.9$. This is to ensure that our pre-trained critic is consistent with the policy before we start TD3 RL training of both the critic and the policy.

6.7 Training Actor-Critic Models Together in TD3-based Actor-Critic Approach

Actor and critic networks were trained in multiple stages as indicated before to evaluate whether Reinforcement learning [63] can produce a policy that will optimize the network architecture for the UCI datasets used. The networks were trained on 9 UCI target problems with alternate individual pre-training of critic and actor serving the goal of evaluating the sufficiency of the used network structures and of stabilizing the initial value function estimate and consistent policy in order to avoid excessive random trajectory generation.

Once critic and actor have been pre-trained, they are trained together using the TD3 actor-critic approach [71] for a few iterations to fine-tune and synchronize them. This allows for better exploration and training for our final policy. Here trajectory data is constructed on-line using the perturbed current policy and used to augment a temporally decaying replay buffer. This buffer stores the states, actions, rewards, and new states, and random sampling on the buffer is used to obtain sufficient training examples to allow the networks to converge.

Rewards are predicted by passing next state to the accuracy model and computing the reward as the difference to the accuracy of the current state. This eliminates the need for further target problem training. The two critic networks

are here trained using a loss obtained from the Bellman equation that provides separate losses for $V(s)$ and $A(s,a)$ and the target critic network is updated by updating its weights in the direction of the better of the two critics (i.e. the one with the larger value). The actor is trained using policy gradient by propagating the gradient of the negative Advantage value as a loss back through the critic network.

We explore and perform actions with exploration noise for target policy smoothing in our continuous embedding action space. To regularize action noise, we add a small amount of random noise to the target averaging over several mini-batches. This stabilizes policies by returning higher values for actions that are more robust to noise and interference. Clipped noise is later added to the selected action when calculating targets to obtain higher values for more robust actions. These transitions are saved and used to update the critic, followed by the actor and target networks. At each stage of training, the actor network helps to build a policy that can find an optimal network for the original classification problem. This process is repeated until stable results are obtained.

The next accuracy value is calculated by passing this next state value to the Accuracy Model and finally computing the reward as the difference between the accuracy of the next state and the current state. This generates the utility value of one-step exploration Equation 6.1. Key exploration results such as state, next state, action, reward, and next accuracy are saved into the first replay buffer (RBuffer) for future training. This is a crucial part of the algorithm where most of the TD3 additional features are implemented.

Next, we sample a mini-batch of stored transitions from the replay buffer, i.e., random samples [60] for a batch size of 300 . We select an action for each of the states pulled from our mini-batch and apply target policy smoothing. Noise is added to this action and clipped to ensure that the noisy action isn't too far away from the

original action value. Once target_actions are computed then target Q values for the critic are computed. This is where the double critic networks (critic-pair) come into play. Q values for each target critic is computed and then the smallest of the two (critic_target and critic_target2) is chosen for target Q value.

Critic value for new states is computed by target critic evaluation of the next states and target actions, squeezed along the first dimension. It does the forward pass to get the value of the successor state for the best action. Whereas, predicted (critic) values are the output of the main critic network which takes states and actions from the buffer sample. This critic value is the value of the current state's with respect to original state and actions the agent actually took during the course of this episode. Which represents value of the action in a given state. The target value for the terminal new state is nothing but just the reward for every other state. According to the Bellman equation, the value of the target critic network is the reward plus the discounted value of the resulting state. If there is no successor state, the target value is the reward itself.

6.7.1 Critic optimization

Critic optimization is carried out by minimizing the loss, updating the critic by applying the gradients on the same critic_main trainable variables. The gradient tape is used to load operations to the computational graph for gradient calculations. The actor network is updated less frequently than the critic network. Note, gradient involves all the weights of all three networks but weights are applied to only main critic networks only. Apply gradients on same critic_main trainable_variables by using all the weights of the three layers in the main critic network.

6.7.2 Actor optimization

The actor's loss function computes the mean of the negative advantage values from the critic network with the actor selecting what action to take given the mini-batch of states. The actor loss is negative as we are doing gradient ascent. Actor network optimization is performed using backpropagation, as actor loss involves the actor_main and critic_main networks. Actor loss is calculated as the negative of critic main values with inputs as the main actor predicted actions.

The actor is much simpler to update when compared to the critic. We make sure that actor is updated fewer times than critic. Therefore, actor is updated every 2nd time step. The actions from the actor are selected based upon its current set of weights. Not based upon the weights it had at the time it is stored in an agent's memory (RBuffer). The actor's loss function simply gets the mean of the -Q values from our critic network with our actor choosing what action to take given the mini batch of states.

The actor_loss is shown as negative ('-') as we are doing gradient ascent here. As explained in policy gradient methods, gradient decent is not used because it will minimize the total score over the time and to maximize the total score it is required to use gradient ascent which is just negative of gradient decent. Actor network is optimized by using backpropagation as actor_loss involves actor_main & critic_main. Actor loss is calculated as negative of critic main values with inputs as the main actor predicted actions.

6.7.3 Target Network Updates

Target networks are time-delayed copies of their original networks, and they slowly track the learned networks. To avoid looping issues, actor target is used and actor_main is kept stable. Hard copies of the initial weights of both actor_main

and critic_ main networks are made to actor_target and critic_target networks, as target networks significantly improve stability in learning.

In the end, frozen target networks are updated using a soft update alongside the actor update. And actor_target and critic_target network are slowly moved using tau(τ) towards the trained actor_main and critic_main network via "soft updates".

Finally, the actor-critic network was trained for 5000 episodes with a new step being generated and 300 samples trained per episode. Actions generated by the actor are here displacement vectors in the 2 dimensional embedding space represented as the sine and cosine of the direction and a length between 0 and 1, and are multiplied by a scaling factor of $\delta = 0.005$ to result in steps in the embedding space as:

$$next\_state = state + \delta * length * \begin{pmatrix} cos(dir) \\ \sin(dir) \end{pmatrix} \tag{6.1}$$

6.8 Evaluating The Policy

Figure 6.10 shows the learned actor values while the Figure 6.11 shows the vector field representing the policy actions for the part of the embedding space networks. Investigating the critic values in the context of the corresponding accuracy values shown in Figure 6.7, reveals that the system successfully learned the utility function, which corresponds to the discounted sum of remaining network accuracy improvements from the given embedding space.

Examining the policy reveals a strategy that converts random starting networks to a small number of final networks which correspond closely to local maxima in the accuracy space. Once the training is finished, we evaluate the learned policy to see if it is able to transform the random start network into a high accuracy

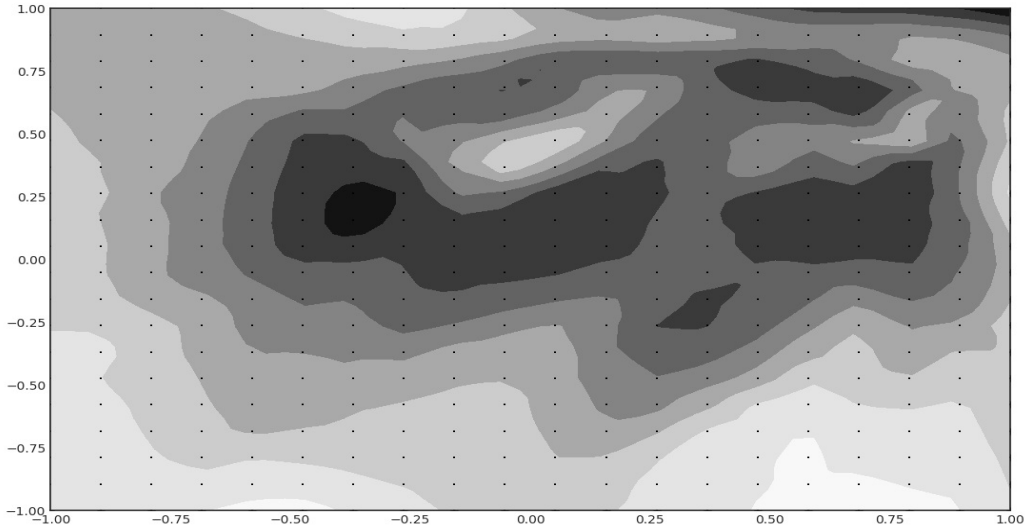


Figure 6.10: Critic Value over Embeddings of legal and illegal Networks

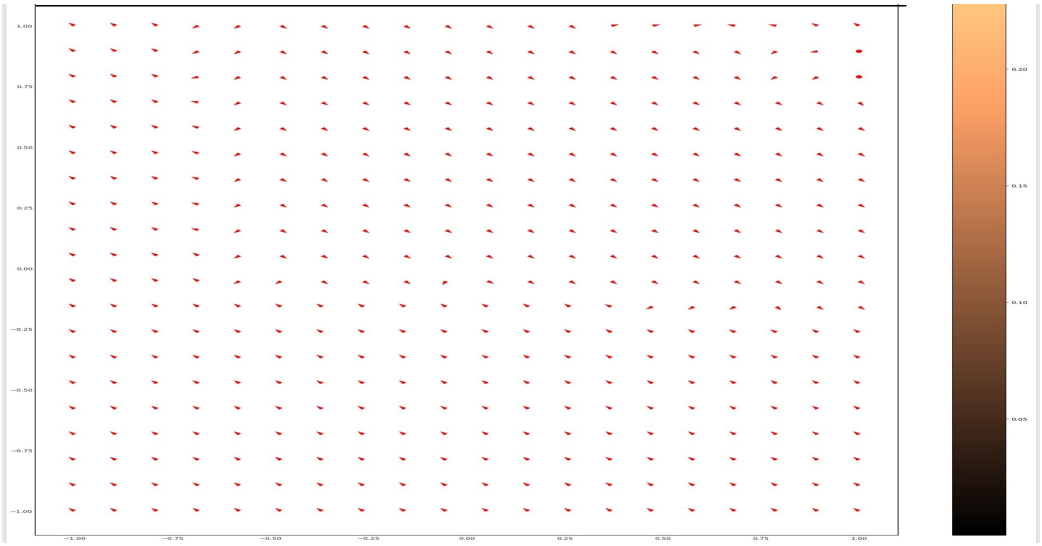


Figure 6.11: Learned Policy Actions (right) over Embeddings of legal and illegal Networks

final network for UCI classification problem from the test set and thus if it is able to generalize network architecture optimization across target problems.

6.8.1 Analyzing Policy Trajectories

Analyzing policy trajectories is an essential aspect of reinforcement learning, as it provides insights into how well the learned policy performs in transforming an initial state into a desired final state. By studying policy trajectories, researchers can understand the underlying strategies and patterns that the agent employs during its exploration and exploitation process. This information can be crucial for improving the learning algorithm, identifying potential issues, and optimizing the agent's performance.

During our experiments of the TD3-based actor-critic approach for network optimization, policy trajectories represent the transformation of random starting networks into high-performing networks through a series of actions. Analyzing these trajectories can help identify the strategies used by the agent and reveal areas of improvement in the learning process.

Here are a few key aspects to consider when analyzing policy trajectories:

Convergence: Examine the convergence of the trajectories towards optimal or near-optimal solutions. Rapid convergence to high-performing networks is an indicator of an effective policy. If convergence is slow or unstable, it may suggest the need for adjustments in the learning algorithm or exploration strategy.

Consistency: Observe the consistency of the trajectories in reaching high-performing networks across different initial conditions. A robust policy should perform well irrespective of the starting network. Inconsistency in the trajectories might suggest that the policy is sensitive to initial conditions, which can be addressed by refining the learning process.

Exploration vs. Exploitation: Analyzing policy trajectories can provide insights into the balance between exploration (searching for new promising regions in the solution space) and exploitation (refining solutions in the current region). Ideally,

the agent should strike a balance between these two aspects to maximize its performance. If the policy is too exploratory, it may fail to adequately exploit promising solutions, whereas if it is too exploitative, it may get stuck in local optima.

Strategy patterns: By visualizing and studying the policy trajectories, researchers can identify patterns in the agent's strategy. For instance, they can observe if the agent tends to focus on specific parts of the embedding space or if it follows certain types of actions more frequently. Understanding these patterns can guide further improvements in the learning algorithm or exploration strategy.

Comparing with baselines: Comparing policy trajectories with those of baseline methods or human-designed architectures can provide valuable insights into the effectiveness of the learning algorithm. If the learned policy performs comparably or better than these baselines, it can be considered a successful approach.

By analyzing policy trajectories in the context of the TD3-based actor-critic approach for network optimization, researchers can gain a deeper understanding of the agent's learning process and identify areas for improvement. This, in turn, can lead to the development of more efficient and effective policies for network optimization and other machine learning tasks.

Our results show that the policy successfully transforms simple fully connected random networks into high-performing networks, matching or surpassing the performance of the best networks in the initial training set for system pre-training. Figure 6.12 shows 8 policy trajectories generated by the policy from random start networks for one of the test problems. These trajectories show that the policy manages to transform random networks into high performing networks that match or exceed the performance of the best of the 300 initial random networks even for problems that were never seen during training.

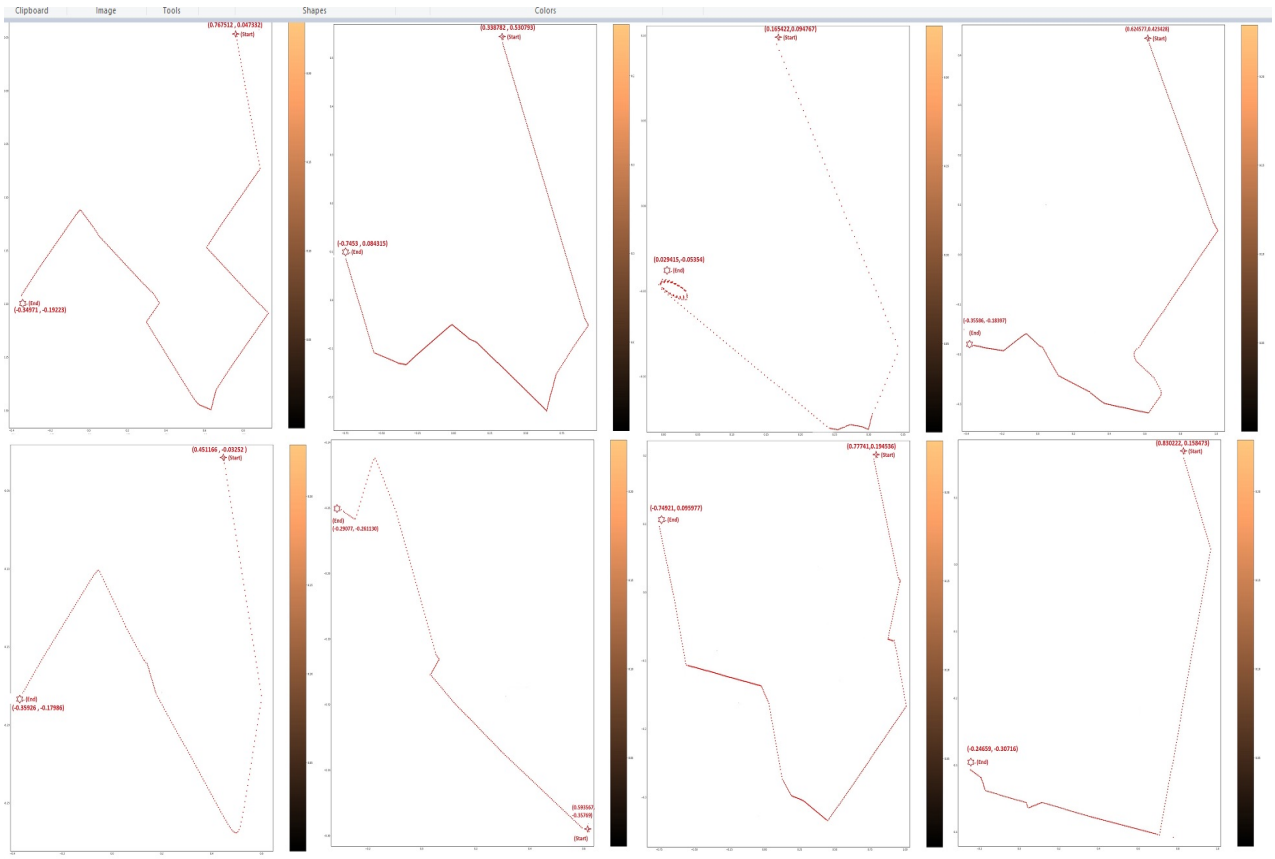


Figure 6.12: Policy Trajectories on fully connected Simple Random Starting Networks

Figures 6.13 and 6.14 shows 12 policy trajectories generated from random more complex flexible length networks for all 9 training and 3 test problems. These trajectories also confirm that the policy manages to transform random networks into high performing networks that match or exceed the performance of the best of the 300 initial random networks even for problems that were never seen during training.

Note, the policy here at times can not follow a straight line but has to traverse accuracy ”valleys”, effectively leaving local maxima in favor of better, more globally optimal networks. This demonstrates that the architecture can effectively

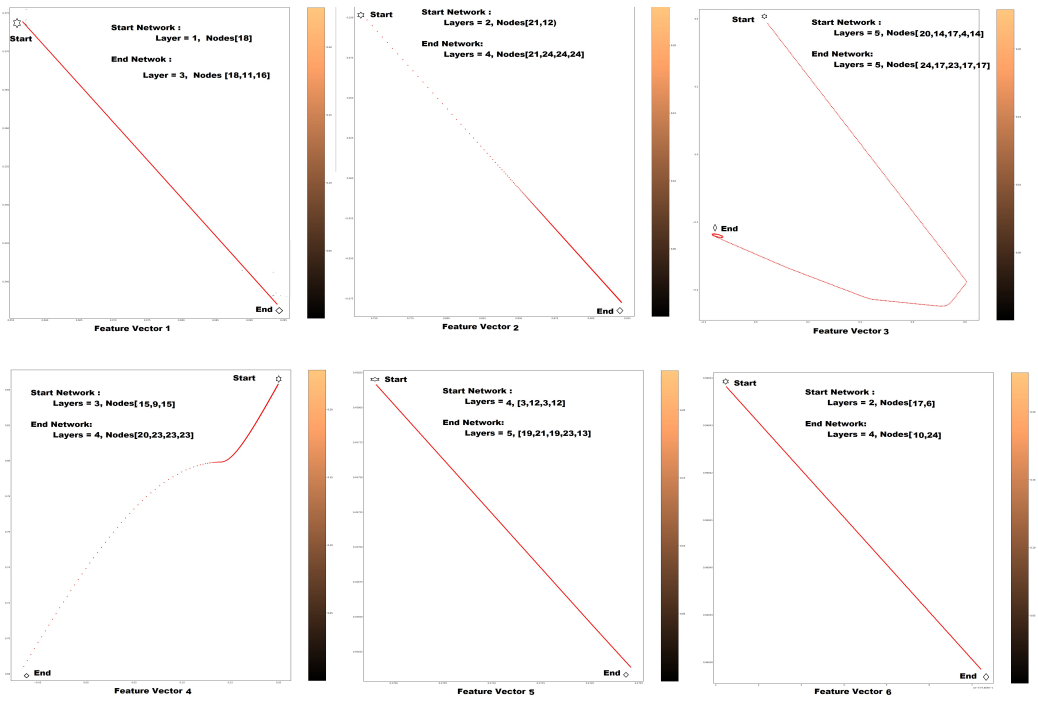


Figure 6.13: Policy Trajectories on complex flexible layers Random Starting Networks-I

labelfig:65-

learn an embedding space, a performance predictor, a critic network, and an actor for the presented network optimization problem.

6.9 Comparing our Results with Initial Networks:

The performance or accuracy comparison, between the best of the initial, randomly generated networks and the networks discovered by the policy provides a means to assess the effectiveness of the reinforcement learning approach, such as the TD3-based actor-critic algorithm, in optimizing network architectures.

To evaluate the success of the learned policy, we compared the performance of the networks discovered by the policy with the performance of the initial networks.

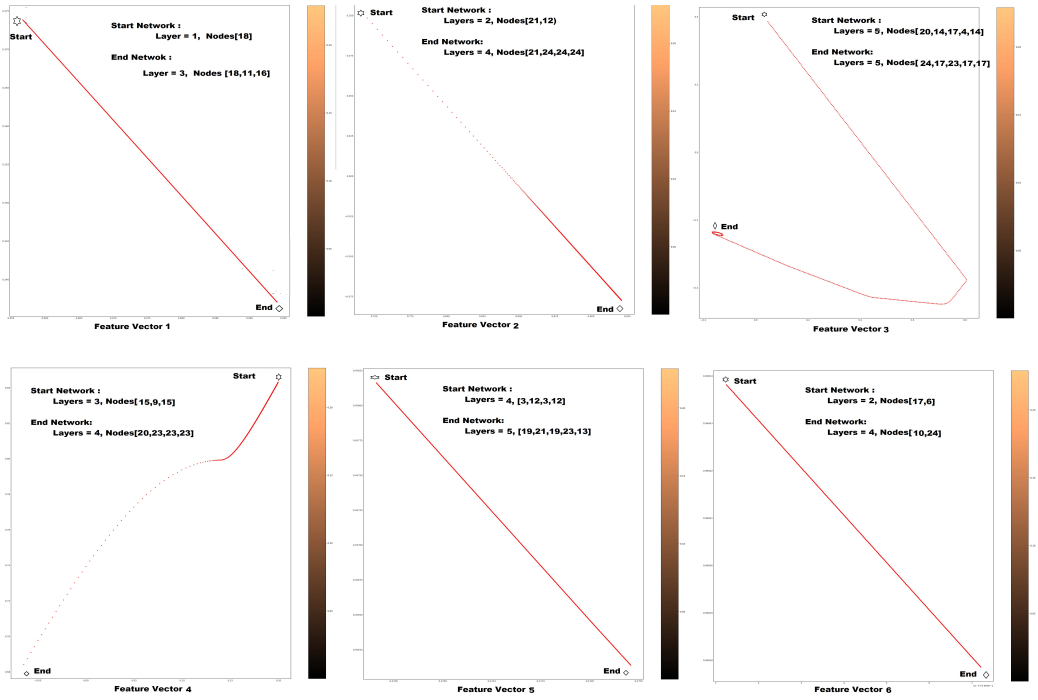


Figure 6.14: Policy Trajectories on complex flexible layers Random Starting Networks-II

The goal is to demonstrate that the policy can transform networks into optimized networks with higher or equivalent accuracies.

This comparison can be performed in several ways:

Overall accuracy improvement: Calculate the average accuracy improvement across all initial networks and the networks discovered by the policy. If the average accuracy of the discovered networks is significantly higher than the initial networks, it indicates that the policy is effective in optimizing network architectures.

Individual accuracy improvement: Compare the accuracy of each initial network with the corresponding optimized network discovered by the policy. This analysis can help identify specific cases where the policy performs exceptionally well or

struggles to improve the initial network's performance. By examining these cases, we can gain insights into potential strengths and weaknesses of the policy.

Best-case comparison: Evaluate the best-performing networks discovered by the policy and compare them with the best-performing networks in the initial set. If the discovered networks can match or surpass the performance of the best initial networks, it demonstrates the policy's ability to find competitive network architectures.

Distribution of accuracy: Analyze the distribution of accuracy scores for both the initial networks and the networks discovered by the policy. A shift towards higher accuracy scores in the discovered networks indicates a successful policy. Additionally, we can examine the spread of the accuracy scores to identify any potential issues, such as overfitting or underfitting, in the discovered networks.

Robustness and generalization: Assess the robustness and generalization capabilities of the networks discovered by the policy. A successful policy should produce networks that perform well across a variety of tasks, datasets, and conditions. By comparing the performance of the initial networks and the discovered networks in various settings, we can evaluate the policy's ability to generate robust and generalizable architectures.

By expanding on the performance metric or accuracy comparison between the initial networks and the networks discovered by the policy, we can better understand the effectiveness of their reinforcement learning approach, such as the TD3-based actor-critic algorithm, in optimizing network architectures. This analysis can guide further improvements in the learning algorithm and help develop more efficient and effective machine learning solutions.

Figure 6.15 further validates our results by comparing the accuracy achieved by the learned policy on the 8 training and 3 test problems during 10 policy execu-

Problem (Feature Vector)	Test NN1	Test NN2	Test NN3	Test NN4	Test NN5	Test NN6	Test NN7	Test NN8	Test NN9	Test NN10	Avg. Accuracy	Std Deviation	Best Accuracy from Orig. Network	Accuracy Found / Best Network Accuracy (Ratio)
FV1 (2,8,24)	0.72	0.71	0.70	0.71	0.70	0.70	0.71	0.71	0.70	0.71	0.71	0.005	0.72	0.98
FV2 (2,8,24)	0.62	0.65	0.62	0.55	0.58	0.68	0.58	0.58	0.51	0.55	0.59	0.051	0.58	1.017
FV3 (2,8,24)	0.90	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.002	0.97	0.94
FV4 (2,8,24)	1	1	1	0.99	1	1	1	1	0.99	1	0.99	0.003	1	0.99
FV5 (2,8,24)	0.63	0.65	0.68	0.63	0.68	0.69	0.70	0.67	0.64		0.66	0.026	0.74	0.90
FV6 (2,8,24)	0.84	0.83	0.85	0.83	0.84	0.86	0.86	0.83	0.86	0.85	0.84	0.009	0.83	1.019
FV7(2,8,24)	0.75	0.59	0.90	0.84	0.57	0.91	0.80	1.18	1.01	0.64	0.82	0.19	0.75	1.09
FV8	1	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.88	0.88	0.93	0.03	0.94	0.99
Test – FV10	0.88	0.88	0.88	0.83	0.83	0.82	0.79	0.80	0.80	0.80	0.83	0.034	0.82	1.016
Test – FV11	0.95	0.95	0.95	0.95	0.97	0.96		0.95		0.95	0.96	0.006	0.96	0.99
Test – FV12		0.83	0.88	0.85	0.87	0.91		0.91			0.88	0.03	0.87	1.012

Figure 6.15: Performance Matrix or Accuracy between Initial Network Vs Network's discovered by Policy

tions from random start networks. Figure 6.16 validates our results by comparing the accuracy achieved by the learned policy on the 9 training and 3 test problems during 10 policy executions from random start networks.

Here the right-most column displays the ratio of the accuracy achieved by the network optimized by the learned policy and the best among the 300 initial random networks and shows that the networks found by the learned optimization policy achieve better or equivalent accuracies even for the never before seen test problems.

By analyzing the performance metrics or accuracy between the initial network and networks discovered by the policy, as shown in Figure 6.15, we can further validate the efficacy of the TD3-based actor-critic approach in network optimization problems. The extreme right column of the figure demonstrates that the networks discovered at the end of the trajectories not only have better or equivalent

Problems (Feature Vector)	MORE COMPLEX FLEXIBLE LAYERS NETWORK'S (Accuracies)												
Problem (Feature Vector)	Test NN1	Test NN2	Test NN3	Test NN4	Test NN5	Test NN6	Test NN7	Test NN8	Test NN9	Test NN10	Avg. Accuracy	Std Deviation	Accuracy Found / Best Network Accuracy (Ratio)
FV1 (18,11,16)	0.82	0.81	0.82	0.83	0.83	0.81	0.79	0.81	0.80	0.80	0.81	$0.11x10^{-1}$	0.97
FV2 (21,24,24,24)	0.85	0.84	0.85	0.88	0.85	0.86	0.78	0.85	0.81	0.85	0.85	$0.24x10^{-1}$	0.93
FV3 (24,17,23,17,17)	0.65	0.65	0.64	0.65	0.56	0.62	0.66	0.65	0.65	0.65	0.64	$0.28x10^{-1}$	0.99
FV4 (20,23,23,23)	0.66	0.62	0.65	0.65	0.65	0.65	0.64	0.65	0.65	0.66	0.65	0.01	1.00
FV5 (19,21,19,23,13)	0.74	0.73	0.73	0.74	0.75	0.72	0.74	0.68	0.73	0.74	0.73	$0.17x10^{-1}$	0.89
FV6 (10,24)	0.70	0.70	0.70	0.70		0.70		0.70	0.72	0.69	0.70	$0.59x10^{-2}$	0.96
FV7(23,22,24)	0.67		0.72	0.66	0.57	0.6	0.58	0.69	0.71	0.68	0.66	0.05	0.91
FV8(24,24,24,24)			0.77	0.75	0.77	0.67	0.68	0.75	0.77	0.76	0.74	0.03	0.98
FV9(23,24,22,24,24)	0.59		0.55	0.64	1.01	0.61	0.63	0.64	0.54	0.62	0.58	0.13	0.98
Test 1-- FV10(9,21,17,14,4)	0.70	0.61	0.66	0.71	0.63	0.56	0.71	0.68	0.71	0.70	0.66	0.05	1.02
Test 2--FV11(7,16)	0.86	0.85	0.72		0.82	0.67	0.79	0.86	0.86	0.85	0.81	0.06	0.99
Test 3-- FV12(17,20,22,23)	0.85	0.85	0.80	0.65	0.78	0.82	0.78	0.85	0.81	0.82	0.80	0.05	0.99

Figure 6.16: Performance Matrix or Accuracy between Complex Initial Network Vs Network's discovered by Policy

accuracies than the originally trained best networks but also showcase the potential to outperform them.

Figure 6.16, also further validate for more complex flexible layered Network in network optimization problems. Again the extreme right column of the figure demonstrates that the networks discovered at the end of the trajectories are better or equivalent in term of accuracies than the originally trained best networks and also show the potential to outperform them.

CHAPTER 7

Conclusions and Future Work

This dissertation introduces a novel and efficient method for optimizing deep learning network architectures by employing Reinforcement Learning, abstract problem embeddings, and transferable policies. Through initial experimentation and evaluation, this research demonstrates the potential of this approach to significantly enhance the optimization process for deep learning systems, making them more accessible and efficient for both non-experts and experts. By bridging the gap between intricate network architectures and practical applications, this innovative approach sets the stage for advancements in deep learning and AI-driven solutions across a wide range of industries and domains.

7.1 The Proposed Approach

The proposed approach in this dissertation optimizes network architectures in a learned network embedding space, constructed using an autoencoder and Siamese network. Leveraging this embedding space and a performance predictor, an actor-critic RL component is trained, which uses the embedding vector as a network representation, as well as problem-specific features, to learn a policy for network architecture modification over the continuous action space of embedding space moves. Initial experiments using classification datasets from the UCI repository demonstrate that the proposed system successfully learns an embedding space and a policy that derives near-optimal network architectures, even for unseen test problems, illustrating that the TD3-based actor-critic approach is a potent method

for optimizing network architectures across various machine learning tasks. By iteratively training and fine-tuning the actor and critic networks, the system effectively learns an embedding space, a performance predictor, and the optimal policy for network optimization. The results show that the learned policy can transform random starting networks into high-performing networks that match or surpass the performance of the best networks in the initial training set. It also shows that this ability generalizes beyond the target problems in the training set and extends to novel problems that the system has not yet seen without requiring any additional training. This highlights the potential of this approach to deliver significant improvements in accuracy and performance across a range of classification and optimization problems.

7.2 Future Enhancements

Building upon the promising results of the TD3-based actor-critic approach for optimizing network architectures, there are several potential avenues for further exploration and development.

Hyperparameter Optimization: The current framework can be extended to incorporate hyperparameter optimization, enabling the discovery of optimal combinations of network architectures and hyperparameters. This would provide a more comprehensive solution for designing efficient deep learning systems.

Multi-task Learning: The transferable policy learned using this approach can be applied to multi-task learning scenarios where multiple related tasks are solved simultaneously. By learning a shared policy for modifying network architectures, the approach could potentially enable the discovery of networks that perform well on multiple tasks.

Scalability and Parallelization: To tackle larger and more complex deep learning problems, the current approach can be scaled up and parallelized. By distributing the training process across multiple computational nodes, it can effectively handle more extensive network architectures and larger datasets, potentially leading to improved performance and accuracy.

Exploration Strategies: The exploration strategy employed in the current approach can be further refined to balance the trade-off between exploration and exploitation more effectively. More sophisticated exploration strategies, such as intrinsic motivation or curiosity-driven methods, can be incorporated to guide the learning process towards more promising areas of the search space.

Combining Different RL Algorithms: The current approach relies on the TD3 algorithm for training the actor-critic networks. Investigating the combination of different reinforcement learning algorithms, such as PPO, SAC, or DDPG, could provide valuable insights into the most effective methods for network optimization.

REFERENCES

[1] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," *Advances in neural information processing systems*, vol. 31, 2018.

[2] S. Bose and M. Huber, "Incremental learning of neural network classifiers using reinforcement learning," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2016, pp. 002 097–002 103.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[7] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[8] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[9] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2. Ieee, 1999, pp. 1150–1157.

[10] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, vol. 1. Ieee, 2005, pp. 886–893.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[14] K. Kapanova, I. Dimov, and J. Sellier, "A genetic approach to automatic neural network architecture optimization," *Neural Computing and Applications*, vol. 29, no. 5, pp. 1481–1492, 2018.

[15] M. Gaurav. (2019) How to find the optimum number of hidden layers and nodes in a neural network model?

[Online]. Available: https://datagraphi.com/blog/post/2019/12/17/how-to-find-the-optimum-number-of-hidden-layers-and-nodes-in-a-neural-network-model

[16] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," *arXiv preprint arXiv:1705.10823*, 2017.

[17] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[18] Z.-H. Zhou, J. Wu, and W. Tang, "Ensembling neural networks: many could be better than all," *Artificial intelligence*, vol. 137, no. 1-2, pp. 239–263, 2002.

[19] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[20] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1188–1196. [Online]. Available: https://proceedings.mlr.press/v32/le14.html

[21] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[22] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[23] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[24] X. Jin, J. Wang, J. Slocum, M.-H. Yang, S. Dai, S. Yan, and J. Feng, "Rc-darts: Resource constrained differentiable architecture search," *arXiv preprint arXiv:1912.12814*, 2019.

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

[26] W. N. Street and Y. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 2001, pp. 377–382.

[27] P. M. Granitto, P. F. Verdes, and H. A. Ceccatto, "Neural network ensembles: evaluation of aggregation algorithms," *Artificial Intelligence*, vol. 163, no. 2, pp. 139–162, 2005.

[28] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: one-shot model architecture search through hypernetworks," *arXiv preprint arXiv:1708.05344*, 2017.

[29] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Reinforcement learning for architecture search by network transformation," *arXiv preprint arXiv:1707.04873*, vol. 4, p. 3, 2017.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[31] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, pp. 55:1–55:21, 2018.

[32] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Reinforcement learning*, pp. 5–32, 1992.

[33] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[34] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

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

[37] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[38] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *International conference on machine learning*. PMLR, 2018, pp. 4095–4104.

[39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*. Springer, 2016, pp. 525–542.

[40] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[41] W. J. Yun, J. Park, and J. Kim, "Quantum multi-agent meta reinforcement learning," *arXiv preprint arXiv:2208.11510*, 2022.

[42] R. J. WILLIAMS and J. PENG, "Function optimization using connectionist reinforcement learning algorithms," *Connection Science*, vol. 3, no. 3, pp. 241–268, 1991. [Online]. Available: https://doi.org/10.1080/09540099108946587

[43] R. Rifkin and A. Klautau, "In defense of one-vs-all classification," *The Journal of Machine Learning Research*, vol. 5, pp. 101–141, 2004.

[44] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, ser. Proceedings of Machine Learning Research, R. P. Adams and V. Gogate, Eds., vol. 115. PMLR, 22–25 Jul 2020, pp. 367–377. [Online]. Available: https://proceedings.mlr.press/v115/li20c.html

[45] . T. Q. Chen X. Xie, L. Wu J., "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," *Proceedings of ICCV*, 2020. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2019/papers/Chen_Progressive_Differentiable_Architecture_Search_Bridging_the_Depth_Gap_Between_Search_ICCV_2019_paper.pdf

[46] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *arXiv: Computer Vision and Pattern Recognition*, 2015.

[47] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. H. Černocký, "Strategies for training large scale neural network language models," *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, pp. 196–201, 2011.

[48] J. Ma, R. P. Sheridan, A. Liaw, G. E. Dahl, and V. Svetnik, "Deep neural nets as a method for quantitative structure-activity relationships," *Journal of chemical information and modeling*, vol. 55 2, pp. 263–74, 2015.

[49] M. Helmstaedter, K. Briggman, S. Turaga, V. Jain, H. Seung, and W. Denk, "Connectomic reconstruction of the inner plexiform layer in the mouse retina," *Nature*, vol. 500, no. 7461, pp. 168–174, 2013.

[50] H. Y. Xiong, B. Alipanahi, L. J. Lee, H. Bretschneider, D. Merico, R. K. C. Yuen, Y. Hua, S. Gueroussov, H. S. Najafabadi, T. R. Hughes, Q. Morris, Y. Barash, A. R. Krainer, N. Jojic, S. W. Scherer, B. J. Blencowe, and B. J. Frey, "Rna splicing. the human splicing code reveals

new insights into the genetic determinants of disease," *Science (New York, N.Y.)*, vol. 347, no. 6218, p. 1254806, January 2015. [Online]. Available: https://europepmc.org/articles/PMC4362528

[51] A. G. Barto, R. S. Sutton, and C. Watkins, "Sequential decision problems and neural networks," *Advances in neural information processing systems*, vol. 2, 1989.

[52] S. Benhur, "A friendly introduction to siamese networks," *Data Science and Machine Learning*, 2022. [Online]. Available: https://builtin.com/machine-learning/siamese-network

[53] G. R. Koch, "Siamese neural networks for one-shot image recognition," 2015.

[54] L. G. Hafemann, R. Sabourin, and L. S. Oliveira, "Learning features for offline handwritten signature verification using deep convolutional neural networks," *Pattern Recognition*, vol. 70, pp. 163–176, oct 2017. [Online]. Available: https://doi.org/10.1016%2Fj.patcog.2017.05.012

[55] S. Zagoruyko and N. Komodakis, "Learning to compare image patches via convolutional neural networks," 2015.

[56] N. Street and Y. Kim, "A streaming ensemble algorithm (sea) for large-scale classification," 07 2001, pp. 377–382.

[57] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002200009791504X

[58] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, "Efficient solution algorithms for factored mdps," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003.

[59] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[60] R. S. Sutton, S. D. Whitehead, *et al.*, "Online learning with random representations," in *Proceedings of the Tenth International Conference on Machine Learning*, 2014, pp. 314–321.

[61] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering Cambridge, UK, 1994, vol. 37.

[62] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in neural information processing systems*, vol. 8, 1995.

[63] A. Likas, "A reinforcement learning approach to online clustering," *Neural computation*, vol. 11, no. 8, pp. 1915–1932, 1999.

[64] W. T. B. Uther and M. M. Veloso, "Tree based discretization for continuous state space reinforcement learning," in *AAAI/IAAI*, 1998.

[65] B. Ravindran and A. Barto, "Smdp homomorphisms: An algebraic approach to abstraction in semi-markov decision processes," 05 2003.

[66] J. TORRES. (2020) The bellman equation, v-function and q-function explained. [Online]. Available: https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7

[67] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 1587–1596. [Online]. Available: https://proceedings.mlr.press/v80/fujimoto18a.html

[68] C. Yoon. (2019) Deep deterministic policy gradients explained. [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

[69] ——. (2018) Deriving policy gradients and implementing reinforce. [Online]. Available: https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

[70] ——. (2019) Deep deterministic policy gradients explained. [Online]. Available: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

[71] D. Byrne. (2019) Td3: Learning to run with ai. [Online]. Available: https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

[72] F. Chollet. (2017) A ten-minute introduction to sequence-sequence learning in keras. [Online]. Available: https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

[73] A. Graves, *Long Short-Term Memory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–45. [Online]. Available: https://doi.org/10.1007/978-3-642-24797-2_4

[74] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[75] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

BIOGRAPHICAL STATEMENT

Raghav Vadhera a.k.a. Bhanu Vadhera was born in Kanpur, India, in 1969. He received his B.S. and M.S degree from DEI University, Agra, INDIA, in 1988, his M.Tech. in Computer Science degree from IIT Delhi, in 1993, his ALM Degree in Technology Management from Harvard University in 2011 and Ph.D. degrees from The University of Texas at Arlington in 2023 , respectively. From 1990 to 1994, he was associated with the department of Computer Science and Engineering at IIT Delhi, as Senior Scientific Officer. In 1996, he joined BankBoston and later MIT Lincoln Labs as Research Scientist then as a Principle Engineer at Raytheon for the AI and ML related projects.

His current research interest are in the area of Reinforcement Learning, Machine Learning and in Explainable AI. His complete profile can be found on Linkedin @ https://www.linkedin.com/in/bvadhera