12-13-2022 2:00 PM

Algorithmic Improvements In Deep Reinforcement Learning

Norman L. Tasfi, *The University of Western Ontario*

Supervisor: Capretz, Miriam, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering

Abstract

Reinforcement Learning (RL) has seen exponential performance improvements over the past decade, achieving super-human performance across many domains. Deep Reinforcement Learning (DRL), the combination of RL methods with deep neural networks (DNN) as function approximators, has unlocked much of this progress. The path to generalized artificial intelligence (GAI) will depend on deep learning (DL) and RL. However, much work is required before the technology reaches anything resembling GAI. Therefore, this thesis focuses on a subset of areas within RL that require additional research to advance the field, specifically: sample efficiency, planning, and task transfer.

The first area, sample efficiency, refers to the amount of data an algorithm requires before converging to stable performance. Within RL, all models require immense amounts of samples from the environment, a far cry from other sub-areas, such as supervised learning which often require an order of magnitude fewer samples. This research proposes a method that learns to reuse previously seen data instead of throwing it away to improve sample efficiency by a factor at 2x, while training 30% faster than state-of-the-art methods.

The second area is planning within RL, where planning refers to an agent using an environment model to predict how possible actions will affect its performance. Improved planning in RL leads to increased performance as the model gains context on its next action. This thesis proposes a model that learns how to act optimally in an environment through a dynamic planning mechanism that adapts on the fly. This dynamic planning ability gives the resulting RL model immense flexibility as it can adapt to the demand of particular states on the environment and outperforms related methods by 30-45%.

The final area is that of task transfer, which deals with how readily a model trained on one task can transfer its knowledge to another related task within the same environment. RL models must be fully retrained on the new task even if the environment structure does not change. Here, we introduce two contributions that improve an existing transfer framework known as the Successor Features (SF). The first introduces a reward model with greater flexibility with stronger performance and transfer abilities than baseline models; achieving nearly 2x the reward on highly demanding tasks. The second contribution rephrases the SF framework as a simple pair of supervised tasks that can dynamically induce policies, drastically simplifying the learning problem and while matching performance.

Keywords: Reinforcement Learning, Deep Learning, Sample Efficiency, Planning, Task Transfer, Successor Features.

Summary For Lay Audience

In Artificial Intelligence (AI), different algorithms are used to solve problems automatically. Some algorithms are designed to excel in different mediums, such as visual tasks, translation of words, or exerting control over other systems. The field of Reinforcement Learning (RL) deals with algorithms that learn to exert control over other systems; they learn to do this automatically without any external direction besides a reward and punishment system, similar to how humans learn. It was discovered that the performance of RL algorithms could be improved significantly by another type of technology known as deep learning. The resulting performance increase of these Deep Reinforcement Learning (DRL) algorithms has been so significant that they have been able to best human experts. A well-known example of one of these algorithms is AlphaGo, which beat grandmasters at the game of Go without any special instruction; it learned to do this on its own by playing the game.

However, these algorithms could be more efficient; they take a very long time to learn, use a lot of energy and are unable to learn as well as a human from small amounts of data. It makes sense to label these types of algorithms as quite inefficient.

This thesis contributes new DRL algorithms to improve the efficiency of the algorithms. Within the thesis, we identified and introduced solutions to areas we felt had the most significant possible impact, such as sample efficiency, planning, and task transfer.

By improving sample efficiency, the amount of information the algorithms need before performing well goes down. Our proposed algorithm does this by reducing the amount of information thrown away by the algorithms. In planning, where the algorithm can think ahead of picking an action, this thesis proposes making another part of the algorithm learnable by the computer, which we show helps improve its performance. Finally, in task transfer, where an algorithm has finished learning and is moved to another related problem, we propose two contributions: one improves the expressiveness of the algorithm while the other breaks it into two simpler problems that are easily solved.

Acknowledgements

Não morra na praia, you are almost there.

First I would like to thank Dr. Miriam Capretz, who has been immensely supportive and extremely patient with me. Without her, I most likely would not have pursued graduate school let alone be completing a Ph.D. I sincerely appreciate the freedom you gave me during my years in the program that allowed me to chase all these exciting questions I felt needed an answer. *Não morra na praia*, will also be something that I echo internally to myself, thank you for everything.

To my father, Dr. Louis Tasfi, and my mother, Agnes: for the boundless love, support, and patience you've had for me over all the years; even when I was uncertain about my path forward in life. I thank my sister Sydney, for also being supportive throughout.

Meu amor, Lorena: without your love and support, I do not know if this degree would have been possible, you were always there to support me through all the highs and lows no matter the day or time; offering words of encouragement and doses of reality when I needed them most.

Thank you Dr. Eder Santana for your friendship, guidance, and support from my journey through Silicon Valley to the end of my degree.

Justin Tomasi, thank you for your friendship throughout all the years we've known each other, the many discussions we had, and providing well timed distractions when I needed them most.

I am grateful for the support and guidance from Dr. Katarina Grolinger and Dr. Hany El Yamany during my undergrad, when I was tackling my first piece of academic research.

Dr. Rui Wu, thank you for patiently explaining the ins and outs of academia, supporting me through my many endeavours into finance, and offering your guidance to me – I will cherish all of the long discussions we have had.

My deepest thanks to Dr. Luisa Liboni, who I unfortunately only met toward at the end of my Ph.D, your help and support over the final hurdles of the program was deeply appreciated.

Thanks to all my friends and colleagues I had met during my time in the program for your support, the laughs we had, and engaging conversations: Ljubisa Sehovac, Jose Miguel,

Willamos Aguiar, Rafael Aguiar, Yifang Tian, Navid Fekri, Dr. Wander Queiroz, Dr. Alex L'Heureux, and Dr. Wilson Higashino.

Dr. Walter Vannini, who over the span of a very long walk through Palo Alto, patiently explained the importance of a Ph.D to a stubborn young man.

Finally, all the new friends I have made over the past few years who made the craziness of the world much more tolerable; Thank you Jeff, Stephanie, Sean, Sam, Chris B., Jae, Peter, N., Chris S., Dave, Matt, Nem, and L. Up only, my friends.

Contents

List of Figures

List of Tables

List of Acronyms

A2C Advantage Actor Critic

A3C Asynchronous Advantage Actor Critic

ACER Actor Critic with Experience Replay

BFS Breadth First Search

BPTT Back Propagation Through Time

DFS Depth First Search

DL Deep Learning

DNN Deep Neural Networks

DPN Dynamic Planning Networks

DQN Deep Q-Learning

DRL Deep Reinforcement Learning

DynSF Dynamic Successor Features

EA Evolutionary Algorithm

GAI General Artificial Intelligence

I2A Imagination Augmented Agents

IS Importance Sampling

KL Kullback–Leibler

LSTM Long Short-Term Memory

MAML Model Agnostic Meta-Learner

MDP Markov Decision Process

ML Machine Learning

MPC Model Predictive Control

NISAC Noisy Importance Sampling Actor Critic

OC Option-Critic

RL Reinforcement Learning

RNN Recurrent Neural Network

S2F Second-Order Successor Features

SF Successor Features

SMDP Semi-Markov Decision Process

SOTA State Of The Art

TD Temporal Difference

TRPO Trust Region Policy Optimization

VIN Value Iteration Network

VISR Variational Intrinsic Successor FeatuRes

VPN Value Prediction Network

t-IS Truncated-Importance Sampling

Chapter 1

Introduction

1.1 Introduction & Motivation

Reinforcement Learning (RL) is the study of learning to do, that is, learning how to map situations to actions to maximize an external reward signal [1]. The field has applications across many diverse areas, such as continuous control in robotics [2], architecture search for neural networks [3], and financial markets [4]. Recently, RL has had tremendous advancements in performance primarily fueled by the development of methods that stably integrate non-linear function approximation through deep neural networks (DNN), resulting in a class of algorithms grouped as *Deep Reinforcement Learning* (DRL) [5]. This breakthrough allowed the application to high-dimensional state and action spaces while encouraging the use of exotic model architectures from deep learning. Together, DRL methods have pushed performance in a large swath of domains to superhuman levels with improvements in complex, and large-scale task environments such as Atari [6], Go [7], and DOTA 2 [8]. RL has been described as the most promising avenue towards general-purpose artificial intelligence by luminaries of the field [9]. While DRL algorithms have outperformed human experts across many benchmarks, the current generation of algorithms lacks notable aspects of human performance.

A cursory examination of current DRL algorithms quickly shows that nearly all DRL al-

gorithms are grotesquely inefficient compared to humans and even against Machine Learning (ML) algorithms used in other domains. However, labeling DRL algorithms as inefficient ignores essential nuances that can be categorized to better understand the problem at hand. While not exhaustive, one such relevant categorization would include action efficiency, sample efficiency, and task transfer. An algorithm with greater efficiency across such a categorization will perform better than one without. Indeed, greater efficiency causes naturally tighter iterative loops during research and decreases compute requirements if a model converges faster.

By observing any state-of-the-art (SOTA) RL model interacting with its environment, the apparent errors in action choice and constant flip-flopping are easily seen. This inefficiency in action selection can cause sub-optimal performance. Terming this as action efficiency, with no analogous metric to other ML algorithms, it measures an RL model's effectiveness in selecting the minimal number of error-free actions for which the desired output is produced. Action efficiency is not strictly concerned with the direct reward produced by a series of actions by an RL model; it adds an implicit cost to the calculus of this efficiency. The subtle difference can be made more evident if we consider a simple scenario where two RL models are optimized to solve a multi-step puzzle task. Suppose both solve the task, but one does so in fewer bouts of trial-and-error. In that case, we would naturally conclude that the RL model with the minimal number of actions with fewer errors has a higher *action efficiency*. This also extends to many actions selected along longer horizons where a model must begin to avoid systematic errors in action selection, not just the next action choice of the model.

Naturally, adding planning ability to an RL model could help improve its action efficiency. Indeed planning, the refinement of the following action choice taken in the environment through some well-defined and repeatable process, exactly fits our problem description. The current landscape of planning architectures in RL uses a predefined planning algorithm heavily relying on a learned state-transition model [10, 11, 12, 13]. However, this is at odds with a reoccurring lesson within ML: it is superior to learn end-to-end instead of engineering knowledge.

While deep neural networks learn the features end-to-end, they require immense amounts of data. The combination with the RL algorithm further compounds the data requirements, as current RL algorithms have poor sample efficiency, magnifying the total combined requirements further by a magnitude more. Here, the measure of sample efficiency, the efficiency with which an algorithm can leverage each data sample, is a vital axis to improve. Indeed, if an algorithm has high sample efficiency, they need significantly fewer samples to reach a reasonable level of performance. Comparatively, high sample efficiency can be seen with many supervised learning algorithms, which are significantly more efficient on a per-sample basis. If comparing two RL algorithms on the same task, the algorithm that learns to master the task quicker is said to have greater sample efficiency. Different classes of RL algorithms will have better sample efficiency than others, such as off-policy methods versus on-policy methods. However, off-policy methods are not without their flaws. A vein of research looked to combine the types to reap the benefits of both [14, 15, 16, 17]. Current cutting-edge implementations achieve this but are incredibly complicated models with many moving parts [17]. Ideally, a simpler variant could capture most of the performance.

Task transfer is an intertwined aspect of sample efficiency, as it involves how easily a converged model can transfer previously gained knowledge to another related task. In supervised learning, fine-tuning is a standard methodology to transfer trained models between related datasets [18]. Fine-tuning involves drastically reducing the model's learning rate while learning on the new but related data. The intimate connection between sample efficiency and task transfer of a model can be made clear. If a model can transfer performance between related tasks with few additional samples, then would be required in collecting an entirely new set, it can be said to have higher sample efficiency. In the longer term view of RL as a general solution to AI, having strong task transfer ability is vital. The RL model should quickly adapt and adjust to changing environmental conditions and perform well on related tasks. However, transferring a learned policy of an RL agent between tasks using a similar technique, even within the same environment, ranges from difficult to impossible for many algorithms. As a

result, the RL algorithm often needs to be fully retrained on the new task requiring many new samples from the environment. Ideally, the RL agent could leverage previously learned knowledge about the environment or previous task structures to accelerate the transfer to a newly presented task.

One avenue of research, Successor Features (SF), aims to accomplish task transfer by rephrasing the RL problem as an environment dynamics model and a reward component [19, 20]. During transfer, only the much smaller reward component is trained, making learning faster while reusing previously learned information about the environment. However, the assumptions required to separate the algorithm into separate components can limit and hinder the final model's performance.

1.2 Contributions

This research provides improvements along the following categorizations of efficiency: action efficiency, sample efficiency, and task transfer. Within each category, we contribute an algorithm that improves over the previous state-of-the-art methods, along with metrics important in the specific domain. In the action efficiency domain, which we refer to as planning from here on in, a novel algorithm is presented that can dynamically choose its planning algorithm depending on the specific task demands. This line of work follows from the lessons learned within deep learning, in that learning the features, in our case planning style, is superior to manually-created features.

An algorithm was developed within the sample efficiency category to optimize an on-policy algorithm with off-policy samples using a novel application of the Gumbel noise distribution for sample weighting.

Finally, our last area of focus was task transfer, improving on a specific framework. Within task transfer, we present two novel algorithms, a second-order variant, and reformulation that dynamically creates acting policies on the fly. With the second-order variant, we present a

mathematical reformulation from first principles that yield a more robust reward model with improved performance. Our dynamic variant takes a different mathematical interpretation of the base framework. It allows us to use a purely supervised approach to the RL problem with increased flexibility while being mathematically equivalent. Therefore, the contributions of this research can be summarized by area as follows:

Planning

- Dynamic Planning Network (DPN): a planning architecture that creates plans with a learned dynamic planning style.
- We show that providing a planner with the option to choose *where* to plan from improves performance by reducing sub-optimal trajectories.
- A loss function for the planner policy that balances exploration and exploitation during planning.
- DPN outperforms other planning architectures on commonly used environments in the domain, both in performance and sample efficiency.

Sample Efficiency

- We introduce Noisy Importance Sampling Actor-Critic (NISAC), an algorithm that can reuse previously seen data efficiently.
- Experimentally proves NISAC outperforms current strong baseline methods in performance and sample efficiency.
- NISAC training time is 40% faster than baseline methods while being significantly easier to implement.

Task Transfer: Second-Order Successor Features

- A novel formulation of the SF framework that uses a second-order reward function. This formulation increases the representational power of the reward function while decreasing the representational load on the state encoder, providing stronger performance guarantees.

- Under the new reward formulation, the extra term that appears was shown to model the future expected auto-correlation matrix of the state features.

- We provide preliminary results that show the second term can be used for guided exploration during transfer instead of relying on ϵ-greedy exploration.

Task Transfer: Dynamic Successor Features

- Dynamic Successor Features (DynSF): An algorithm that enables a state-transition model to be used for state rollouts where the discount factor and policy can be set on the fly, which is advantageous compared to common model-based approaches.

- An examination of how rollout length and policy choice affect the performance of DynSF compared to the original framework and baselines.

- Through experiments, the flexibility of DynSF is analyzed.

- Evidence is provided that DynSF performs better during task transfer than baseline methods.

1.3 Thesis Organization

This thesis is organized as follows:

- **Chapter 2**: Here, the essential background concepts are presented. A general introduction to DL and RL is given, with a presentation of the underlying mechanics and assumptions in the field. Finally, a brief overview of the foundational algorithms is presented.

- **Chapter 3**: This chapter provides an extensive literature review of the studies pertinent to the presented research and discusses avenues for improvement.

- **Chapter 4**: Here, the first piece of our work is presented and focuses on improving sample efficiency. NISAC. NISAC is a fully *off-policy* actor-critic algorithm that learns from stored off-policy trajectories. The model, experimental design, experimental results, and analysis are presented in the chapter.

- **Chapter 5**: This chapter introduces DPN, a novel architecture for DRL that combines model-based and model-free aspects for online planning. Our architecture learns to construct plans dynamically using a learned state-transition model by selecting and traversing between simulated states and actions to maximize information before acting. The model, experimental design, experimental results, and analysis are presented in the chapter.

- **Chapter 6**: This chapter presents a task transfer algorithm: Second-Order Successor Features (S2F). S2F is a novel extension to the SF framework, where the rewards are modeled with a second-order function. The model, experimental design, experimental results, and analysis are presented in the chapter.

- **Chapter 7**: The final presented work, Dynamic Successor Features (DynSF), reformulates the SF framework as two supervised learning problems. This reformulation dramatically improves the model's flexibility while providing evidence of what the original formulation learns. The model, experimental design, experimental results, and analysis are presented in the chapter.

- **Chapter 8**: Finally, this chapter discusses the contributions brought by the presented research and directions for future work.

Chapter 2

Background

This chapter covers the essential background material of the work presented in this thesis. The first portion discusses deep learning, while the second half focuses on RL concepts.

2.1 Deep Learning

DL models are large artificial neural networks, often containing many layers. The neural network component can easily be described as a directed acyclical graph, which many will recognize as a multi-layer perception network. These networks typically take input in the form of vectors and sequentially process them through each of the layers. A network is a strong function approximator, able to represent arbitrary functions [21, 22, 23]. DL can be seen as an improvement on existing methods found within machine learning, which is mostly concerned with learning a well-defined function from observed data. The primary improvement is loosening the requirements for hand engineering of features derived from the dataset, which it instead learns automatically and for the task at hand [18]. Indeed, the general recipe required to apply deep learning to any problem, often without the need for domain knowledge, involves a deep network acting as an expressive function approximator, a task-specific loss function, and a way to optimize the parameters of said function with gradient descent. This has proven to work extremely well, especially with the tremendous amount of compute and data available today, as

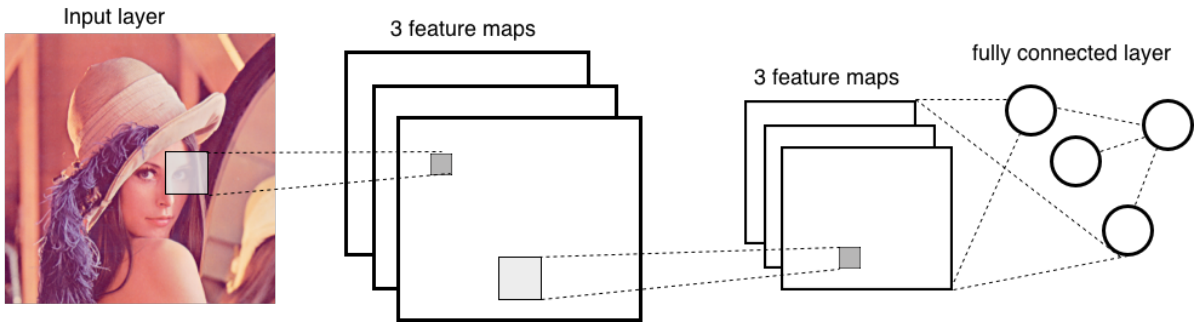


Figure 2.1: Convolutional network. Earlier convolutional layers extract features and build a feature hierarchy. The final layers are fully-connected units. Adapted from: http://deeplearning.net/tutorial/lenet.html

the same recipe has produced state-of-the-art results across language modeling [24] and image generation [25].

While DL algorithms are widely applicable to different problems, often with little modification, the data's medium has an important consideration on the architecture of the deep learning model. Two special model architectures are considered in the following subsections: convolutional and recurrent.

2.1.1 Convolutional Networks

Convolutional Neural Networks [26] (CNNs) are a specialized network architecture for processing data with a spatial component, such as images or time-series data. CNNs have had immense success within computer vision applications [27, 28, 29] and are often considered a hard requirement in any modern architecture that processes visual information. Convolutional networks build upon a specialized linear mathematical operation called *convolution* [18] between two matrices.

Fundamentally, a CNN learns many small matrices, referred to as feature maps, that capture specific features on the input space and use the convolution operation to measure the "similarity." The same feature map is applied over the entire input space with fixed step size shifts, repeating over several layers. The output of a convolutional layer, the result of the convolution operation with the feature map and input, is one activation map per learned feature map. Over

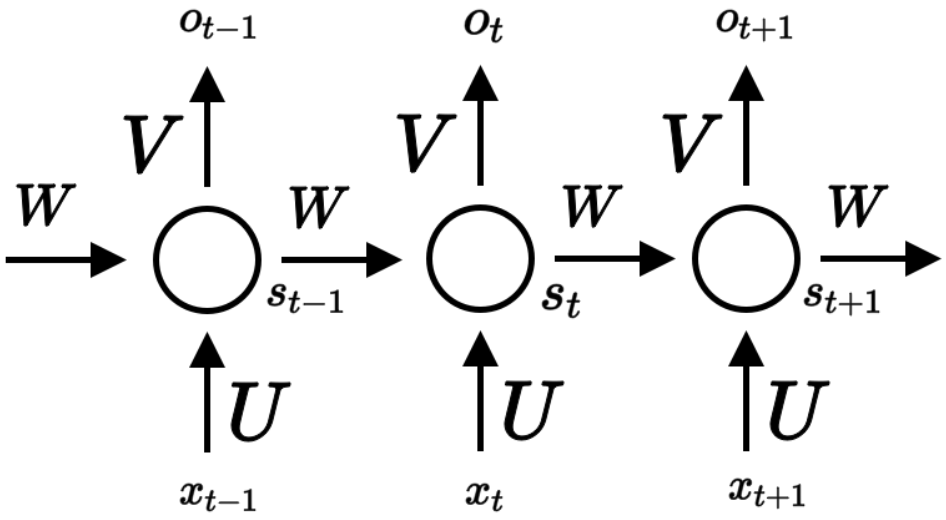


Figure 2.2: A recurrent neural network with one hidden unit unrolled over time. Adapted from: [30, LeCun, Bengio, and Hinton, 2015; Figure 5]

many convolutional layers, the network learns a hierarchy of features that build upon the lower levels. An example would be that earlier layers learn edges or textures while later layers learn high-level concepts such as eyes.

The learned features are used for object reasoning in later fully connected layers, as shown in Figure 2.1. Doing so allows the features to be automatically optimized based on the task at hand, reducing the requirement for hand engineering.

2.1.2 Recurrent Networks

Recurrent Neural Networks [31] (RNNs) are a family of networks used for processing data with a sequential component and have had a tremendous impact in fields such as speech recognition and machine translation [32, 33]. Similar to convolutional networks, in that they exploit a unique axis of the data, RNNs process a sequence of values and use some notion of the input order to inform output decisions. The standard approach uses an RNN to process one element of a sequence at each time step; the RNN uses some notion of internal hidden memory to incorporate the sequence information for later steps. In doing so, the RNN can recall information from previous time steps for use in its output. Mathematically, with biases omitted, a simple

model with a single hidden unit can be expressed by the following equations:

$$\bar{x}_t = \mathbf{U}x_t \tag{2.1}$$

$$s_t = \tanh(\mathbf{W}s_{t-1} + \bar{x}_t) \tag{2.2}$$

$$o_t = Vs_t \tag{2.3}$$

where $\{U, W, V\}$ are the set of learnable parameters of the model, $\bar{x}_t$ is the transformed input at time t, s_t is the new hidden state at time t, and o_t is the output of the network. This process is illustrated in Figure 2.2, where an RNN with one hidden unit consumes a sequence of data x_{t-1}, x_t, x_{t+1} at each timestep. In the second step t, we see the RNN updates the previous internal memory s_{t-1} with the new information contained in x_t, outputs a value o_t, and produces the new state of its internal memory. While the above model scales well to many additional hidden units, which increases the memory available to the model, if the information contained within the data was seen too much earlier or the sequence data has a high dimension, the RNN will have issues remembering and incorporating this information. This issue is resolved by introducing another variant of a recurrent network known as a Long-short Term Memory (LSTM) network [34] that uses an internal memory structure better tuned for remembering long-term dependencies. An LSTM introduces a gating mechanism that allows the network to learn how long it should hold on to old memory, when to forget, when to incorporate new information, and how to mix old and new information. Again, for a single hidden unit with biases omitted, the model can be expressed mathematically as:

$$f_t = \sigma(W_f \cdot [s_{t-1}, x_t]) \tag{2.4}$$

$$i_t = \sigma(W_i \cdot [s_{t-1}, x_t]) \tag{2.5}$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [s_{t-1}, x_t]\right) \tag{2.6}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.7}$$

$$o_t = \sigma(W_o[s_{t-1}, x_t]) \tag{2.8}$$

$$s_t = o_t * \tanh(C_t) \tag{2.9}$$

where additional learning parameters $\{W_f, W_i, W_C, W_o\}$ are introduced to control the forget, input, cell, and output gates, respectively. σ is the sigmoid function and $*$ is the hadamard product. The LSTM also introduces the notation of an additional memory referred to as the "cell state", denoted by C_t, and its proposal update $\tilde{C}_t$. The gates output a value between [0, 1], which attenuate the input based on the context of the last hidden state s_{t-1} and current input x_t. By doing so, the LSTM can track many longer-term dependencies than a plain RNN. Both the RNN and LSTM are trained using a backpropagation through time (BPTT) algorithm that allows gradients to flow backgrounds through time [35].

2.2 Deep Reinforcement Learning

RL is a subarea within ML concerned with finding the optimal series of decisions to maximize an external signal. RL involves an agent acting within an environment such that at each timestep, the agent takes action, observes a new state, and receives a reward. An RL algorithm aims to maximize the series of received rewards within an unknown environment [1]. The field has applications across many diverse areas, such as continuous control in robotics [2] and financial markets [4].

DRL, the extension to RL, is the study of using RL algorithms with DNNs acting as function approximators. The application of DL to RL allowed the field to move away from using hand-crafted linear features or tabular function approximators. It enabled algorithms to learn features required for the task at hand automatically from data. This led to an explosion of breakthroughs, from Mnih *et al.* [6], who demonstrated an agent capable of playing through assorted games from the Atari collection from pixels using the Q-learning algorithm, to more

recent work by Silver *et al.* [36] where an agent mastered the game of Go beating out human experts.

2.2.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical description of an RL agent interacting with a stochastic environment. An MDP consists of the following components: a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a transition probability function $P : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to [0, 1])$, and a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ [37]. Throughout the literature, each of these components has had various definitions, which allow immense flexibility in how the specific task at hand is modeled. For example, the reward function could be expressed as a deterministic function $r(s)$ on the state or state and action $r(s, a)$. The end goal is to find a policy π which maps states to actions. A *policy* can either be a probability distribution over actions conditioned on states, $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ or a mapping of the expected value of an action in a specific state, $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. An agent acting within an MDP is executed as follows: at each time step t, the agent observes the environment state $s_t \in \mathcal{S}$, chooses an action $a_t \in \mathcal{A}$ from its policy $\pi(a_t|s_t)$ acting with it, and finally a new state s_{t+1} and reward r_t are sampled from transition function [37]. In this work, we are interested in the episodic reinforcement learning problem, where the agent's experience is broken into a series of "episodes". An episode can be an arbitrarily long but finite sequence of states, actions, and rewards. In this case, the previous definition of the MDP changes slightly in that the episode ends when the transition probability function emits a terminal state s_T. In this case, the ,agent's goal is to maximize the expected total reward of the episode, that is $R_t = \sum_{t=0}^{T} r_t$.

2.2.2 Policies

Within this work, the definition of a policy π changes based on the particular learning algorithm used, where it can either be a stochastic policy written as $\pi(a|s; \theta)$ or deterministic as $\pi(a, s; \theta)$. In both cases, the policy is parameterized by a set of parameters represented by θ, with θ meant

to capture all the parameters used in the model at hand. The type of policy used will have implicit assumptions to the activation and inference method. A stochastic policy will always use a softmax function as we must output a valid probability distribution over all available actions; the actions will be sampled by assuming its output forms a categorical distribution. While the deterministic policy can either be a straight linear output, that is, with no activation function, or one using some form of clipping.

2.2.3 Temporal Difference Learning

Temporal-Difference is a model-free algorithm that learns from tuples of experience, meaning completed episodes are not required. TD methods rely on the bootstrapping method, where targets are updated from existing estimates instead of a ground truth value. The central idea of TD learning is to update a value function, such as the state-value function $V(s_t;\theta)$ parameterized by θ, towards a TD target, $r_{t+1} + \gamma V(s_{t+1};\theta)$, which is estimated from the current set of parameters θ. The updates are controlled by a learning rate hyperparameter α:

$$V(s_t;\theta_{t+1}) = V(s_t;\theta_t) + \alpha(r_{t+1} + \gamma V(s_{t+1};\theta_t) - V(s_t;\theta)) \tag{2.10}$$

Where γ is the discount factor. Another commonly used value function is the state-action value that estimates the value of the future state given a specific action with the update given as:

$$Q(s_t,a_t;\theta_{t+1}) = Q(s_t,a_t;\theta_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1},a_t^*;\theta_t) - Q(s_t,a_t;\theta_t)) \tag{2.11}$$

The state-action value function can easily be adapted for control, as shown by Watkins & Dayan [38], which proposes the Q-Learning algorithm. The Q-Learning algorithm makes critical changes to how the state-action value function is used for policy evaluation and improvement. During policy evaluation of the state-action value function, the action is selected by taking the largest Q value such that $a_t^* = \arg\max_{a\in\mathcal{A}} Q(s_t,a)$ and applying an exploration strategy to the output, such as ϵ-greedy. With the collected data from the interaction, that is,

the observed state s_{t+1} and reward r_{t+1}, the update for the state-action value function becomes:

$$Q(s_t, a_t; \theta_{t+1}) = Q(s_t, a_t; \theta_t) + \alpha(r_{t+1} + \gamma \arg\max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta_t) - Q(s_t, a; \theta_t)) \tag{2.12}$$

Q-Learning has many different implementations available, such as a tabular lookup table for a particular state and action. However, as the action and state space grow in dimensionality, this becomes intractable, so the application of function approximation techniques must be used, such as DNNs offer. This leads naturally to work proposed by Mnih *et al.* [6] in Deep Q-Learning Networks (DQN), where the state action-value function is approximated with a CNN and a set of fully connected layers. However, simply applying a DNN to control with a state action function has significant issues with stability, leading to the "deadly triad" as coined by Sutton [39], where bootstrapping, off-policy learning and function approximation causes immense problems with stability during learning and divergences in policy performance. To this end, Mnih *et al.* [6] propose the use of three innovations: experience replay, periodical target updates, and gradient clipping. Experience replay accumulates the necessary information for a single Q-Learning update over many environment interactions in a large data store and randomly samples during learning. This drastically improves the sample efficiency and removes correlations in the observation sequences. As the state action function is updated towards a bootstrapped target value, which can vary wildly between updates, DQN proposes using a target that has slow and periodically updated weights to increase stability and reduce short-term oscillations. Finally, DQN uses gradient clipping in its updates to avoid significant and harmful changes to its gradients that could induce stability issues.

2.2.4 Policy Gradient Methods

In RL, the direct optimization of a stochastic policy, with parameters θ, requires using the policy gradient theorem [40]. The policy gradient theorem provides an expression for the gradient of the discounted reward objectives with respect to the parameter θ. Therefore, the

parameters θ of the differentiable stochastic policy $\pi_\theta(a_t|s_t)$ are updated using the following gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\Psi^\pi(s_t, a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{2.13}$$

Where $\Psi^\pi(s_t, a_t)$, as shown by Schulman *et al.* [2], can be replaced with quantities such as the total reward of the trajectory, the temporal difference residual, or the state-action value function $Q^\pi(s_t, a_t)$. The choice of Ψ^π affects the variance of the estimated gradient. A common advantage function $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$, provides a relative measure of value for each action and has been used extensively throughout the literature [41]. The advantage function helps reduce the gradient estimator's variance while keeping the bias unchanged. The Asynchronous Advantage Actor-Critic (A3C) algorithm [41] uses policy gradients with an advantage function with many parallel actors, all trained in parallel, each interacting with a copy of the environment. The parameters of each actor are synced periodically with the global parameters. This greatly speeds up learning and allows agents to converge quickly. A2C, the synchronous and deterministic version, where multiple actors interact with their environment for a fixed number of steps and then perform a global parameter update.

2.2.5 On- & Off-Policy Algorithms

A common classification for RL algorithms is whether they are on- or off-policy, where the distinction is based on the behavior and updated policies of the agent [1]. Where the classification hints at how the algorithms are updated and collect information. On-policy algorithms update their parameters with the same policy that interacted with the environment; that is, their behavior and update policies must match, such as algorithms that used policy gradients [2]. Off-policy algorithms, such as Q-Learning [39], allow the behavior and update policies to be different. Off-policy algorithms are often much more sample efficient as they can reuse all the data from the environment, even with older or different behavior policies, than what the agent currently posses [17].

2.3 Evolutionary Algorithms

Evolutionary Algorithms (EA) are a class of population based stochastic search algorithms that draw inspiration from biological evolution; using terminology common to the field such as population selection, fitness scoring, and offspring[42]. In general, an EA algorithm takes a population of learners and repeatedly prunes the population according to a fitness function and expands the remainder by recombination (offspring). A generation is produced from one iteration of this process and repeats until a certain number of generations has been reached or a stopping criterion is met. The fitness function is usually manually specified and defines how each member of the population is performing and if it needs to be pruned. While performant on tasks with small state spaces, EAs do not scale well with complexity, that is when the number of parameters being optimized over increases. Within this thesis, the tasks we explore require immense numbers of parameters, often number in the tens of millions, which is intractable with current EA methods. Therefore in this work, we use stochastic gradient descent as the optimization algorithm of choice.

2.4 Summary

This chapter has presented a background review of important core concepts that were foundational to the contributions made within this thesis. The first section has provided a description and definitions for core concepts within DL. The second section provided a review of DRL, MDP, core concepts, and three foundational learning methodologies: TD learning, policy gradient methods, and evolutionary methods. In the next chapter a review of the relevant academic literature is given.

Chapter 3

Literature Review

This chapter provides a literature review of the foundational and relevant methods within DRL. The review is organized around the categories focused on by this thesis: sample efficiency, planning, and task transfer.

3.1 Sample Efficiency

Despite its many successes, a noticeable limitation of current RL methods is the low sample efficiency, requiring a considerable number of samples from the environment to converge to an appropriate solution. Sample efficiency has improved from different paths, such as exploration methods, improved environment modeling, or importance sample. The work within this thesis focuses on the last pathway, which is importance sample, as we felt it had the most significant impact via its general applicability and had the most straightforward path forward. However, to be thorough, we provide a literature review of each of the aforementioned avenues.

Exploration Methods

Exploration methods adjust how the agent explores the environment, with basic methods including random perturbation of the action space with methods like ϵ-greedy. A better explo-

ration method improves the variety of the data and, therefore, the quality of the policy such that it generalizes better and will, on average, require fewer samples. However, these methods often rely on adjusting the output of the agent's action choice, which can be quite far from the current policy. Parameter space noise has been a proposed exploration method that injects randomness into the model via the parameter space. The intuition is that the randomness induced in the action choice is closer to the current agent policy [43]. Indeed, Rückstiess *et al.* [44] show that this is possible and has several advantages over action space noise in continuous control tasks. More recently, Plappert *et al.* [45] demonstrate that parameter space noise has greater efficiency than action space noise in some scenarios across many standard algorithms in DRL. Fortunato *et al.* [46] proposed a method that strikes a balance between the parameter and action space noise with their *NoisyNet*, where noise is added to the parameters immediately preceding the output of the agents.

Environment Modeling

An axis that DRL algorithms can be classified along is if it incorporates a model of the environment, that is *model-based* or lacks one, implying it is *model-free*. An environment model includes a transition function that, given an action and current state, can predict the next state the environment transitions. Learning these transition models is a field on its own with many variations, such as those based upon RNNs [47] or conditional feed-forward models [13]. Once a learned environment model is available, the agent can learn from generated samples instead of relying on expensive environment interactions. A variant of the Dyna model [48] proposed by Holland *et al.* [49] does precisely this by learning an environment model to generate experience for policy training in the Atari suite. In the low data regime, Kaiser *et al.* [50] found that generated samples from a learned environment model can improve the performance by order of magnitude on some game tasks. The environment model can also selectively model portions of the environment, as shown by Oh *et al.* [13], who use a model of rewards to augment a model-free agent to improve learning with solid results on many games in the Atari suite.

Variations on the environment models architecture also exist, with Ha & Schmidhuber [51] using a variations autoencoder with an RNN to successfully simulate and evaluate the Doom environment.

Importance Sampling

Important sampling is a widely used method that enables the usage of samples generated by one policy to be used by another for learning. This is important, as many RL algorithms require parameter updates from data generated by the same policy. This is referred to as an *on-policy* algorithm. In contrast, an *off-policy* algorithm can use mismatched policies [52]. Intuitively, the off-policy algorithm will have a higher sample efficiency as any generated sample from an older version of the current policy, or a completely different policy can be used [39]. As an on-policy algorithm must have exact parity between the policies, even the samples generated one update step apart cannot be used, implying previous data cannot be kept and must be discarded. An on-policy algorithm can be modified with importance sampling to use samples generated by other policies, making it *off-policy*.

Therefore, a highly performant on-policy method, such as actor-critic, could use *off-policy* samples within its learning update through the usage of *importance sampling* [14, 15, 53, 54]. The actor-critic method uses the policy gradient method to compute the direction of its parameter updates [41]. In practice, the policy gradient is estimated from a trajectory of samples generated by the *on-policy* stationary distribution $\pi(a|s)$. Given a trajectory of samples generated by some behaviour policy $\mathcal{B}(a|s)$, the policy gradient from Equation 2.13 is modified to be:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{B}}\Big[\rho_t \Psi^\pi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\Big] \tag{3.1}$$

where ρ is known as the *importance weight* and is defined as a ratio between the current policy $\pi(a|s)$ and the behaviour policy $\mathcal{B}(a|s)$ as $\rho_t = \frac{\pi(a_t|s_t)}{\mathcal{B}(a_t|s_t)}$. Unfortunately, the importance-weighted gradient in Equation 3.1 suffers from high variance. To reduce variance, Wawrzyński [16] pro-

posed truncating each importance weight to the interval $[0, c]$ where c is some constant.

Wang *et al.* [17] proposed Actor-critic with experience replay (ACER), an off-policy actor-critic algorithm that uses experience replay. ACER builds upon the on-policy A3C algorithm [41]. ACER proposes three changes to A3C to convert it to an off-policy method: truncated importance sampling with bias correction, retrace Q-value estimation, and updates performed with Trust Region Policy Optimization (TRPO) [55]. As the gradients from importance sampling-based updates can suffer from high variance, the quantity is usually truncated by a constant c as done by Wawrzyński [16]. However, this truncation introduces a bias, so Wang *et al.* [17] propose a correction term. Therefore, the gradient for ACER is defined as:

$$\begin{aligned}\nabla_\theta J(\theta) = \min(c, \rho_t)\{Q^{ret}(s_t, a_t) - V_w(s_t)\}\nabla_\theta \ln \pi_\theta(a_t|s_t) \\ + \mathbb{E}_{a\sim\pi}[\max(0, \frac{\rho_t - c}{\rho_t})(Q_w(s_t, a) - V_w(s_t)\nabla_\theta \ln \pi_\theta(a|s_t)]\end{aligned} \tag{3.2}$$

where Q_w and V_w are value functions predicted by the networks critic with parameters w and Q^{ret} is the estimated Q-value given by Retrace [56]. Retrace is a return-based Q-value estimation algorithm that can be used with off-policy data. The first term of Equation 3.2 is the truncated importance weight which reduces variance. The second term corrects the bias introduced by the first term such that ACER's policy gradient is an unbiased estimation. ACER showed a big improvement in sample efficiency over the A3C algorithm while being close to the sample efficiency of off-policy algorithms.

3.2 Planning

In RL, the archetype structure in planning typically involves an agent that chooses actions given some environment state and a planning module which contains a learned environment model [57]. The environment model often referred to as a state-transition model in the literature, learns the environment dynamics and is used to predict the next state s_{t+1} given the current

state s_t and current action a_t. Between action selection steps in the environment, through repeated application of the state-transition model and agent policy, it is possible to roll forward a simulated prediction, or plan, of likely paths forward. The repeated application involves feeding in the current state s_t, choosing an action selected by the agent given this state $a_t \sim \pi(a_t|s_t)$, and passing this to the state-transition model, which predicts the next state s_{t+1} – which is then fed back in on the next iteration. Using this internal environment model, the agent can reason about the future, seek positive outcomes, and avoid consequences of trial-and-error in the environment which can often result in making irreversible action choices [11]. The choice of state-transition architecture is highly flexible with successful applications of an encoder-decoder network [58], recurrent networks [13], or residual networks [59]. Further, the output of the state-transition model is also open, with some works predicting the entire state s_t [58] or a latent representation [10], often denoted by ϕ_t.

Modern Planning

State-transition models have been used to improve the sample efficiency in various ways, such as using the information gained from rollouts as inputs to the policy [11], training the model on samples generated from the planner [60, 48], or by aiding the agents learning by improving policy evaluation [61]. Planning is strongly related to improving sample efficiency, as the learned state-transition model gives the agent additional samples to learn from [60] or avoid costly environmental trial and error [11]. Indeed, model-based algorithms such as PILCO by Deisenroth *et al.* [62], which reduces model bias through explicitly incorporating model uncertainty, have shown that it is possible to learn from orders-of-magnitude fewer samples. Various efforts along this avenue combine model-free, and model-based methods, such as the *Dyna-Q* algorithm [48] that learns a model of the environment and uses this model to train a model-free policy. Initially applied in the discrete setting, Gu *et al.* [63] extended Dyna-Q to continuous control.

Pascanu *et al.* [64] implemented a model-based architecture comprised of several individu-

ally trained components that learn to construct and execute plans and use a separate specialized policy. This is problematic as the policy used during planning and acting in the environment have no guarantee of being similar, potentially causing disagreement, such that the planning policy could explore or exploit a plan for which the acting policy cannot traverse. Vezhnevets *et al.* [65] proposed a method that learns to initialize and update a plan; their work does not use a state-transition model and maps new observations to plan updates. Guez *et al.* [66] proposed MCTSnets, an approach for learning to search where they replicate the process used by MCTS. MCTSnets replace the traditional MCTS components with neural network analogs. The modified procedure evaluates, expands, and back-ups a vector embedding instead of a scalar value. The entire architecture is end-to-end differentiable.

Sequential Planners

Value prediction networks (VPNs) by Oh *et al.* [67], Predictron by Silver *et al.* [61], and *ATreeC* by Farquhar *et al.* [10], an expansion of VPNs, combine learning and planning by training deep networks to plan through iterative rollouts. The Predictron predicts values by learning an abstract state-transition function that is used to improve the evaluation of the policy instead of directly for planning. Oh *et al.* [67] and Farquhar *et al.* [10] both construct trees to improve value estimates using forward-only rollouts, exhaustively expanding each state's actions. In addition, Farquhar *et al.* [10] use the tree for training and acting. Farquhar proposed two variants: ATreeC, building upon the actor-critic method, and TreeQN, building upon the Deep Q-Learning method [6]. Both methods exhaustively expand out all states and actions and then pick the pathway with the highest value and, therefore, the following action. However, the expansion along all actions across the state space for a fixed depth is computationally expensive and lacks the flexibility to adjust based on the particular state. Similarly, Francois-Lavet *et al.* [68] proposed a model that combined model-free and model-based components to plan on embedded state representations similar to TreeQN [10]. Learning on embedded state representations eases the computational requirements of both the state-transition models and

agent as it does not need to process high dimensional inputs.

Similarly, Weber *et al.* [11] proposed Imagination Augmented Agents (I2As), an architecture that learns to plan using a separately trained state-transition model. Planning is accomplished by expanding all available actions $\mathcal{A}$ of the initial state and then performing $\mathcal{A}$ rollouts using a tied policy for a fixed number of steps. The authors claim that the model learns to select pertinent information from the series of rollouts via an aggregator to inform its decision; as state-transition models suffer from compounding errors due to prediction errors, the aggregator also serves as an interpreter. We argue this is an improvement on the work of Farquhar *et al.* [10] and Oh *et al.* [67] as the planning step only forces breadth-first expansion on the first step and then allows the free selection of actions according to a planning policy. However, I2As uses a separate policy for planning step rollouts but links them together via an auxiliary entropy loss function which helps but still does not guarantee similarity. Further, in terms of sample efficiency, I2As require hundreds of millions of steps to converge, with the Sokoban environment taking roughly 800 million steps.

Within continuous control learning, a state-transition model for planning has been used in various ways. Finn *et al.* [69] demonstrated using a predictive model of raw sensory observations with model-predictive control (MPC), where the model is learned entirely self-supervised. Srinivas *et al.* [70] proposed using an embedded differential network that performs iterative planning through gradient descent over actions to reach a specified target goal state within a goal-directed policy. Henaff *et al.* [71] focus on model-based planning in low-dimensional state spaces and extend their method to perform in discrete and continuous action spaces.

Value Iteration Networks (VIN) proposed by Tamar *et al.* [59] is an exciting avenue of work explored planning in an agent without an explicit state-transition model. Instead, VIN uses an explicit differentiable planning structure, implemented with convolutions, to perform approximate on-the-fly value iteration. The essential contribution is using a convolutional network as the backbone of the planning module, which can learn an equivalent approximation to the value-iteration algorithm. However, as the planning module relies on a convolutional network,

the architecture is only suitable for environments where spatial information is informative, such as top-down gridworld maps.

3.3 Task Transfer

Like humans, who do not learn new tasks from a completely blank slate but continually learn to adjust to new tasks, RL agents can significantly improve their sample efficiency by transferring accumulated experience. Indeed, if given a newly presented task, even if similar to a previously mastered task, the current standard practice is to reset the agent's parameters. Many methods have been explored such as transfer of skills [72, 73], adaptively tuned models [74, 75], and disentangling representations [76, 77, 78]. This thesis contributes to representation transfer but provides a literature review of the related efforts.

Options Framework

In skill transfer, Sutton *et al.* [72] define a framework for temporally extended actions, known as the options framework. Introducing options Ω into the MDP, M forms a Semi-Markov Decision Process (SMDP) [72]. The options framework defines a set of parameterized skills that allows temporally extended policies to be learned for specific state spaces. An SMDP has a corresponding value function over options $V_\Omega(s)$ and option-value function $Q(s, \omega)$. The option-value function is defined as:

$$Q(s, \omega) = \sum_a \pi_\omega(a|s) Q(s, \omega, a) \tag{3.3}$$

where $Q(s, \omega, a)$ is the value of each state-option pair (s, ω) and action a. This is defined by the expected return, which depends on the policy over options, the options, and termination functions:

$$Q(s, \omega, a) = \mathbb{E}_{\pi,\Omega,p,r}[\textstyle\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} | s_t = s, \omega_t = \omega, a_t = a] \tag{3.4}$$

Formally, an option $\omega \in \Omega$ is defined as the triplet $(\mathcal{I}_\omega, \pi_\omega, \beta_\omega)$ where $\mathcal{I}_\omega \subseteq \mathcal{S}$ is an initiation set, π_ω is an intra-option policy, and $\beta_\omega : \mathcal{S} \to [0, 1]$ is a termination function. The initiation set controls when a particular option begins execution, the intra-option policy defines the behavior for its duration, and the termination function dictates when it ends. An analogy would be how a human learns contextually relevant skills such as chopping vegetables or riding a bike; they have a clear initiation state, specific interactions, and a natural end.

Bacon *et al.* [73] proposed an option framework extension by deriving a policy gradient-based update for options, resulting in the Option-Critic (OC) architecture. In contrast to the options framework, the OC can learn both the internal policies and termination conditions of options at once. They provide the following gradient-based updates, given the set of intra-option policies $\{\pi_\omega\} \in \Omega$ parameterized by θ and set of termination functions $\{\beta_\omega\} \in \Omega$ parameterized by ϑ:

$$\nabla_\theta V(s_0) = \sum_{s,\omega} \rho(s, \omega | s_0, \omega_0) \sum_a Q(s, \omega, a) \nabla_\theta \pi_\omega(a|s) \tag{3.5}$$

$$\nabla_\vartheta V(s_0) = \sum_{s',\omega} \rho(s', \omega | s_1, \omega_0) \{Q(s', \omega) - V(s_{t+1})\} \nabla_\vartheta \beta_\omega(s') \tag{3.6}$$

where ρ is a discounted weighting of state option pairs from $s_1, \omega_0) : \rho(s, \omega | s_1, \omega_0) = \sum_{t=0}^{\infty} \gamma^t p(s_{t+1} = s, \omega_t = \omega | s_1, \omega_0)$ as defined by Bacon *et al.* [73]. OC assumes the *call-and-return* option execution model, where the option ω is picked according to the policy over options π_Ω, then follows the intra-option policy π_ω until termination, dictated by β_ω, at which point this procedure is repeated.

Meta Learning

Another way to have an RL model adapt is by building the adaptability into the model itself. This is formally known as meta-learning, where the model learns a set of parameters that can be adapted quickly to related tasks. Wang *et al.* [74] proposed Deep Meta-Reinforcement Learning, a generic framework that can adapt rapidly to new tasks. The framework requires three components. First, the backbone of the model must be based around a recurrent network

and trained with a deep RL algorithm, such as A3C [41]. Second, training must be done over a distribution $\mathcal{D}$ of interrelated tasks where a drawn task i is held fixed for a set number of trials T. The network's hidden state is reset each time a new trial is sampled from the training distribution. Furthermore, the network sees two additional inputs: the action selected and the reward received on the previous timestep. This gives the model information on the value of its previous action. Under these conditions, the model learns to implement a fast adapting "meta" algorithm that is completely encapsulated in the hidden state of the recurrent network. This enables it to adapt to unseen tasks without additional gradient updates as the model adapts to the presented task with only updates to its hidden state. This style of meta-learning is powerful but suffers from similar issues around stability common to recurrent networks. Related to this avenue of research, Finn *et al.* [75] proposed a Model-Agnostic Meta-Learner (MAML), which learns a set of parameters that, on average, perform well on a set of tasks and can be quickly updated to specific tasks.

Successor Features

An additional avenue for transfer learning is to decouple the state features of a domain from its reward distribution; this enables transfer between domains as long as the only difference between them is their reward distributions. SF offers this decomposition of the Q-value function and has been mentioned under various names, and interpretations [19, 20, 79, 76]. This decomposition follows from the assumption that the reward function can be approximately represented as a linear combination of learned features $\phi(s;\theta_\phi)$ extracted by a neural network with parameters θ_ϕ and a reward weight vector w. As such, the expected one-step reward can be computed as $r(s,a) = \phi(s;\theta_\phi)^\top w$. Following this, the Q function can be rewritten as:

$$Q(s,a) \approx \mathbb{E}^\pi\Big[r_{t+1} + \gamma r_{t+2} + \ldots | S_t = s, A_t = a\Big]$$

$$= \mathbb{E}^\pi\Big[\phi(s_{t+1};\theta_\phi)^\top w + \phi(s_{t+2};\theta_\phi)^\top w + \ldots | S_t = s, A_t = a\Big]$$

$$Q(s,a) = \psi^{\pi}(s,a)^{\top} \cdot w$$

where $\psi(s,a)$ are referred to as the *successor features* under policy π. The i^{th} component of $\psi(s,a)$ provides the expected discounted sum of $\phi_t^{(i)}$ when following policy π starting from state s and action a. It is assumed that the features $\phi(s;\theta_\phi)$ are representative of the state s, such that $\psi(.)$ can be turned into a function $\psi^{\pi}(\phi(s_t;\theta_\phi), a_t)$. For brevity, $\phi(s_t;\theta_\phi)$ is referred to simply as ϕ_t and $\psi^{\pi}(s,a)$ as $\psi(s,a)$.

The decomposition neatly separates the Q-function into two learning problems, for ψ^{π} and w: estimating the features under the current policy dynamics and estimating the reward given a state. Because the decomposition still has the same form as the Q-function, the successor features are computed using a Bellman equation update in which the reward function is replaced by ϕ_t:

$$\psi^{\pi}(\phi_t, a_t) = \phi_t + \gamma \mathbb{E}\Big[\psi^{\pi}(\phi_{t+1}, a_{t+1})\Big]$$

Such that approximate successor features can be learned using an RL method, such as Q-Learning [80]. The approximation of the reward vector w becomes a supervised learning problem. Hansen *et al.* [81] introduce a Variational Intrinsic Successor Feature (VISR) that introduces an unsupervised learning object to learn controllable features, which after training the unsupervised state representation, enables adaption across tasks by solving a linear regression problem and is evaluated on the Atari suite.

In an interesting cross between the options framework and SFs, Machado *et al.* [76] propose eigenoptions, which use the eigenvectors of the learned successor features to obtain options that encode diffusive information flow in the environment. This framework leads to highly composable options that work well in stochastic environments.

3.4 Summary

This chapter has presented a review of academic research related to the contributions of this thesis. Foundational and state-of-the-art methods within DRL focusing on sample efficiency, task transfer, and planning were reviewed. The first proposed algorithm for improving sample efficiency with a novel form of importance sampling is introduced in the next chapter.

Chapter 4

Noisy Importance Sampling Actor-Critic

This chapter presents the Noisy Important Sampling Actor-Critic (NISAC) method. NISAC uses a noisy weighting for off-policy samples, enabling their use in an on-policy algorithm. The method presented in this chapter is from the published work [82] and focuses on improving the sample efficiency of DRL algorithms.

The organization of this chapter is as follows: Section 4.1 presents the introduction and our contributions, Section 4.2 covers background information, Section 4.3 describes the NISAC algorithm, Sec. 4.4 details the experimental results, analysis, and ablations of our methodology, and finally Sec. 4.5 provides concluding remarks.

4.1 Introduction

Recent advances in RL have enabled the extension of long-standing methods to complex and large-scale tasks such as Atari [6], Go [83], and DOTA [84]. The key driver has been the use of deep neural networks, a non-linear function approximator, with the combination usually referred to as DRL [30, 6]. However, deep learning-based methods are usually data-hungry, requiring millions of samples before the network converges to a stable solution. As such, DRL methods are usually trained in a simulated environment where an arbitrary amount of data can be generated.

RL algorithms can be classified as either learning in an *off-policy* or *on-policy* setting. In the *on-policy* setting, an agent learns directly from experience generated by its current policy. In contrast, the *off-policy* setting enables the agent to learn from experience generated by its current policy or/and other separate policies. An algorithm that learns in the *off-policy* setting has much greater sample efficiency as old experience from the current policy can be reused; it also enables *off-policy* algorithms to learn an optimal policy while executing an exploration-focused policy [52].

A famous off-policy method is Q-Learning [38] which learns an action-value function, $Q(s, a)$, that maps the value to a state s and action a pair. DQN, the marriage of Q-Learning with deep neural networks, was popularised by Mnih *et al.* [6] and used various modifications, such as experience replay, for stable convergence. Within DQN, experience replay [85] is often motivated as a technique for reducing sample correlation. Unfortunately, action-value methods, including Q-Learning, have two significant disadvantages. First, they learn deterministic policies, which cannot handle problems that require stochastic policies. Second, finding the greedy action with respect to the Q function can be costly for large action spaces. To overcome these limitations, one could use policy gradient algorithms [86], such as A2C an on-policy actor-critic method [41], which learn in an *on-policy* setting at the cost of sample efficiency.

The ideal solution would be to combine the sample efficiency of *off-policy* algorithms with the desirable attributes of *on-policy* algorithms. Work along this line has been done by using importance sampling [14] which adjusts off-policy samples according to a weight ρ, the ratio between the current policy $\pi(a|s)$ and experience policy $\mathcal{B}(a|s)$. Where the weight is defined as $\rho = \frac{\pi(a|s)}{\mathcal{B}(a|s)}$. NISAC modifies A2C by using additive action space noise, aggressive truncation of importance sample weights, and large batchsizes enabling off-policy learning from stored trajectories. The contributions of this work are as follows:

- We introduce Noisy Importance Sampling Actor-Critic (NISAC), a fully *off-policy* actor-critic algorithm, that learns from stored off-policy trajectories.

- Experimentally prove in the Atari domain, that NISAC outperforms, in performance

and sample efficiency, both A2C [41] and the off-policy truncated importance sampling method [16].

- Show the addition of additive action space noise, to the numerator of ρ, changes the distribution of weights, improves learning, and that Gumbel noise, rather than Normal or Uniform, is most performant.
- NISAC training time is 40% faster than ACER, a SOTA off-policy actor-critic methods, nears its performance on several environments, and is easier to implement.

4.2 Background

The Gumbel distribution is a probability distribution that is used to model the maximum of value from a set of independent samples [87]. The density of this distribution is defined as:

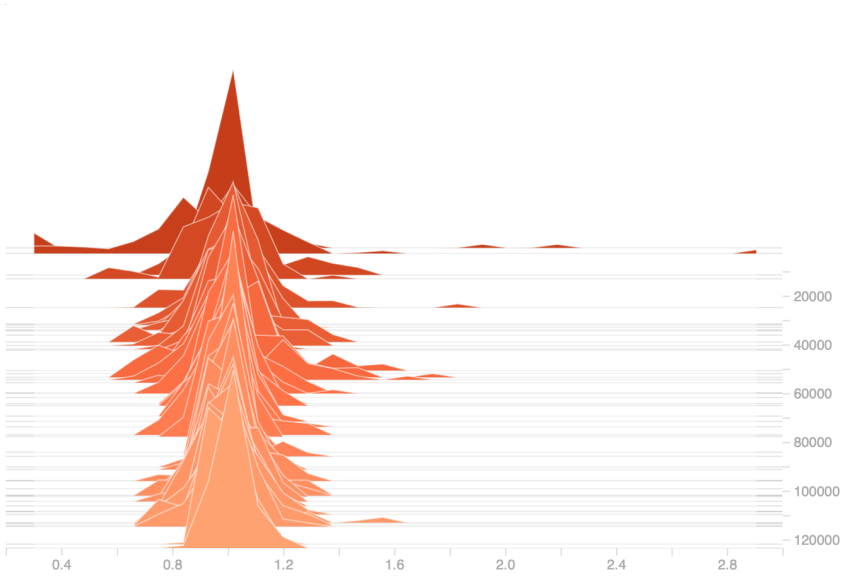
$$p(x) = \frac{1}{\beta} \exp(-z - \exp(-z)) \tag{4.1}$$

where $z = \frac{x-\mu}{\beta}$ and parameterized by scale β and location μ. We evaluate noise sampled from the Gumbel distribution as a potential candidate for additive noise in NISAC. The Gumbel distribution has several applications within machine learning, such as the Gumbel-Max [88] and Gumbel-Softmax tricks [89]. However, our usage of the Gumbel distribution differs from the Gumbel-Max and Gumbel-Softmax in that we see Gumbel noise as perturbing the current policy – as noise would from any other distribution.

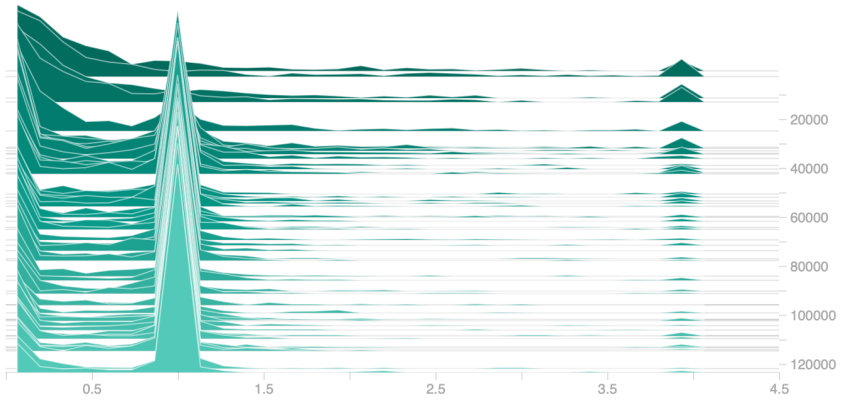
4.3 Model

4.3.1 Model Architecture

NISAC builds off the *on-policy* actor-critic algorithm A2C, such that direct comparison to methods like ACER are possible. To enable *off-policy* learning NISAC uses aggressive clipping



(a) Truncated ρ with current policy



(b) Truncated ρ with noisy policy

Figure 4.1: Histograms of importance sampling weights ρ during training. The x-axis is the magnitude of $\bar{\rho}$ and the y-axis is the number of updates since start of training. Both ratios ρ are truncated between $[0, c]$, where is c the truncation value and here, $c = 4$. a) Using truncated ρ with the current policy in the numerator and b) the ratios seen during training from NISAC, which corresponds to use of the noisy policy in the numerator. In this case noise was sampled from a standard Gumbel distribution.

on importance sampling, additive noise, and large batches sampled from a replay memory.

Psuedo-code for NISAC is provided in Algorithm 1. We begin by defining a few quantities

used in the importance sampling weight ρ. In importance weighting, two policy classes exist: the current policy $\pi(a|s;\theta)$ and the behaviour policy $\mathcal{B}(a|s)$. Because replay memory is being used, the behaviour policy is simply the distribution over actions with an old parameter setting θ^*:

$$\mathcal{B}(a_t|s_t) = \pi(a_t|s_t;\theta^*) \tag{4.2}$$

NISAC introduces a third policy, the noisy policy $\mathcal{F}(a|s)$[1], which results from adding noise ϵ drawn from some distribution $\mathcal{D}$ to the normalized logits of the current policy and passing them through a softmax:

$$\mathcal{F}(a_t|s_t) = \text{softmax}(\log \pi(a_t|s_t;\theta) + \epsilon) \tag{4.3}$$

We hypothesize that the addition of action space noise will reduce the bias of the agents action selection throughout training against the current policy and will act as a regularizer on the learned policy. We can see, from Figure 4.1(b), where we truncate values to some constant c, that the addition of noise forces the ratio ρ into one of three modes instead of clumping around 1. In this work we examine the difference between noise drawn from Uniform, Gumbel, and Normal distributions. One major qualitative difference between these types of noise, with respect to the distributions shape, is that additive Gumbel noise results in less mass between modes. Further, Gumbel noise has been successfully applied to the learning of discrete samples via gradient descent in RL [90]; making its use within this work ideal.

4.3.2 Training

Similarly to previous work, this study uses importance sampling ρ_t to weight the updates of the loss function [16, 17, 14]. However, instead of using the current policy, $\pi(\cdot|s_t)$, in the numerator,

[1] As $\mathcal{N}$ is typically used for Normal distributions, we used $\mathcal{F}$ to avoid confusion.

it is replaced with the noisy policy $\mathcal{F}(a_t|s_t)$:

$$\rho_t^{(i)} = \frac{\mathcal{F}(a_t^{(i)}|s_t)}{\mathcal{B}(a_t^{(i)}|s_t)} \tag{4.4}$$

The range of ρ_t is clipped to $[0, c]$ and this clipped importance weight is referred to as $\bar{\rho}_t$. Clipping the upper bound prevents the product of many importance weights from exploding and reduces the variance [16]. Wang *et al.* [17] notes that truncating the importance weights in this way introduces bias to the estimator. However, the value of having an unbiased algorithm is unclear, as shown by Thomas [91], and does not always correspond to improved performance. The effect of aggressively clipping ρ_t and using the noisy policy $\mathcal{F}(\cdot|s)$ in ρ_t has an interesting effect on the way the policy updates are weighted. To understand the effect of these two modifications we again refer to Figure 4.1, which shows the history during training of the ratios $\bar{\rho}$, truncated between $[0, c]$ to a constant c, where Figure 4.1(a) is plain truncated importance sampling and Figure 4.1(b) is truncated importance sampling with added action space noise.

From Figure 4.1(a), we notice that the majority of the importance weights $\bar{\rho}$ are centered around 1 resulting in off-policy samples that are weighted approximately the same. However, by using additive action space noise, in this case from the Gumbel distribution, we can see from Figure 4.1(b) a multi-modal distribution appears. The weights $\bar{\rho}$ form three distinct modes around $\{0, 1, c\}$, with small amounts of additional density "smeared" between modes. We hypothesize that the addition of noise, via the noisy policy $\mathcal{F}(a|s)$, has the effect of stabilising the updates to the network as the weighting assigned to each off-policy sample is near $\{0, 1, c\}$. Interestingly, each of the modes can be grouped into cases of "agreement" or "disagreement" between the noisy policy $\mathcal{F}(\cdot|s)$ and behaviour policy $\mathcal{B}(\cdot|s)$. Where the case of agreement corresponds to ratios and the mode near 1 while disagreements correspond to ratios and modes at 0 and c. More exactly, when the noisy policy disagrees with the behaviour policy, say $\mathcal{F}(\cdot|s) \approx 1$ and $\mathcal{B}(\cdot|s) \approx 0$, the update the policy receives is at most clipped by the upper bound of our interval: c. On the other hand, when the situation is reversed, but still in disagree-

Algorithm 1 Pseudo-code for k-step NISAC; Highlights are additions.

Initialize parameters θ and θ_v.
Initialize replay memory $\mathcal{M}$ with capacity N.
repeat
  for $i \in \{0, \cdots, k\}$ **do**
    Perform a_i according to $\pi(\cdot|s_i;\theta)$.
    Receive reward r_i and new state s_{i+1}.
    Store $(s_i, a_i, r_i, \pi(\cdot|s_t))$ in $\mathcal{M}$.
  end for
  Sample b trajectories $\{s_0, a_0, r_0, \mathcal{B}(\cdot|s_0), \cdots, s_k, a_k, r_k, \mathcal{B}(\cdot|s_k)\}$ from the replay memory $\mathcal{M}$.
  for $i \in \{0, \cdots, k\}$ **do**
    Sample ϵ from noise distribution $\mathcal{D}$.
    Compute $\pi(\cdot|s_i;\theta)$ and $\mathcal{F}(\cdot|s_i)$.
    $\bar{\rho}_i = \text{clamp}\left(\frac{\mathcal{F}(a_i|s_i)}{\mathcal{B}(a_i|s_i)}, 0, c\right)$
  end for
  $R \leftarrow \begin{cases} 0 & \text{for terminal } s_k \\ V(s_k;\theta_v) & \text{otherwise} \end{cases}$
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  for $i \in \{k-1, \cdots, 0\}$ **do**
    $R \leftarrow \begin{cases} 0 & \text{for terminal } s_i \\ r_i + \gamma R & \text{otherwise} \end{cases}$
    Accumulate gradients $d\theta \leftarrow d\theta + \bar{\rho}_i \nabla_\theta \log \mathcal{F}(a_i|s_i)\{R - V(s_i;\theta_v)\}$
    Accumulate gradients $d\theta_v \leftarrow d\theta_v + \nabla_{\theta_v}(R - V(s_i;\theta_v))^2$
  end for
  Perform update of θ using $d\theta$ and θ_v using $d\theta_v$.
until Max iteration or time reached.

ment, with $\mathcal{F}(\cdot|s) \approx 0$ and $\mathcal{B}(\cdot|s) \approx 1$, the policy has an importance weight of approximately 0.

Putting this all together the update equation for NISAC is as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{B}}[\bar{\rho}_t A(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{4.5}$$

However, empirically we found that using the noisy policy in place of the current policy, in $\nabla \log \pi(a_t|s_t)$, produces better performance. This is empirically validated by an ablation in Section 4.4.1. We hypothesize that the usage of the noisy policy in place of $\pi(a|s)$ affects three changes during learning. First, by adding action space noise the resulting policy will tend towards a categorical distribution earlier in training which will affect how strongly the networks weights are updated. Second, as the noisy policy used in both the numerator of ρ_t and in place of the policy score use the same set of noise samples $\{\epsilon_i, \ldots, \epsilon_k\}$ the updates

synchronize such that the policy is updated maximally in agreement. And finally, we believe the additive noise acts as strong exploration force to the policy as it can "shake-out" policies that are near deterministic, even later in training.

4.4 Experiments

Our experiments focus on the Atari domain [92] as there exists a large amount of variety between environments and the states are represented as raw high-dimensional pixels. The gym software package by Brockman *et al.* [93] was used to conduct all the experiments. The network architecture and hyper-parameters, for each respective algorithm, were constant throughout all experiments.

This study used the same input pre-processing and network architecture as Mnih *et al.* [41]. The network architecture consists of three convolutional layers as follows: 32 8×8 filters with stride 4, 64 4×4 filters with stride 2, and 32 3×3 filters with stride 1. The final convolutional layer feeds into a fully-connected layer with 512 units. All layers are followed by rectified non-linearity. Finally, the network outputs a softmax policy over actions and a state-value.

The experimental set-up used 16 threads running on a GPU equipped machine. As is standard in the Atari domain [6, 41, 17], all experiments are trained for 40 million frames. The optimization procedure used RMSProp [94] with a learning rate of 0.0005, policy entropy regularization weight of 0.01, and a discount factor of $\gamma = 0.99$. We estimate the gradient given in Equation 4.5 by uniformly sampling b trajectories of length k from a replay memory with size N. The advantage function $A(s_t, a_t) = R_t^{(k)} - V(s_t)$ is used, where $R_t^{(k)}$ is the bootstrapped k-step return for time t. A replay memory of size $N = 250000$ was kept, an update was performed every $k = 5$ steps in the environment, and a clamping coefficient of $c = 4$ was used. We sample $b = 64$ trajectories of length $k = 5$ for each update. Learning begins after we have collected $10,000$ samples in the replay memory. All experiments used the same hyperparameter settings and network architecture.

We tuned the hyperparameters and developed NISAC on the *FishingDerby* environment only; the other environments can be considered *"out of sample"*. We use the best 3 seeds and as standard we report the mean value of rewards achieved during training with 1 std. deviation as shaded areas on all graphs.

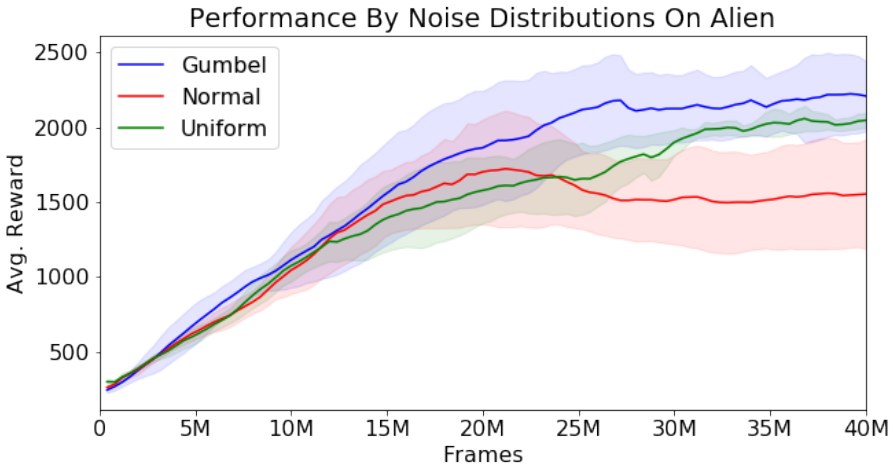
Due to limited computational resources, we were only able to evaluate NISAC on a subset of the environments and with a smaller replay memory size. Therefore, in this study, an effort was made to select environments that best showcase the performance of off-policy (truncated importance sampling) and on-policy (A2C) actor-critic methods. We note that the performance can be expected to improve with a larger replay memory, as seen with DQN and other off-policy methods using replay memory. Additionally, we focused the examination of our ablations on the *Alien* environment to reduce computational requirements.

The present work was compared with a SOTA off-policy actor-critic algorithm, ACER [17], an on-policy actor-critic algorithm, A2C, the synchronous version of A3C [41], and an off-policy actor-critic with truncated importance sampling (t-IS) [16, 14]. A2C and ACER used the *baselines* package provided by OpenAI [95]. The hyperparameters of ACER and A2C are identical to those used on the Atari environment by previous works [17] [41]. For t-IS we used a clipping constant $c = 10$ and replay memory of 250000 samples.

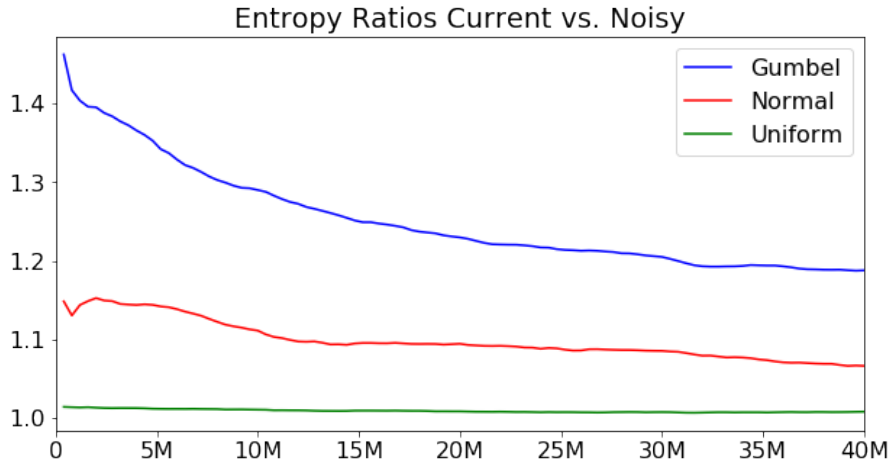
4.4.1 Additive Noise Distribution

Within this section, we vary the noise generating distribution used in NISAC and examine the effect on performance and learning. In particular, we compare noise sampled from the standard Gumbel distribution, the standard Normal distribution, and the Uniform distribution $[0, 1]$. Our analysis looked at the performance each variant achieved on the *Alien* Atari game over 40 million frames. The experiments only adjust the noise generating distribution with no other parameter changes.

From the results in Figure 4.2(a), we see that the Gumbel and Normal distributions have the same initial rate of improvement but diverge roughly midway through training. The Normal



(a)



(b)

Figure 4.2: Above we compare the behaviour between Gumbel, Normal, and Uniform distribution. a) Performance on the Atari game Alien under different noise distributions. b) We look at the entropy ratio between the current policy and the noisy policy $\frac{\mathbf{H}[\pi]}{\mathbf{H}[\mathcal{F}]}$.

distribution appears to degrade in performance before becoming stable at ~1500. The Uniform distribution, sampled between $[0, 1]$, has the same convergence characteristic as the other two distributions but instead of diverging away, similar to the Normal distribution, it continues upward before converging at ~ 2000. We can see that the Gumbel distribution is the most performant.

Next, in Figure 4.2(b), we compared the entropy $\mathbf{H}[.]$ of the current policy $\pi(a|s)$ over the noisy policy $\mathcal{F}(a|s)$. This experiment helps us understand how much each policy is exploring

relative to the other as well as how the ratios ρ change over time. For Uniform noise, we see it is flat throughout training implying that the current policy and noisy policy have equivalent entropy and therefore will "explore" at the same rate and produce importance sampling weights which are roughly 1. However, when examining the Gumbel and Normal noise variants we see that the noisy policy has greater entropy. When using Normal noise we see that initially the noisy policy has +15% more entropy but this quickly decays towards the end of training, falling below a 10% difference. We would like to note the small dip of the entropy ratio roughly coincides with the peak and draw down of the Normal variant in Figure 4.2(a). Looking towards the Gumbel variant, we can see the noisy policy has over +40% more entropy than the current policy and declines to about +20% towards the end of training. This implies that Gumbel noise will tend to "explore" more throughout training with the effect magnified earlier on. As the behaviour policy $\mathcal{B}(a|s)$ will roughly trail behind the current policy $\pi(a|s)$ during training we can expect the ratios ρ to be at the outer modes 0 and c. Later in training, as the entropy ratio drops the likelihood that the policies disagree decreases which corresponds to ratios ρ at roughly 1. Referring back to Figure 4.1(b), we can indeed see the two aforementioned trends occur: ρ is mostly around 0 and c and then most of the ratios go to 1. The earlier stages coincide with greater exploration as the two policies are in disagreement more often.

4.4.2 Noisy Policy Placement

Here, we examine the effect of using the noisy policy $\mathcal{F}(a_t|s_t)$ in the importance ratio $\bar{\rho}_t$ and as a replacement for the current policy $\pi(\cdot|s)$ in the policy gradient update. Going forward we use the Gumbel distribution, as by Section 4.4.1, it is the most performant. We examine different combinations of the current or noisy policy which modify Equation 4.5 as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathcal{B}}[\min(c, \frac{x}{\mathcal{B}(a_t|s_t)})A(s_t, a_t)\nabla_\theta \log y] \tag{4.6}$$

with each combination specified as a tuple of the form (x, y). Specifically, we look at the

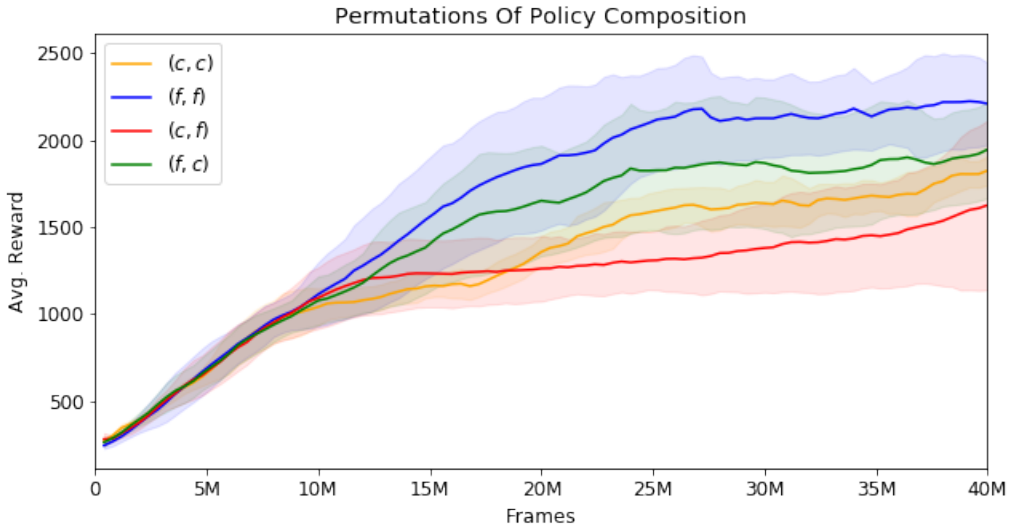


Figure 4.3: Variations in the choice of policy x in numerator of ρ_t and the policy y used to update the network. We see that the combination corresponding to NISAC has the highest performance and faster convergence speed. In the legend, c corresponds to current policy π and f corresponds to the noisy policy $\mathcal{F}$.

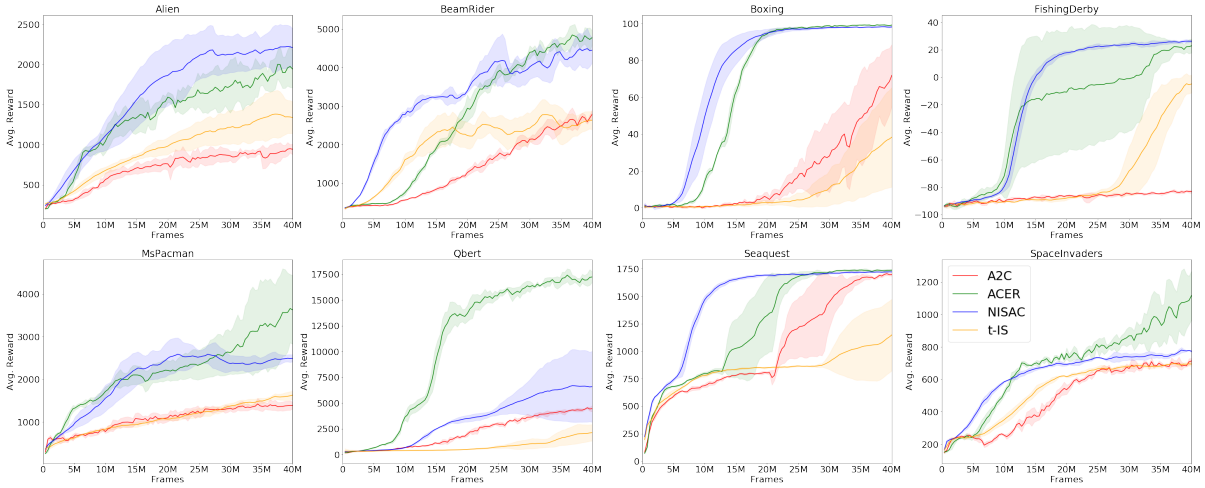


Figure 4.4: Training performance across 8 Atari games. We see the performance of NISAC (shown in blue) against ACER (shown in green), an off-policy actor-critic algorithm, the on-policy algorithm A2C (shown in red), and the off-policy actor-critic algorithm t-IS (shown in orange). The graphs show the average performance over 3 seeds with 1 standard deviation shown as the shaded region. NISAC matches or exceeds the performance of A2C and t-IS on all environments shown; while in all cases achieving improved convergence speed.

following combinations (π, π), $(\pi, \mathcal{F})$, $(\mathcal{F}, \pi)$, and $(\mathcal{F}, \mathcal{F})$. The combinations specified by (π, π) results in t-IS, no noise, and $(\mathcal{F}, \mathcal{F})$ in NISAC. From Figure 4.3 where c corresponds to current policy π and f corresponds to the noisy policy $\mathcal{F}$, we see that variant $(\mathcal{F}, \mathcal{F})$, corresponding to

NISAC, has the highest performance and fastest rate of convergence. This variant outperforms $(\mathcal{F}, \pi)$ showing that there is indeed some added benefit of additional noise. Finally, we note that the lowest performing variant, $(\pi, \mathcal{F})$, uses additive noise to the policy in the importance weight ratio. As mentioned previously and shown in Section 4.4.1, we see that the noisy policy strongly affects how the networks weights are updated during all points of training.

4.4.3 Atari Results

To test the proposed methodology, the performance of NISAC on a subset of Atari environments was examined. In particular, the following environments were investigated: *Alien, BeamRider, Boxing, FishingDerby, MsPacman, Qbert, Seaquest,* and *SpaceInvaders*. We report the average reward every 1000 episodes over 40 million frames. As mentioned previously, environments were chosen where either the *on-policy* algorithms perform well or where there is a clear difference in performance between an *off-policy* and *on-policy* method. For NISAC, we used noise sampled from the Gumbel distribution, validated in Section 4.4.1, and the noisy policy in the numerator of the importance weight and the policy gradient update, as validated in Section 4.4.2.

From Figure 4.4, we see the performance of NISAC, shown in blue, in comparison to the *off-policy* ACER algorithm, shown in green, the *on-policy* A2C algorithm, shown in red, and the t-IS algorithm, shown in orange. We see that the use of replay memory and learning with *off-policy* samples significantly improves the sample efficiency of both NISAC over A2C. Qualitatively, we see that NISAC converges significantly faster than A2C while also exceeding A2C's performance on this subset of Atari environments.

The performance gap between t-IS and NISAC is also large, with NISAC showing both increased performance and sample efficiency. The difference from NISAC to t-IS is an aggressive truncation of the importance weights and the noisy policy $\mathcal{F}(\cdot|s)$ used in both the importance weight and policy. As seen from Figure 4.4 this gives a significant increase in performance.

We achieve similar sample efficiency between the *off-policy* actor-critics, NISAC and ACER,

Algorithm	Wallclock Time
ACER	8h47m
NISAC	6h14m
A2C	4h53m
t-IS	4h39m

Table 4.1: Average Wallclock Time over Atari games: We can see that NISAC is 40% faster than ACER. The other baseline methods, A2C & t-IS, are faster still but are less performant across all Atari environments.

across each environment. We see that NISAC sees a fast initial increase in performance across almost all environments. ACER, a SOTA algorithm, out performs NISAC on the *MsPacman*, *Qbert*, and *SpaceInvader* environments. However, we note that NISAC trains 40% faster than ACER, shown in Table 4.1, and is significantly easier to implement with fewer modifications to the A2C algorithm, while providing better performance than both A2C and t-IS.

4.4.4 Stability

It is natural to inquire on the stability of this method, as we rely on additive noise which could cause instability after an optimal policy has been reached. To this end, we evaluate the stability of NISAC by increasing the number of training iterations such that 150 million frames are seen. The *Boxing* and *FishingDerby* environments are used for this evaluation. The environments were chosen as the policy had achieved the highest score during training, a known ceiling, and any instability would cause a divergence to a sub-optimal policy.

From the rather uninteresting graphs, shown in Figure 4.5, we see that NISAC can converge to and maintain a stable policy even with continued parameter updates. To overcome the noise added by the Gumbel distribution requires the network to output a near one-hot-encoded categorical distribution.

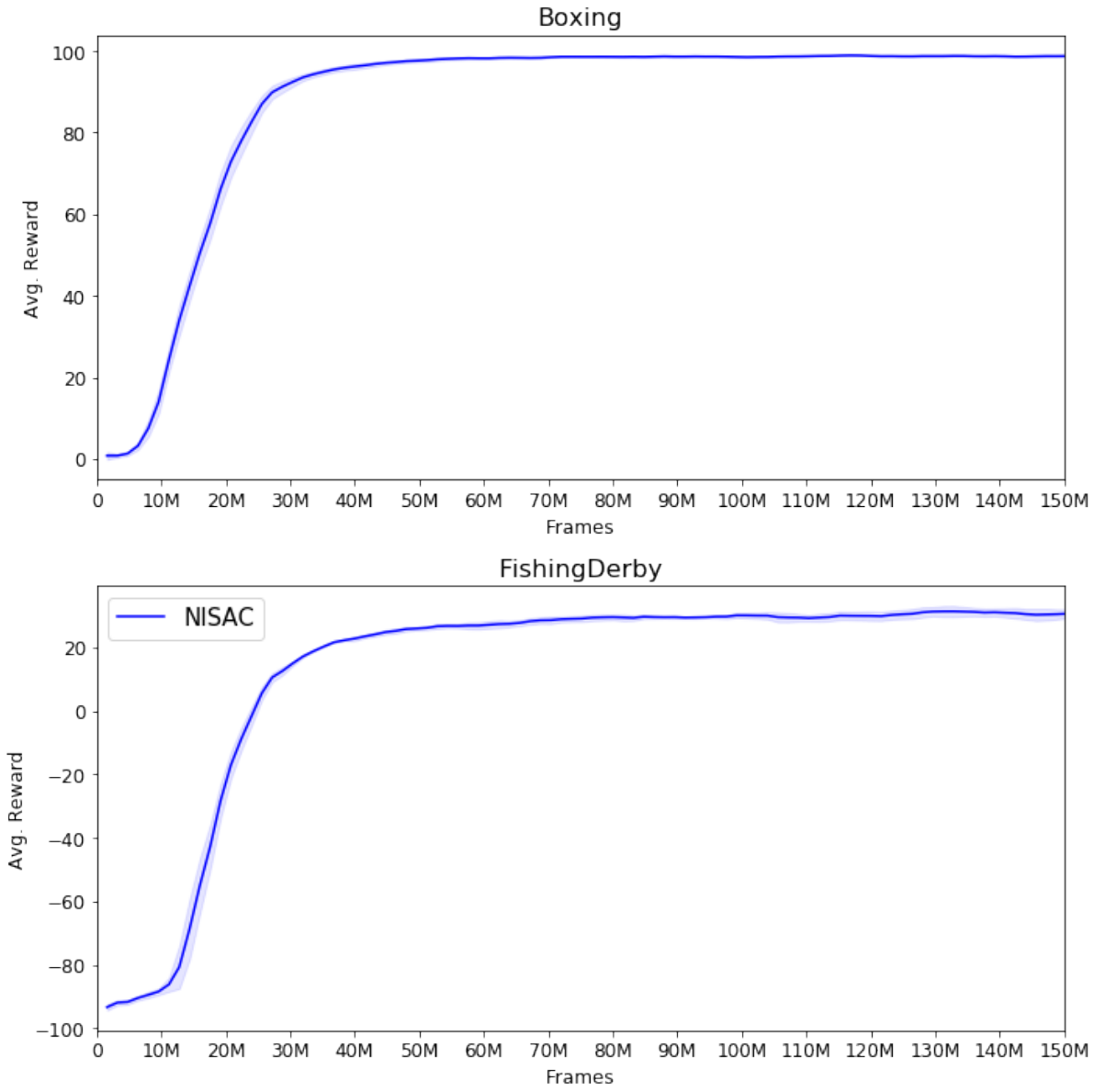


Figure 4.5: Stability of NISAC with extended training time. NISAC is trained for 150 million frames, $\sim 4x$ longer, on the *Boxing* and *FishingDerby* environments. We see that NISAC experiences little to no oscillations in performance even with continued weight updates.

4.4.5 Ablations Of Components

In this experiment, we performed ablations of NISAC to understand the importance of each component and impact on NISAC's performance. The results of the ablation are shown in Figure 4.6 with the complete NISAC algorithm, that is all the components that comprised NISAC, in blue and a stripped version as a red curve. We start with a *base* version, essentially

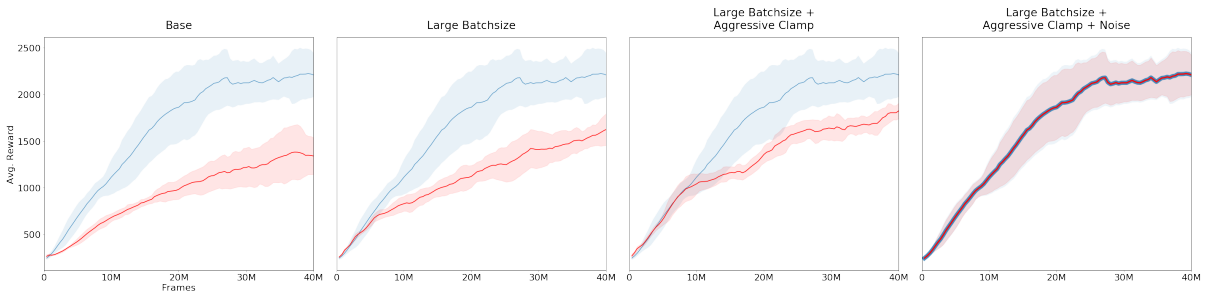


Figure 4.6: Ablations of NISAC. The full NISAC algorithm is shown as the blue curve while the stripped versions are shown in red. From left to right we gradually add the components onto the *base* version until we arrive at the full NISAC algorithm, shown in the last pane. The lines in the last pane are identical but with one graph stylized.

the truncated importance sampling algorithm (t-IS) [16], shown in the left-most panel in Figure 4.6.

From Table 4.2 we see that the *base* version has a performance -39.39% lower than NISAC. The addition of a larger batchsize, from 16 sampled trajectories, to 64 as shown in the second panel in Figure 4.6 causes an increase of $+21.30\%$ over the *base* version and narrows the difference to NISAC to -26.49%. Using an aggressive clamp of 4 instead of 10 on the importance sampling ratio $\bar{\rho}$ improves performance by an additional $+12.33\%$. Finally, the addition of noise sampled closes the gap with a final increase of $+21.09\%$. It is clearly shown that *large batchsize* and *additive noise* contribute the most to the performance increase between stripped versions.

	%Δ to NISAC	%Δ from last
Base	-39.39%	N/A
+Large Batchsize	-26.49%	+21.30%
+Aggressive Clamp	-17.43%	+12.33%
+Noise	0%	+21.09%

Table 4.2: The table provides the percent deltas between either the stripped version to NISAC or the current model to the last. We measure the change between the last 100 episodes.

Additionally, while difficult to quantify, we can see from the plots that the *aggressive clamp* and *additive noise* improve sample efficiency the most. It is clear that all the components, large batchsize, aggressive clamping, and additive policy space noise are crucial to NISAC's

aggregate performance.

4.5 Summary

This chapter has introduced Noisy Importance Sampling Actor-Critic (NISAC), a fully *off-policy* actor-critic algorithm that learns from stored off-policy trajectories. We have proven, experimentally, that NISAC improves upon the performance and sample efficiency of A2C [41], an *on-policy* actor-critic, and truncated importance sampling [16], an *off-policy* algorithm. NISAC nears the performance of ACER [17], a SOTA *off-policy* actor-critic method, on several environments while completing a training session in 40% less time and being significantly easier to implement. We have analyzed the effect of additive action space noise, identified the Gumbel distribution as the most performant variant, and examined where the noisy policy can be used within the importance sampling weight ρ and policy gradient update. Our analysis shows that additive action space noise fundamentally changes the distribution of importance sample weights ρ during training. Furthermore, finally, we have shown that each component in NISAC contributes to its improved performance over the baseline methods. Even with additive action space noise, the learned policies are stable. However, a limitation of the aforementioned work is that it cannot be easily applied to environments with continuous action space, such as those in robotics. This is due to how additive action space noise is injected and the choice of distribution. In particular the Gumbel noise, which was most performant, is suited for categorical distributions only. Therefore, future work is required to find a distribution that is applicable to continuous action spaces.

Chapter 5

Dynamic Planning Networks

This chapter presents Dynamic Planning Networks (DPN), an adaptive planning method for deep reinforcement learning. The method learns how to dynamically adjust its internal planning algorithm based on the needs of the particular task. The work presented in this chapter is taken from the published work [96] and focuses on improving the action efficiency of DRL algorithms. The chapter is organized as follows: Section 5.1 presents an introduction and contributions, Section 5.2 covers our architecture and training procedure, Section 5.3 details the experimental design used to evaluate our architecture, the experimental results, and analysis, and finally Section 5.5 provides a summary of the method.

5.1 Introduction

The central focus of RL is the selection of optimal actions to maximize the expected reward in an environment where the agent must rapidly adapt to new and varied scenarios. Various avenues of research have spent considerable efforts improving core axes of RL algorithms such as performance, stability, and sample efficiency. Significant progress on all fronts has been achieved by developing agents using deep neural networks with model-free RL [6, 41, 97, 98]; showing model-free methods efficiently scale to high-dimensional state space and complex domains with increased compute. Unfortunately, model-free policies are often unable to gen-

eralize to variances within an environment as the agent learns a policy which directly maps environment states to actions. A favorable approach to improving generalization is to combine an agent with a learned environment model, enabling it to reason about its environment. This approach, referred to as model-based RL learns a model from past experience, where the model usually captures state-transitions, $p(s_{t+1}|s_t, a_t)$, and might also learn reward predictions $p(r_{t+1}|s_t, a_t)$. Usage of learned state-transition models is especially valuable for planning, where the model predicts the outcome of proposed actions, avoiding expensive trial-and-error in the actual environment – improving performance and generalization. This contrasts with model-free methods which are explicitly trial-and-error learners [99]. Historically, applications have primarily focused on domains where a state-transition model can be easily learned, such as low dimensional observation spaces [100, 62, 101], or where a perfect model was provided [102, 83] – limiting usage.

The planning style, how the state-transition model is applied, is an important factor to consider as it can affect the efficiency of the simulated trajectory. Typically, the planning style is explicitly set per architecture, with various styles used such as: recursively expanding all available actions per state for a fixed depth [98, 10], expanding all actions of the initial state and simulating forward for a fixed number of steps with a secondary policy [11], or performing many simulated rollouts with each stopping when a terminal state is encountered [83]. Using a single type of planning style is limiting as various situations in an environment might call for a dynamic planning style. For example, when the agent needs to explore the immediate surrounding area a breadth-first search is optimal – instead of proceeding depth-first down one trajectory. If the agent cannot adjust planning styles the resulting plan can be sub-optimal.

In typical planning architectures, the planner is unable efficiently reverse trajectories. The planner must undo previous actions, wasting simulation steps, before continuing down an alternative path. If the planner does not have enough remaining simulation steps, to fully or even partially undo a sub-optimal trajectory, the agent using this plan might perform poor actions. Additionally, if certain actions cannot be reversed the planner might try a nonsensical

move, which violates environment dynamics, and produce a plan with an unrealistic prediction. Again, this would lead the agent using this plan to incorrectly value a particular path forward. Ideally, in both of the aforementioned cases, the planner would be able to either "reset" the planning trajectory in one step or could "undo" the last action – bypassing the limitations imposed by the environment and state-transition model.

DPN aims to improve the planning efficiency via various architectural choices. By providing DPN's planner with a triplet of options to "reset", "undo", or "continue" from tracked states during planning it can avoid sub-optimal trajectories and dead-ends. Additionally, as DPN's planner is based on a recurrent network and has no imposed planning structure which, together with the tracked triplet of state, allows it to dynamically adjust planning styles depending on the current context. The contributions of this work are as follows:

- Dynamic Planning Network (DPN): a planning architecture that create plans with a learned dynamic planning style.

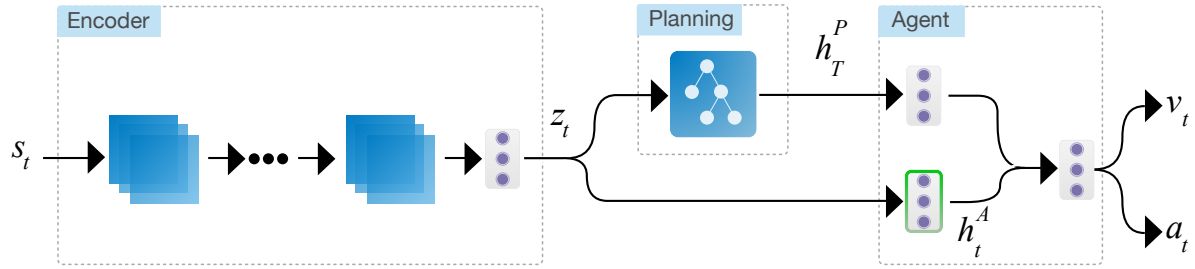
- We show that providing a planner with the option to choose *where* to plan from improves performance by reducing sub-optimal trajectories.

- A loss for the planner policy that balances between exploration and exploitation during planning.

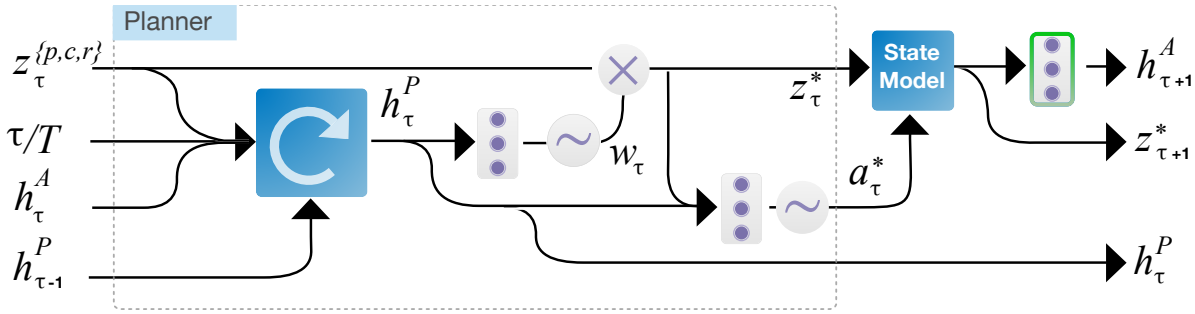
- DPN outperforms, both in performance and sample efficiency, other planning architectures on commonly used environments in the domain.

5.2 Model

We denote steps taken in the environment with subscript t and planning steps use subscript τ.



(a) Architecture Overview



(b) A Planning Step

Figure 5.1: a) Network Architecture: Encoder is comprised of several convolutional layers and a fully-connected layer (a box with 3 dots). Planning occurs for $\tau = 1, ..., T$. The result of planning is sent to the agent which emits an action a_t and state-value v_t. Planning uses a fully-connected layer within the agent, outlined in green, to generate an updated hidden state. *b)* A single planning step τ. The planner performs a step of planning using the state-transition model. Circles containing $\times$ indicate multiplication and circles with $\sim$ indicate sampling from the Gumbel Softmax distribution.

5.2.1 Model Architecture

DPN is composed of an agent policy π^A, multi-step planner policy $\pi^P_{w,a}$, a state-value function V, a learned state-transition model $\mathcal{M}$, and shared state encoder. Figure 5.1(a) illustrates a high-level diagram of the DPN architecture.

At its core, DPN extends actor-critic algorithms by adding a pathway dedicated to planning with a learned state-transition model, similar to other planning work [11, 65, 64, 10, 103]. We define a state-transition model as any model which predicts the next state given an action and the current state.

Using state-transition models in environments with complex dynamics and high dimen-

sional observation spaces has proven difficult as state-transition models must learn from agent experience and require significant amounts of samples and compute [104, 47, 105]. Often, it is much more efficient to instead learn and make predictions in a lower dimensional space [10]. Therefore, within this work we consider the state-transition model used by Farquhar *et al.* [10] which predicts within the latent embedding space z; where z is produced by an encoder or the state-transition model itself.

Planning components aim to improve the performance of a model-free agent by avoiding costly trial and error in the environment. However the style of planning, the way a state-transition model is used, differs between architectures each with its own benefits. The planning style can be a simple forward rollout [67, 104], enforce a particular structure [106, 10], or use predesigned patterns [11]. In this regard, DPN's architecture is constructed in such a way that it *learns* the best planning pattern to employ. Therefore, DPN's planner is based upon a recurrent neural network which naturally incorporates the recent history in a short-term memory, allowing flexible planning patterns to emerge.

However, an additional issue arises regardless of the planning strategy used: what should the planner do if the last action taken cannot be reversed? Even if, in theory a opposing action exists, there is no guarantee that the state-transition model will produce a coherent prediction. Assuming, temporarily, that the planner chooses an opposing action, which cannot undo the last action but is opposing, and the state-transition model happily follows through: the resulting predicted state would either be nonsensical or unreachable. Therefore to help alleviate this issue, we provide DPN's planner access to an "undo" and "reset" option. This requires tracking a triplet of state embeddings z during planning: continue (current state) z_τ^c, previous (parent state) z_τ^p, and reset (root state) z_τ^r. Where the reset state is the current state of the agent within the environment and the previous state is the last observed state during planning. Both options allow the planner to short-circuit the state-transition model.

Figure 5.2 illustrates, in a fictional environment during a round of planning, the utility provided by tracking and allowing the planner to select between the triplet of states. From

the *top row* of Figure 5.2, we show how the planner can use a state-transition model and the triplet of states to construct plans. Here, we interpret the plans as the dynamic expansion of a state-action tree. While in the *bottom row* of Figure 5.2, show the corresponding fictional environment where the red agent must capture the blue goals. The agent can only *push* the grey obstacles which means they can become irreversible stuck as no opposing action exists. The fictional environment illustrates how the added ability to select the "reset" or "previous" states gives improved efficiency to the planner.

As the planner progresses through the environment, shown in the *bottom row* of Figure 5.2a-d, that it pushed the grey obstacle to the left, blocking the goal. By using the "reset" state option[1], shown in the *top row* of Figures 5.2e, the planner can create an alternative route to the goal, shown in the *bottom row* of Figures 5.2e-f.

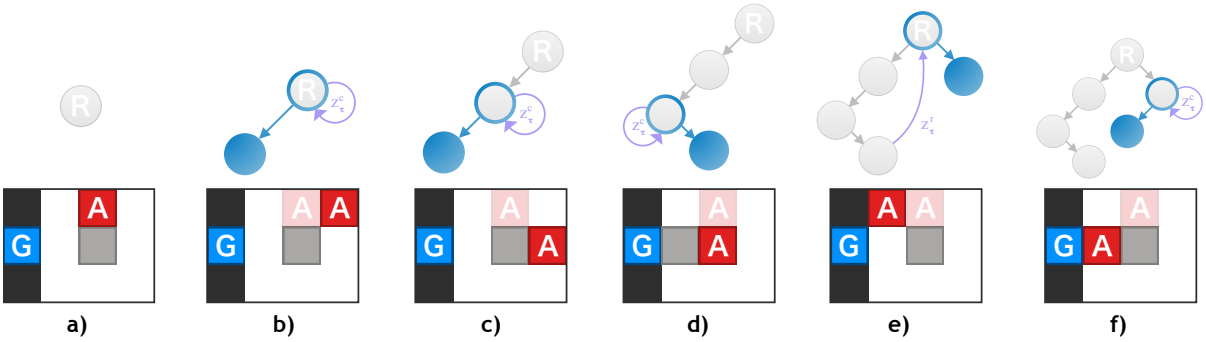


Figure 5.2: Tree Interpretation. *Top Row*: A tree interpretation of a created plan by DPN. State selections are shown in light purple and state-transitions are shown as blue. The source state is shown as a grey circle with a blue outline and the transitioned state as a fully blue circle. *Bottom Row*: A fictional environment in which the red agent must visit blue goals and can only *push*, and not *pull*, grey obstacles around. The faded agent is meant to signify the current state of the agent in the environment.

If the planner did not have access to the "reset" or "previous" states, the resulting trajectory would be sub-optimal in their predictive value or might contain incorrect information had the state-transition model violated the environment dynamics. Additionally, even if an opposing action did exist, in this case *pull*, the planner would waste planning steps to unroll this poor decision. Planning efficiency becomes especially important when a limited number of planning

[1]The "previous" state option would be a valid choice as well.

steps are budgeted.

In DPN, our planner runs for a fixed number of planning steps T, interacting with the state-transition model $\mathcal{M}$, before the agent selects an action a_t in the environment. Pseudo-code is provided for one step of acting by DPN in Algorithm 2. The weight W^A belongs to the agent and its output, given some embedding z, captures the agent's current view of the state in an embedding h^A. We refer to this as the "hidden state" of the agent. This is shown Figure 5.1(a) as the bottom pathway where W^A is the box with a green border.

At each planning timestep τ where $\tau \in \{1, \ldots, T\}$, the planner's policy, in a two-step manner, picks which state of the triplet to plan from using a sampled weighting w^*_τ and then selects an appropriate action a^*_τ given this selected state and history. The weighting w^* and action a^* are sampled using the Gumbel-Softmax trick [89] so we can learn in an end-to-end manner. The planner uses the state-transition model $\mathcal{M}$ to predict the next state $z^*_{\tau+1}$ given the selected state z^*_τ and action a^*_τ. The triplet of embeddings are then updated.

Algorithm 2 Pseudo-code for action selection with DPN

// Given StateModel, Encoder, Planner, and Agent policy.
// Given current state x_t.
$h^P_{t,\tau=0} \leftarrow$ init hidden state of Planner
$z_t = \text{Encoder}(x_t)$
$z^p_{t,\tau=1}, z^c_{t,\tau=1}, z^r_{t,\tau=1} = z_t$
for $\tau \in \{0, \cdots, T-1\}$ **do**
  $h^A_{t,\tau} = W^A z^c_{t,\tau}$
  $h^P_{t,\tau} = \text{RNN}([z^p_{t,\tau}, z^c_{t,\tau}, z^r_{t,\tau}, \frac{\tau}{T}, h^A_{t,\tau}], h^P_{t,\tau-1})$
  $w^*_{t,\tau} = \pi^P_w(\cdot|h^P_{t,\tau})$ // 1-hot action
  $z^*_{t,\tau} = [z^p_{t,\tau}, z^c_{t,\tau}, z^r_{t,\tau}]^T w^*_{t,\tau}$
  $a^*_{t,\tau} = \pi^P_a(\cdot|h^P_{t,\tau}; z^*_{t,\tau})$ // 1-hot action
  $z^*_{t,\tau+1} = \mathcal{M}(a^*_{t,\tau}, z^*_{t,\tau})$
  $z^c_{t,\tau}, z^p_{t,\tau} = z^*_{t,\tau+1}, z^c_{t,\tau}$
end for
$h^A_t = W^A z_t$
$a_t = \pi^A(a|h^A_t, h^P_{t,\tau=T})$

The planner is provided with a context comprised of the current triplet of embeddings, a float indicating the planning step, and the "hidden state" of the agent. Inclusion of the agent's "hidden state" h^A in the planner's is detailed when we discuss training, but briefly: the planner is partially trained to maximize the "surprise" of the agent so providing this information to the

planner is beneficial. As the planner uses a recurrent network, we found it best if the agent's policy π^A also consumes the final hidden state $h^P_{\tau=T}$ produced by the planner, shown in Figure 5.1(a). Doing so forces the planner to keep a running summary of the constructed plan and provides the agent's policy π^A with additional context.

5.2.2 Architecture Components

Model-free Pathway: As seen in Figure 5.1(a), the components along the model-free path, that is the bottom connections, are nearly identical to architectures used by actor-critic methods. Containing an convolutional encoder, optional hidden layers, and two outputs each representing the learned policy and state-value function. In this case the convolutional encoder processes a 2d input image $x_t \in \mathbb{R}^{C\times W\times H}$, with C channels and dimensions $W \times H$, to produce an embedded representation $z_t \in \mathbb{R}^Z$. We refer to the parameters of the actor's policy with θ_a and state-value function as θ_v. As the encoder is shared, it's parameters are a subset of all component parameters in DPN and therefore not explicitly mentioned.

From Figure 5.1(a), we see an additional two fully-connected layers along with the planning component. The bottom layer, outlined in green, is used to represent the agents current representation of the environment. It is used by the planner to estimate the "surprise" its plans provide. While the fully-connected layer, along the top connecting the planner to the agent, is used to further processes plans such that the agent can learn to extract pertinent information. We found this helpful as the produced plans might contain inaccuracies, caused by repeated application of the state-transition model, or contains non-actionable information. Weber *et al.* [11] use a module in I2A, which they refer to as a rollout encoder, with similar functional and purpose.

Planner: As shown in Figure 5.1(b), the planner is comprised of a RNN and two fully connected layers. While other types of architectures are possible for the planner, such as a Decision Transformer [107], a RNN was used for simplicity and because the learning process is quite compute intensive and a large transformer would slow the iteration cycle. Each layer

represents either the sub-policy used to choose a^* or w^*. The weight $w^*_\tau \in [0,1]^3$ only considers the current hidden state $h^P_\tau \in \mathbb{R}^H$ produced by the RNN. However, the simulated action $a^*_\tau \in \mathcal{A}$ considers both h^P_τ and the selected embedded state $z^*_\tau \in \mathbb{R}^Z$. We refer to the collective parameters of the planner as θ_p.

Figure 5.1(b) is primarily used to show the flow of information during one planning step τ. We can clearly see how tightly coupled the interactions between the planner and state-transition model are and how the planner is can fully manipulate the state-transition model based on the selected state z^*_τ and action a^*_τ.

State-Transition Model: The state-transition model is composed of two computation steps with a residual connection in between. The first step is meant to compute an action agnostic representation of the current state embedding z_τ. While the second computes the expected change to the environment given an action a_i. It is defined as follows:

$$\begin{aligned} z' &= z_\tau + tanh({}^{\text{env}}z^*_\tau) \\ z^*_{\tau+1} &= z' + tanh({}^{a_i}z') \end{aligned} \tag{5.1}$$

both ${}^{env} \in^{Z\times Z}$ and ${}^{a_i} \in^{\mathcal{A}\times Z\times Z}$ are learnable parameters of the state-transition model and are referred collectively to as θ_m[2]. We use the same state-transition model presented by Farquhar *et al.* [10] and also perform normalization of z^* after prediction. Doing so keeps the magnitude of the representation more consistent after several application of the state-transition model.

5.2.3 Training

DPN is trained to maximize the expected reward and as such we train the encoder, value network, and agent's policy using the k-step synchronous version of the advantage actor-critic algorithm (A2C) [41]. We refer to their collective loss, excluding the entropy regularization term, as $\mathcal{L}_{\text{A2C}}$

We treat the planner's policy as an actor, fitting into the A2C framework loss as an additive

[2]The action matrix is selected by multiplying a 1-hot encoding of the action.

term, but make two adjustments. First, the planner is trained within planning trajectories only, such that state-transitions in its "environment" are emulated by the state-transition model. This means that for a k-step trajectory, under A2C, the planner will see $k \times T$ samples. Second, the planner's reward is redefined to be the composition between the state-value the agent predicts for the next state $z_{\tau+1}$ and distance between the agent's hidden representations from states z_τ to $z_{\tau+1}$. We term this pseudo-reward as the utility the planner provides to the agent and define it as:

$$\mathcal{U}_\tau(h^A_{\tau+1}, h^A_\tau, z_{\tau+1}) = V(z_{\tau+1}; \theta_v) + \mathbf{D}[h^A_{\tau+1}, h^A_\tau] \tag{5.2}$$

where $z_{\tau+1}$ is the state transitioned to after performing an action a_τ in state z_τ, h^A_τ and $h^A_{\tau+1}$ are the hidden states of the agent after perceiving the current state z_τ and state transitioned to $z_{\tau+1}$ respectively, $\mathbf{D}$ is a distance measure, and $V(z_{\tau+1})$ is the value the agent assigns the next state $z_{\tau+1}$.

The two terms in Equation 5.2 tease between exploitation and exploration during planning. If only state-value term $V(.)$, analogous to the reward, where to be maximized by the planner then the produced plans would aim to maximize the reward – exploiting what is already known. In the opposite direction, if the planner focuses only on the distance term $\mathbf{D}$, then it will chose states producing larger differences in the agent's hidden state. More than likely, this would correspond to states which involve some "surprise" to the agent. A similar formulation has been proposed, outside of the planning domain, in work on intrinsic motivation; where the agent sees an external reward r_{ext} and an internal reward r_{int} [108]. This formulation also balances between the notion of exploration and exploitation, as the internally generated reward can be tangential to the reward produced by the environment. We extend this idea to the planning domain.

A secondary choice in the design of utility in Equation 5.2 is how to combine between the state-value and distance term. We found that an additive distance term helps useful reward signals through, instead of being attenuated, as the reward and distance can disagree on the usefulness of the next state; such as subsequent states with a epsilon distance but non-zero

Algorithm 3 Pseudo-code for DPN

Initialize parameters θ_a, θ_p, θ_v, and θ_m.
repeat
  for $i \in \{0, \cdots, k\}$ **do**
    Pick a_i by calling Algorithm 2 with x_t.
    Receive reward r_i and new state x_{i+1}.
  end for

$$R \leftarrow \begin{cases} 0 & \text{for terminal } x_k \\ V(z_k; \theta_v) & \text{otherwise} \end{cases}$$

  Reset gradients: $d\theta_{\{a,o,v,m\}} \leftarrow 0$.
  for $i \in \{k-1, \cdots, 0\}$ **do**
    for $\tau \in \{0, \cdots, T-1\}$ **do**
      $\mathcal{U}_{i,\tau} = V(z_{i,\tau+1}; \theta_v) + \mathbf{D}[h^A_{i,\tau+1}, h^A_{i,\tau}]$
      $d\theta_p \leftarrow d\theta_p + \nabla_{\theta_p} log \pi^P_{w,a}(\cdot|z_{i,\tau}; \theta_p) \mathcal{U}_{i,\tau}$
    end for

$$R \leftarrow \begin{cases} 0 & \text{for terminal } s_i \\ r_i + \gamma R & \text{otherwise} \end{cases}$$

    $d\theta_a \leftarrow d\theta_a + \bar{\rho}_i \nabla_{\theta_a} \log \pi(a_i|s_i; \theta_a)\{R - V(z_i; \theta_v)\}$
    $d\theta_v \leftarrow d\theta_v + \nabla_{\theta_v}\{R - V(z_i; \theta_v)\}^2$
    $d\theta_m \leftarrow d\theta_m + \nabla_{\theta_m}\{z_{i+1} - \mathcal{M}(z_i, a_i; \theta_m)\}^2$
  end for
  Perform update of θ_p using $d\theta_p$, θ_a with $d\theta_a$, θ_v with $d\theta_v$, and θ_m with $d\theta_m$.
until Max iteration or time reached.

rewards. Here, a multiplicative term can hinder learning as either quantity can be a near-zero number, such as the reward, causing the provided utility to register as essentially zero. Additionally, in the initial stages of our work, we did indeed consider a variant with a multiplicative distance term **D** but found sub-par performance when compared to an additive distance term. We hypothesize that the aforementioned effects caused the gradients to vanish, slowing learning along the planning pathway. Following from this, the planner is trained as a policy network only, with the loss $\mathcal{L}_P$ over the planning sequence T:

$$\mathcal{L}_p = \frac{1}{T-1} \sum_{\tau=0}^{T-1} \nabla_{\theta_p} log \pi^P_{w,a}(\cdot|z_\tau; \theta_p)\ \mathcal{U}_\tau(h^A_{\tau+1}, h^A_\tau, z_{\tau+1}) \tag{5.3}$$

During parameter updates to the planners parameters we block gradient computations to the parameters belonging to the agent. In this case the state-value function and the weight W^A used to update the agent's hidden state. We perform updates to the planner in this way as to

stop the planner from cheating by modifying the parameters of the agent that define its reward via the quantities in Equation 5.2. Various choices for distance functions exist, such as the cosine or L2 distance function. In this work we use the L1 distance function, as after empirical evaluation it was the most performant. Results of this evaluation are provided in Section 5.3.

We train the state-transition model by performing state grounding. As such, it is trained to minimize the L2 distance between the next embedded state z_{t+1}, produced by the encoder, and its prediction $\hat{z}_{t+1}$. The state-transition model makes its prediction from an embedding of the current state z_t and the action taken by the agent a_t that resulted in z_{t+1} [10]:

$$\mathcal{L}_{\mathcal{M}} = \left\{ z_{t+1} - \mathcal{M}(z_t, a_t; \theta_m) \right\}^2 \tag{5.4}$$

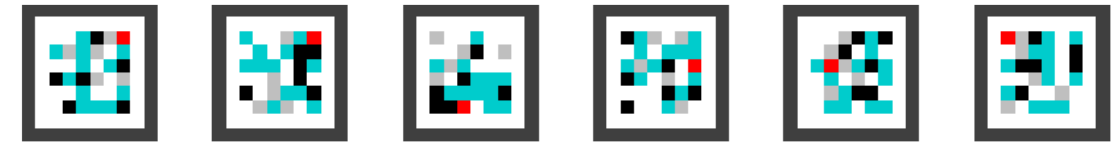
Combining our losses, the architecture is trained using the following gradient:

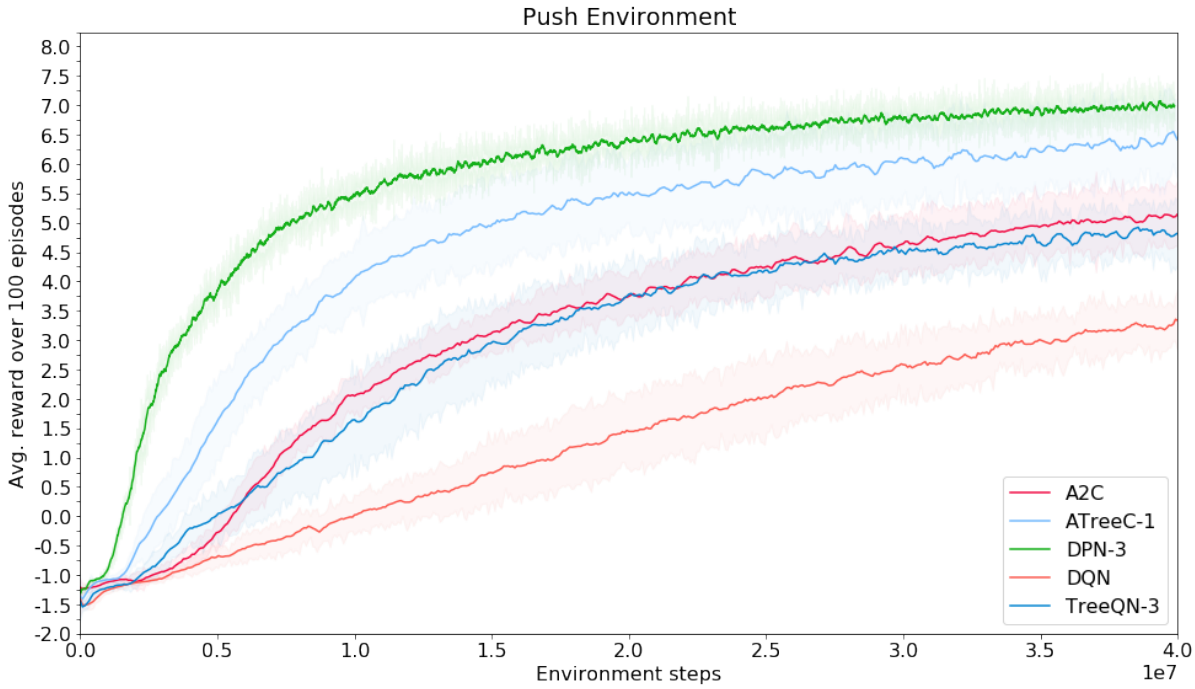
$$\Delta\theta = \nabla_{\theta_{\text{A2C}}} \mathcal{L}_{\text{A2C}} + \nabla_{\theta_p} \mathcal{L}_P + \lambda \nabla_{\theta_M} \mathcal{L}_{\mathcal{M}} - \beta \nabla_{\theta_{[a,p]}} H \tag{5.5}$$

where $\mathcal{L}_{\text{A2C}}$ is the agent's loss, both its policy and value function, $\mathcal{L}_P$ is the planner loss, λ is a hyperparameter controlling the state-grounding loss, H is the entropy regularizer computed for the agent and planner's policies, and β is a hyperparameter tuning entropy maximization of all policies; we used the same β value for each policy. The losses $\mathcal{L}_{\text{A2C}}$ and $\mathcal{L}_{\mathcal{Z}}$ are computed over all parameters; while $\mathcal{L}_P$ and its entropy regularizer losses are computed with respect to only the planner's parameters. DPN is fully specified in Algorithm 3.

5.3 Experiments

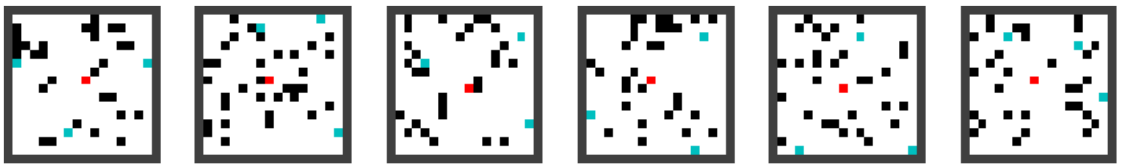
We evaluated DPN on a Multi-Goal Gridworld environment and Push [10], a box-pushing puzzle environment. Push is similar to Sokoban used by Weber *et al.* [11] with comparable difficulty. Within our experiments, we evaluated our model performance against either model-free



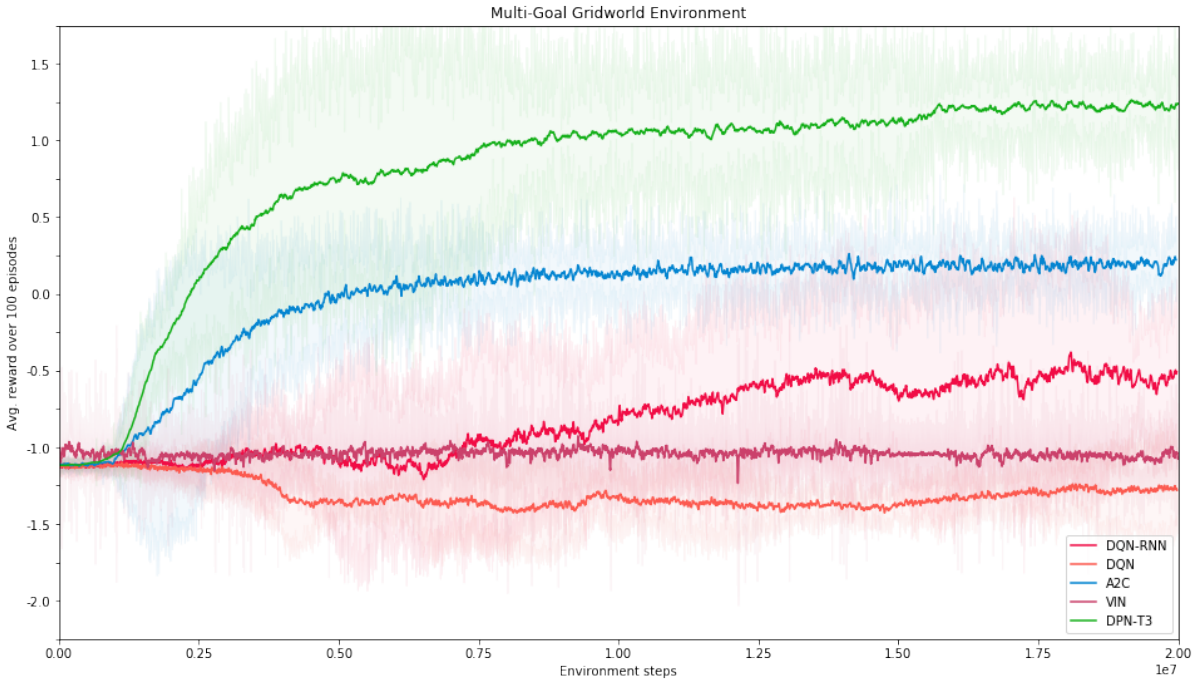


(a) Training Curve.

Figure 5.3: *Push Environment.* *a)* Randomly generated samples of the Push environment. Each square's coloring represents a different entity: the agent is shown as red, boxes as aqua, obstacles as black, and goals as grey. The outside of the environment, not visible to the agent, is shown as a black border around the map. *b)* Training curves with DPN compared to various baselines on Push environment.



(a) Multi-Goal Gridworld Environment Samples.



(b) Training Curve.

Figure 5.4: *a)* Randomly generated samples of a 16×16 Multi-Goal Gridworld environments. The agent is shown as red, goals in cyan, obstacles as black, and outside of the environment, not visible to the agent, is shown with a black border. *b)* Training curves with DPN compared to various baselines on 16×16 Gridworld with 3 goals.

baselines (A2C, DQN, and VIN) [3] or planning baselines (TreeQN and ATreeC). The experiments are designed such that a new scenario is generated across each episode, which ensures that the solution of a single variation cannot be memorized. We are interested in understanding how well our model can adapt to varied scenarios. Additionally, we investigate how planning length T affects model performance, how planner branching affects performance, different dis-

[3] We tested both vanilla implementations and versions using our architecture. A version of DPN with the planning components disabled, equivalent to an A2C model, was evaluated as well. We used the best performing version.

tance functions for the planner's reward function, and planning patterns that our agent learned in the Push environment. Full details of the environments, experimental setup, hyperparameters are provided in the supplemental material. Unless specified otherwise, each model configuration is averaged over 3 different seeds and is trained for 40 million steps. As mentioned earlier, we use a version of A2C algorithm with 16 workers, the RMSprop optimizer [109] with a learning rate of $5e-4$ and $\epsilon = 1e-5$.

5.3.1 Push

The Push environment is a box-pushing domain, where an agent must push boxes into goals while avoiding obstacles, with samples shown at the top of Figure 5.3. Since the agent can only push boxes, with no pull actions, poor actions within the environment can lead to irreversible configurations. The agent is randomly placed, along with 12 boxes, 5 goals, and 6 obstacles on the center 6x6 tiles of an 8x8 grid. Boxes cannot be pushed into each other and obstacles are "soft" such that they do not block movement, but generate a negative reward if the agent or a box moves onto an obstacle. Boxes are removed once pushed onto a goal. We use the open-source implementation provided by Farquhar *et al.* [110]. The episode ends when the agent collects all goals, steps off the map, or goes over 75 steps. We compare our model performance against planning baselines, TreeQN and ATreeC [10], as well as model-free baselines, DQN [6] and A2C [41].

5.3.2 Multi-Goal Gridworld

We use a Multi-Goal Gridworld domain with randomly placed obstacles that an agent must navigate searching for goals. The environment, randomly generated between episodes, is a 16x16 grid with 3 goals. We force a minimum distance between goals and between the agent and goals. The agent must learn an optimal policy to solve new unseen maps. Figure 5.4(a) shows several instances of a 16x16 Multi-Goal Gridworld. The rewards that an agent receives are as follows: +1 for each goal captured, -1 for colliding with a wall, -1 for stepping off the

map, -0.01 for each step, and -1 for going over the step limit. An episode terminates if the agent collides with an obstacle, collects all the goals, steps off the map, or goes over 70 steps. We evaluate our algorithm against model-free baselines such as A2C [41], variants of DQN (recurrent and non-recurrent) [6], and Value Iteration Network (VIN) [59]. Each baseline, with the exception of VIN, used the same encoder structure as DPN. We train for 20 million environment steps.

5.3.3 Planner Distance Functions

We vary the distance function used by the planner's loss defined in Equation 5.2. We examine L1, L2, KL, and Cosine distance functions.

5.3.4 Planner Branching

We examine the affect on performance of different branching options: current, reset, or all. We also included the ATreeC-1 baseline as this corresponds to the reset branching option of our architecture and serves as a sanity check.

5.3.5 Planning Length

Using the Push environment, we varied the parameter T, which adjusts the number of planning steps, with $T = \{1, 2, 3, 5\}$ evaluated. The Push environment was chosen because the performance is sensitive to an agent's ability to plan effectively.

5.3.6 Planning Patterns

We examine the planning patterns that our agent learns in the Push environment with $T = 5$. Here we are interested in understanding what information the agent extracts from the simulation as context before acting.

5.4 Results & Discussion

5.4.1 Push Environment

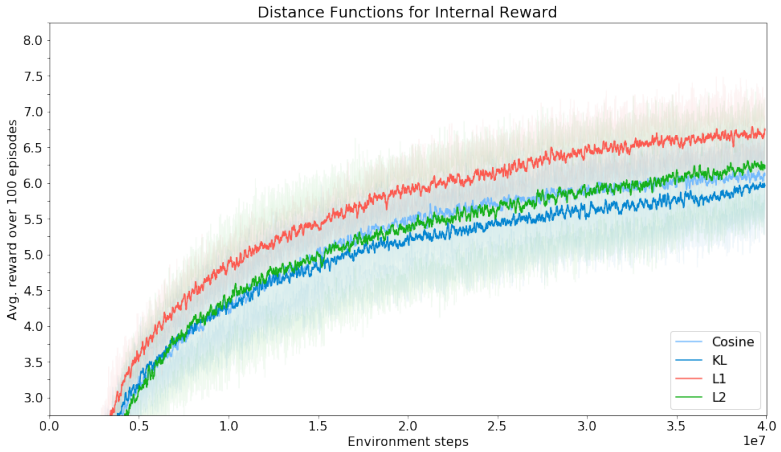
Figure 5.3(a) shows DPN compared to DQN, A2C, TreeQN and ATreeC baselines [4]. For TreeQN and ATreeC, we chose tree depths which gave the best performance, corresponding to tree depths of 3 and 1 respectively. Our model clearly outperforms both planning and non-planning baselines: TreeQN, ATreeC, DQN, and A2C. We see that our architecture converges at a faster rate than the other baselines, matching ATreeC-1's final performance after roughly 20 million steps in the environment. In comparison to the other planning baselines, TreeQN and ATreeC, require roughly 35-40 million steps: ∼2x additional samples.

We note that the planning efficiency of DPN is higher in terms of overall performance per number of state-transitions. On the Push environment, with $\mathcal{A} = 4$ actions, TreeQN with tree depth of $d = 3$ requires $(\frac{\mathcal{A}^{d+1}-1}{\mathcal{A}-1}) - 1 = 84$ state-transitions. In contrast, using DPN with a planning length of $T = 3$ requires only T state-transitions – a 96% reduction. Loosely comparing to I2As, simply in terms of state-transitions, we see that I2As require $\mathcal{A} \times L$ state-transitions per action step, where L is the rollout length. This performance improvement is a result of DPN learning to selectively expand actions and being able to dynamically adjust previously simulated actions during planning.

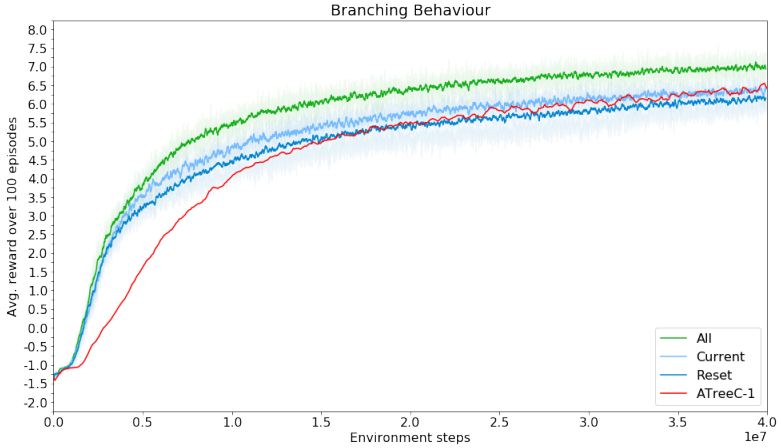
5.4.2 Multi-Goal Gridworld

Figure 5.4(b) shows, the results of DPN compared to various model-free baselines. Within this domain, the difference in performance is clear: our model outperforms the baselines by a significant margin. The policies that DPN learns generalizes better to new scenarios, can effectively avoid obstacles, and is able to capture multiple goals. Of the model-free baselines, we see that the A2C baseline performs the best. We believe that the A2C baseline is able

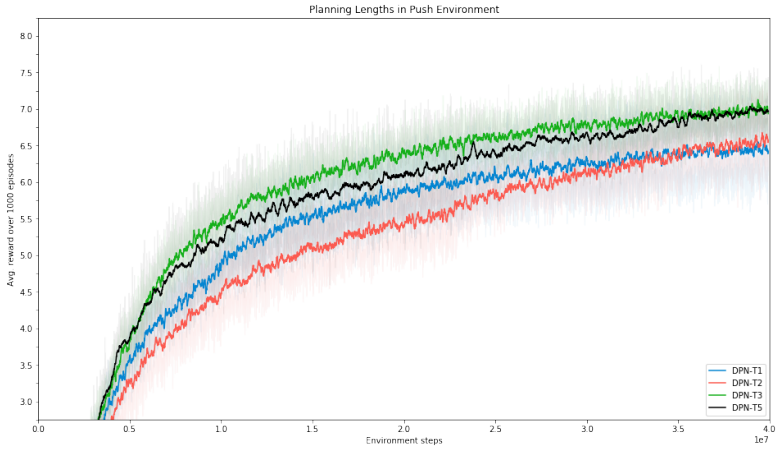
[4]The data for the training curves of DQN, A2C, TreeQN, and ATreeC were provided by Farquhar *et al.* via email correspondence. Each experiment was run with 12 different seeds for 40 million steps.



(a) Distance Functions



(b) Planner Branching



(c) Planning Length

Figure 5.5: a) *Distance Functions*: the performance of different distance functions. Centered on curve differences. b) *Planner Branching*: Various branching choices for the planner. *All* corresponds to the default architecture, *Current* results in a forward rollout, and *Reset* is the same as 1-step look ahead. ATreeC-1 corresponds to 1-step look ahead as well. c) *Planning Length*: Training over varying planning lengths, $T = \{1, 2, 3, 5\}$, in the Push Environment. Centered on curve differences.

to better explore the environment due to the multiple workers running in parallel throughout training. We trained VIN without curriculum learning for 20million steps with near identical settings[5] prescribed by Tamar *et al.* [59]. As seen in Figure 5.4(b), the DQN variants and VIN fail to capture any goals and do not achieve a score higher than -1.0. It should be noted we saw little performance improvement even when allocating the A2C and DQN baselines an additional 2x environment steps (40 million) or, in the case of DQN models, a 2-4x longer exploration period (8-16 million). The poor performance of the baseline models might be the result of high variance in the environment's configuration between episodes and sparse rewards. We believe that DPN performs better because it captures common structure present between all permutations of the environment by using the environment model. This allows it to exploit the model for planning in newly generated mazes.

5.4.3 Planner Distance Functions

From 5.5(a), we see an evaluation of distance functions used in Equation 5.2. The L1 distance function has the best performance with slightly faster convergence. While the L2, Cosine, and KL functions have worse performance. We hypothesize that the L1 distance performed better due to its robustness to outliers, a likely event during learning, as the distance is a function of noisy and changing vectors from the agent and state-transition model.

5.4.4 Planner Branching

In Figure 5.5(b), we see how different branching options affects the architecture performance. Our proposed branching improves performance of the architecture as compared to the *current* and *reset* options. Interestingly, the performance of our architecture when using the *reset* options is roughly the same as ATreeC-1. This is unsurprising as the *reset* and ATreeC-1 options employ a similar planning strategy of a shallow 1-step look-ahead. The small discrepancy

[5]The original results with VIN relied on curriculum learning. To provide a fair comparison all methods are trained without curriculum learning.

in performance could be due to ATreeC-1 evaluating all 4 actions while DPN evaluates only 3. We see that the *current* branching option results in better performance and amounts to a forward-only rollout. We hypothesize the performance difference between *current* and *reset* is from DPN being able to see the results of its actions from the first planning step over a longer time span.

5.4.5 Planning Length

In Figure 5.5(c), we see the performance of our model over the planning lengths $T = \{1, 2, 3, 5\}$. As seen in Figure 5.5(c), model performance increases as we add additional planning steps, while the number of model parameters remains constant.

As the planning length increases, we see the general trend of faster model convergence. Even a single step $T = 1$ of planning allows the agent to test action-hypotheses and avoid poor choices in the environment. From Figure 5.5(c) we see that an additional planning step, from $T = 1$ to $T = 2$, does not provide benefit until later in training. We see that both the planning lengths $T = 3$ and $T = 5$ tie for the best performance. Similar to Farquhar *et al.* [10], we hypothesize that this is due to a ceiling effect in this domain. This is clear as there are diminishing returns in performance with increased planning lengths. In terms of the shorter planning lengths, we suspect that they do not allow the planner to learn a policy that provides enough utility to the agent. Ideally, the architecture would be able to adjust the number of planning steps T dynamically based on current needs. We see this expansion, similar to the adaptive computation presented by Graves [111], as an interesting avenue for future work.

5.4.6 Planning Patterns

By watching a trained agent play through newly generated maps, we observe the possible emergence of planning-type patterns, which are shown in Figure 5.6 for $T = 5$. We see what appears to be a mixture between breadth-first search (BFS) and depth-first search (DFS). We found the resulting patterns to be quite consistent. Furthermore, the entropy of the planner's

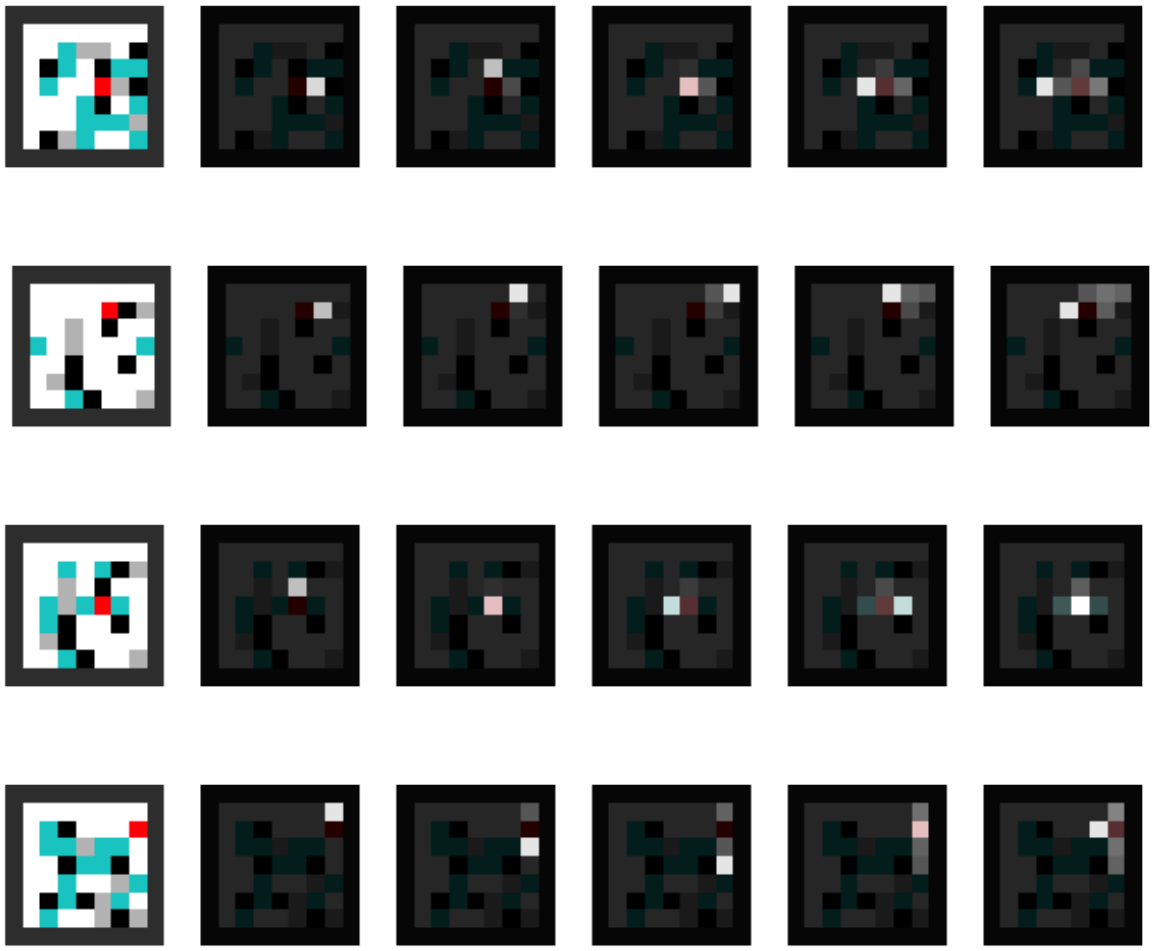


Figure 5.6: Samples of observed planning-type patterns the agent uses in the Push environment with $T = 5$. The faded environments, to the right of each sample, is used to signify when the agent is planning. Highlighted squares represent the location that the planner chose to move towards during planning. Faded squares show where it has been during planning.

policy was quite low at the end of training, which indicates the produces planning patterns are near-deterministic. In Figure 6(a) we see the agent does first performs a partial breadth-first search around itself before continuing farther left. Figure 6(b) shows the agent learning to exploit the "previous" state, which is a "lazy reset" to explore the upper right corner. The agent moves as follows: right with the current state, upward with the current state, right again with the current state, left from the previous state, and right one last time after resetting. In Figure 6(c) we see the planner does a partial breadth first search around the agent. In this case the

planner used the current state for the entire planning trajectory. Finally, in Figure 6(d) we see the planner alternate between a breadth-first and depth-first search. The planner moves upward from the root tree state, resets moving downward, picks the current state moving downward once more, selects the previous state while moving up, and then finishes by moving left using the current state.

5.5 Summary

This chapter has presented DPN, a new architecture for DRL that uses a planner and agent working in tandem. The planner is optimized to maximize a pseudo-reward, the utility provided to the agent, balancing exploitation and exploration during planning. We have demonstrated that DPN outperforms the model-free and planning baselines in the Mutli-Goal Gridworld and Push environments while using ~2x fewer environment samples. Furthermore, the ability of DPN to learn a dynamic planning style enables it to achieve much greater efficiency in terms of the state transitions required; this is especially evident when comparing TreeQN, with a fixed planning style, and DPN, with a dynamic planning style. By letting the planner learn its planning style, we see evidence of emergent planning patterns, such as breadth-first search. In the Push environment, we see DPN achieves greater or equal performance to TreeQN while requiring 96% percent fewer applications of the state-transition model. Taken all together, DPN, compared to other architectures, reduces the computational requirements to reach a similar level of performance. Finally, we have shown that allowing the planner to select *where* to plan from helps avoid sub-optimal trajectories. In our studies, we have provided evidence that the triplet previous, current, reset provides the greatest performance. While DPN clearly shows that by allowing the model to learn *how to plan*, there exist some limitations on the approach due to the planner module. As the planning module depends on a recurrent network, with a fixed memory space, the planner is limited in the amount of information it can store. This im-

plies that using highly complex states, with large amounts of information, could overwhelm a planner with a small memory. Using a strong memory store for the planner would unblock this limitation by allowing it to reason about rich state spaces, but we leave this for future work.

Chapter 6

Second-Order Successor Features

This chapter introduces the Second-Order Successor Feature (S2F) method, which extends the SF framework with a second-order reward function improving its robustness. The method presented here focuses on improving the transfer efficiency of DRL algorithms.

This chapter is organized as follows: Section 6.1 provides an introduction to the methodology, Section 6.2 introduces the SF framework with non-linear reward function (S2F) model along with its training methodology, Section 6.3 details the environments used in this chapter and the experiments carried out in each, Section 6.4 presents and discusses the results of the experiments, and finally Section 6.5 provides a summary conclusion for the chapter.

6.1 Introduction

Recent advances in RL have enabled the extension of long-standing methods to complex and large-scale tasks such as Atari [6], Go [83], and DOTA [84]. The main driver of these successes has been the use of deep neural networks, which are a class of powerful non-linear function approximators [112, 30, 113, 114]. However, this class of DRL algorithms requires immense amounts of data within an environment for training, often ranging from tens to hundreds of millions of samples [112, 114, 115]. Furthermore, commonly used algorithms often have difficulty in transferring a learned policy between related tasks, such as those where the environmental

dynamics remain constant, but the goal changes [116, 117, 118, 119]. In this case, the model must either be retrained completely or fine-tuned on the new task, in both cases requiring millions of additional samples. If the state dynamics are constant, but the reward structure varies between tasks, it is wasteful to retrain the entire model.

A more pragmatic approach would be to decompose the RL agent's policy so that separate functions can learn the state dynamics and the reward structure; doing so enables reuse of the dynamic model and only requires learning the reward component. SF do precisely this; a model-free action-value function is expressed as the dot product between a vector of expected discounted future state occupancies, the SFs, and another vector representing the immediate reward in each of those successor states [19, 76, 120, 121, 78, 122, 123, 124]. The factorization follows from the formulation of the reward as the dot product between a state representation vector and a learned parameter vector, which is a linear product. Therefore, transfer to a new task requires relearning only the reward parameters instead of the entire model and amounts to the supervised learning problem of predicting the current state's immediate reward. SF have strong roots in Neuroscience, where recent studies have provided evidence for the SF representation in the brain [125, 126]. It has also been suggested that the SF representation is the computational substrate within humans that leads to semi-flexible decision making [127].

Because the reward function of the SF framework is linear, it is fair to question whether the model can always accurately predict the reward. In Barreto *et al.* [128], an in-depth discussion is carried out on this question and given that no assumptions are made about the state representation, *theoretically* it is possible to enable perfect recovery of any reward function given a comprehensive predictive state representation.

The state representation within the SF framework is learned end-to-end by a state encoder, which should have enough representational power to disentangle the factors that are useful for reward prediction by a linear model. Indeed, given a large enough set of parameters, the encoder can perform well at both state reconstruction and reward prediction tasks. In the SF framework, the encoder is often trained to output a dense vector representation of the state that

helps minimize the least-squared error loss for both state reconstruction and reward prediction [79, 128, 81, 122]. This implies that the parameters of the encoder, which output the dense vector, must be used for both the reconstruction and prediction tasks. If the encoder learns a sub-optimal state representation for reward prediction, for example because of a highly complex environment, then the reward model may be unable to compensate in terms of reward prediction, given the limited set of reward models parameters. Eysenbach *et al.* [129] and Hansen *et al.* [81] have shown, within the successor feature framework, that there is no strong guarantee that the state encoder can always learn features that enable accurate modelling of the rewards. This has significant implications, since recent research state the importance on reward maximisation task [130, 131].

In this chapter, S2F (Second-Order Successor Features), a novel extension to the SF framework, is proposed, where the rewards are modelled with a second-order function. We show that the second-order function, which follows naturally from the original linear variant, gives a stronger guarantee of the model performance due to its non-linear representational structure and extra parameters. This is especially true in cases where the encoder learns state representations that are less than optimal and where a linear model does not have enough representational power to compensate. In particular, the additional parameters of the reward model should lessen the representation load of the encoder with regard to the reward task, enabling more of its representational capacity to be dedicated to modelling the environment in a task-agnostic manner. As explained in detail in the next sections, the new extra term in the reward model represents the future expected auto-correlation matrix of the state features. Therefore, this new formulation will also be key to a more comprehensive theory that addresses environmental stochasticity and the ability to use directed exploration during transfer, which will be dealt with in future works.

The contributions of this research are as follows:

- A novel formulation of the SF framework that uses a second-order reward function. This formulation increases the representational power of the reward function while decreasing

the representational load on the state encoder, providing stronger guarantees on performance.

- Under the new reward formulation, the extra term that appears is shown to model the future expected auto-correlation matrix of the state features.
- We provide preliminary results that show the second term can be used for guided exploration during transfer instead of relying on ϵ-greedy exploration.

6.2 Model

This section discusses the change to the SF framework, which adjusts the reward function from a linear function, to a non-linear function. First, a discussion of the new decomposition is given with the full derivation, after which, experimental support for this change will be presented and carefully analyzed.

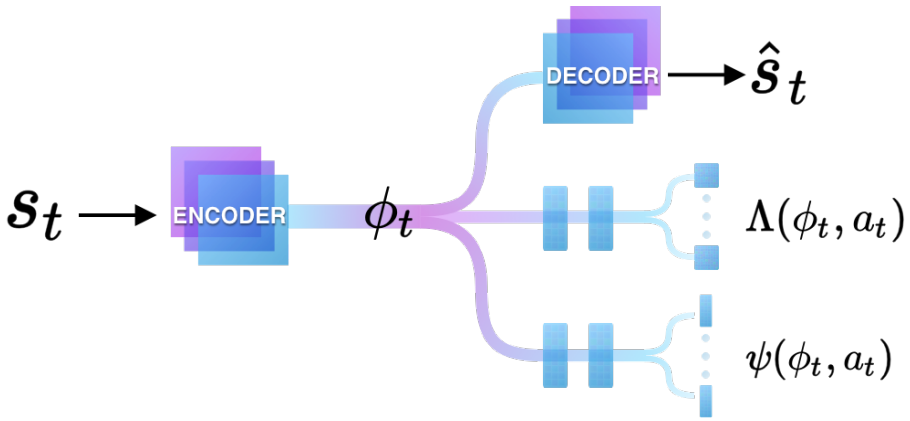
6.2.1 Non-linear Reward Function

The SF framework builds upon the functional representation of the current reward r_t as a linear combination of the current state representation $\phi_t \in \mathbb{R}^z$ and a learned reward vector $w \in \mathbb{R}^z$, such that $r_t = \phi_t^\top w$. In this work we extend the reward model by changing this linear reward model to one with the following form:

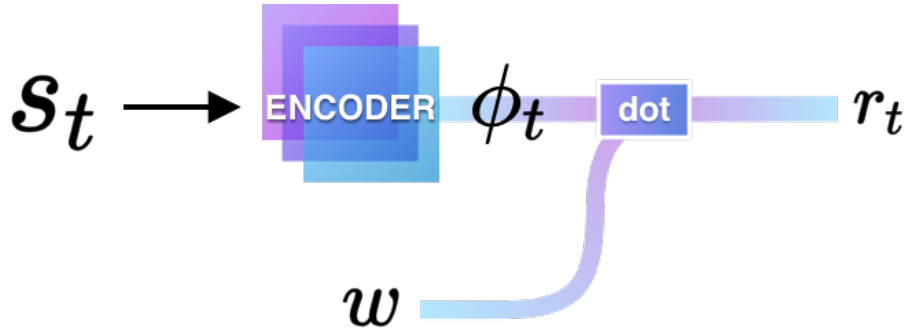
$$r_t = \phi_t^\top \mathbf{o} + \phi_t^\top \mathbf{A}\phi_t, \tag{6.1}$$

where $\{\phi_t, \mathbf{o}\} \in \mathbb{R}^z$ and $\mathbf{A} \in \mathbb{R}^{z \times z}$. Both $\mathbf{o}$ and $\mathbf{A}$ are learnable parameters modelling the reward structure of the environment. Note that w in $\phi_t^\top w$ is replaced by $\mathbf{o} + \mathbf{A}\phi$ and that this formulation introduces a non-linear transformation with respect to ϕ_t.

Following from the definition of the state-action value function $Q(s, a)$, and by also dropping the conditional portion of the expectation for brevity, the adjusted reward function can be



(a) S2F Model Overview



(b) Reward Prediction

Figure 6.1: **S2F Model Overview** a) The encoder transforms the raw state into an internal state representation ϕ_t. The state representation ϕ_t is used by the decoder, Λ_t, and ψ_t. The decoder tries to reconstruct the raw input s_t from the state representation ϕ_t. Λ_t and ψ_t produce one output per action, with the former predicting matrices and the latter predicting vectors. b) Reward prediction by dotting the current state ϕ_t, produced by the encoder, and the reward weight w.

substituted to yield

$$Q(s,a) = \mathbb{E}^{\pi}[\phi_{t+1}^{\top}\mathbf{o} + \phi_{t+1}^{\top}\mathbf{A}\phi_{t+1} + \gamma\phi_{t+2}^{\top}\mathbf{o} + \gamma\phi_{t+2}^{\top}\mathbf{A}\phi_{t+2} + \dots].$$

Because the expectation operation is linear, $\mathbf{o}$ can be pulled out from the first term:

$$Q(s,a) = \mathbb{E}^{\pi}[\phi_{t+1} + \gamma\phi_{t+2} + \dots]^{\top}\mathbf{o} + \mathbb{E}^{\pi}[\phi_{t+1}^{\top}\mathbf{A}\phi_{t+1} + \gamma\phi_{t+2}^{\top}\mathbf{A}\phi_{t+2} + \dots]. \tag{6.2}$$

By recognizing the first expectation term as the Successor Features ψ_t, (6.2) can be rewritten as

$$Q(s,a) = \psi_t^\top \mathbf{o} + \mathbb{E}^\pi[\phi_{t+1}^\top \mathbf{A}\phi_{t+1} + \gamma\phi_{t+2}^\top \mathbf{A}\phi_{t+2} + \dots].$$

Because $\phi_t^\top A\, \phi_t$ results in a scalar, the trace operator $tr(\cdot)$ can be used inside the right-hand term, and by exploiting the fact that $\mathbf{tr}(\mathbf{AB}) = \mathbf{tr}(\mathbf{BA})$, the terms inside the trace function can be swapped to yield

$$Q(s,a) = \psi_t^\top \mathbf{o} + \mathbb{E}^\pi[\mathbf{tr}(\mathbf{A}\phi_{t+1}\phi_{t+1}^\top) + \mathbf{tr}(\gamma\mathbf{A}\phi_{t+2}\phi_{t+2}^\top) + \dots].$$

Because both $tr(\cdot)$ and $\mathbf{A}$ are linear, they can be pulled out of the expectation, resulting in

$$Q(s,a) = \psi_t^\top \mathbf{o} + \mathbf{tr}(\mathbf{A}\mathbb{E}^\pi[\phi_{t+1}\phi_{t+1}^\top + \gamma\phi_{t+2}\phi_{t+2}^\top + \dots]).$$

Finally, the remaining expectation can be expressed as a function,

$$Q(s,a) = \psi_t^\top \mathbf{o} + \beta\mathbf{tr}(\mathbf{A}\Lambda_t), \tag{6.3}$$

where $\beta \in \{0, 1\}$ is a hyperparameter that controls the inclusion of the non-linear component.

The next step is to define the novel function Λ_t, which appears using the shorthand notation. The complete definition of Λ_t is given as:

$$\Lambda^\pi(s,a) = \mathbb{E}^\pi[\phi_{t+1}\phi_{t+1}^\top + \gamma\Lambda^\pi(s_{t+1}, a_{t+1}))|S_t = s, A_t = a], \tag{6.4}$$

with the $\Lambda^\pi(s,a)$ form indeed satisfying the Bellman equation [132]. Now, in addition to ψ_t, it is now necessary to also model Λ_t, which outputs an $\mathbb{R}^{z\times z}$ matrix *per* action. Note that the quantity $\phi_t\, \phi_t^\top$ can be interpreted as an auto-correlation matrix of learned state features.

6.2.2 Model Structure & Training

The proposed S2F model, shown in Figure 6.1(a), uses an encoder to produce a state embedding ϕ_t, which is consumed by downstream modelling tasks. As we are tightly constrained by the mathematical structure and assumptions of the SF framework, using models such as a LSTM[34] or Decision Transformers [107] are not possible. Figure 6.1(b) shows how the current reward r_t is predicted using w, with $w = \mathbf{o} + \mathbf{A}\,\phi_t$, and the current state representation ϕ_t; this process is defined in (6.1). As in previous studies with SF [76, 79, 120], the structure includes pathways for an encode-decode task and a SF prediction ψ_t. The decoder network ensures that the features learned by the encoder, which produces ϕ_t, contain useful information for prediction. Furthermore, only the gradients from the state-dependent and reward prediction tasks modify the encoder parameters, and therefore ϕ_t. An additional branch is added, by way of the non-linear reward function, to model Λ_t. This branch output is a matrix, which differs from the vector-predicting branches ψ_t and ϕ_t.

The encode-decode task is trained by minimizing the mean squared difference between the input s_t and the decoder's reconstructed version $\hat{s}_t = g(\phi_t; \hat{\theta}_\phi)$ from ϕ_t:

$$\mathcal{L}^d(s_t; \theta_\phi, \hat{\theta}_\phi) = [s_t - g(\phi_t; \hat{\theta}_\phi)]^2,$$

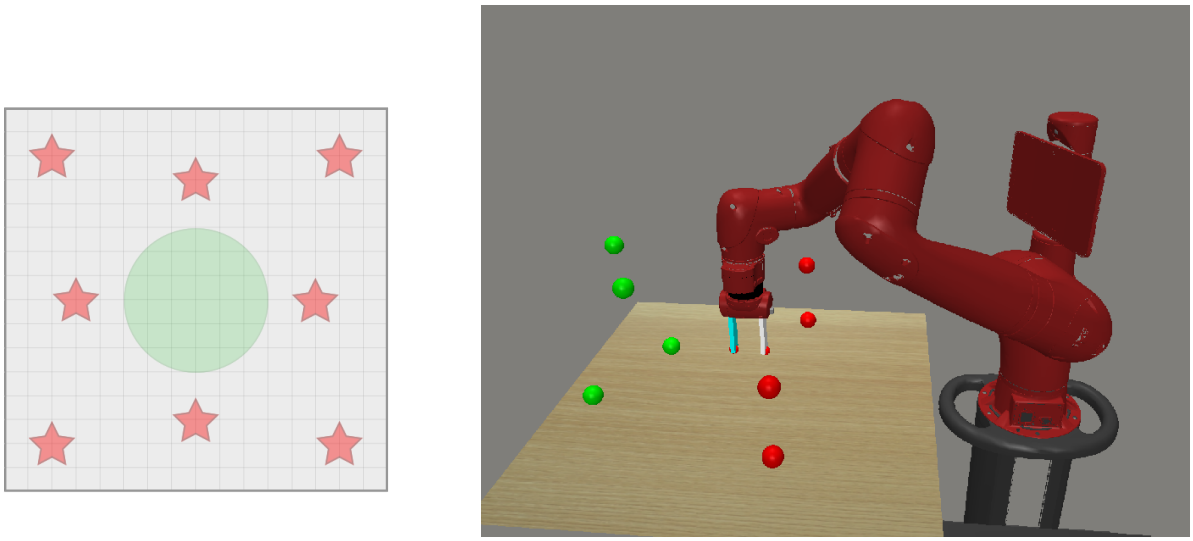
where ϕ_t is the output of the encoder with parameters θ_ϕ and $g(\phi_t; \hat{\theta}_\phi)$ produces the output of the decoder with parameters $\hat{\theta}_\phi$. As mentioned previously, we train ψ_t and Λ_t, parameterized with θ_ψ and θ_Λ respectively, using the Bellman equations to minimize the following losses:

$$\mathcal{L}^\psi(s_t, a_t; \theta_\psi) = \mathbb{E}[(\phi_t^- + \gamma\psi(s_{t+1}, a^*; \theta_\psi^-) - \psi(s_t, a_t; \theta_\psi))^2],$$

$$\mathcal{L}^\Lambda(s_t, a_t; \theta_\Lambda) = \mathbb{E}[(\phi_t^- \phi_t^{-\top} + \gamma\Lambda(s_{t+1}, a^*; \theta_\Lambda^-) - \Lambda(s_t, a_t; \theta_\Lambda))^2],$$

where $a^* = \max_{a^*} Q(s, a^*)$ is the optimal action at the current time step according to the state-action value function. To help stabilize learning, lagged versions of θ_ϕ, θ_ψ, and θ_Λ are used as

was done by Mnih *et al.* [6]; the lagged version is denoted by the − symbol in the exponent.



(a) Axes Environment.

(b) Reacher Environment.

(c) Doom Environment.

Figure 6.2: **Environments** a) A graphical representation of the Axes environment. The agent must traverse to various goal locations marked by the star symbol. The eight goal locations are split between training and testing. b) A rendering of the Reacher task. The agent controls the robotic Sawyer arm to move the end-effector to a 3D point in space. The eight goal locations are shown as balls. Training goals are in green, and test goals are in red. d) Images of the Doom environment where the agent must move between rooms looking for a goal location.

Unfortunately, as the dimensionality z of states and rewards grows, the number of parameters needed by Λ_t grows quadratically. However, by identifying the $\phi_t \phi_t^\top$ term in Λ_t as a symmetric matrix, it is possible to model only the upper triangular portion of the matrix[1], requiring about half the number of parameters. To further reduce parameters, each ψ_t and Λ_t pathway has two hidden layers before its outputs, as reflected in Figure 6.1(a). In this way, the parameters are shared amongst pathways, which contrasts with other works with multiple sets of layers per action a [79, 120]. To learn the reward parameters $\mathbf{A}$ and $\mathbf{o}$, which are the parameters of the approximated non-linear reward function, the following squared loss function is used:

$$\mathcal{L}^r(s_t; \mathbf{o}, \mathbf{A}) = [r_t - \phi_t^\top \mathbf{o} - \beta \phi_t^\top \mathbf{A} \phi_t]^2. \tag{6.5}$$

Note that (6.5) does not adjust the feature parameters involved in the prediction of ϕ_t. As to Λ_t, as the dimensionality of z increases, so does the number of parameters needed for modelling matrix $\mathbf{A} \in \mathbb{R}^{z \times z}$. Therefore, in the interest of reducing the number of parameters, a factorization is used that splits the matrix $\mathbf{A} \in \mathbb{R}^{z \times z}$ into two parts with a smaller inner dimension f, $\mathbf{A} = \mathbf{L} \cdot \mathbf{R}^\top$, where $\{\mathbf{L}, \mathbf{R}\} \in \mathbb{R}^{z \times f}$. By factoring the matrix in this way, $2 \times z \times f$ parameters are required instead of $z \times z$. If values for f smaller than $\frac{z}{2}$ are used, the number of parameters required by matrix $\mathbf{A}$ is further reduced. A similar factorization was suggested in the context of visual question answering [133, 134]. The factorization of $\mathbf{A}$ was primarily done to reduce the total number of parameters in the proposed model.

Finally, by combining all losses, the composite loss function is the sum of the four losses given above:

$$\mathcal{L}(\theta_\phi, \hat{\theta}_\phi, \theta_\psi, \theta_\Lambda, \mathbf{o}, A) = \mathcal{L}^d + \mathcal{L}^\psi + \beta \mathcal{L}^\Lambda + \mathcal{L}^r. \tag{6.6}$$

In practice, to optimize (6.6) with respect to its parameters, $(\theta_\psi, \theta_\Lambda)$ and $(\theta_\phi, \hat{\theta}_\phi), o, \mathbf{A}$ are iteratively updated. Doing so increases the stability of the approximations learned by the model and ensures that the branches modelling ψ_t and Λ_t do not backpropagate gradients to affect θ_ϕ

[1] It would still be necessary to manipulate this matrix so that it forms a full matrix.

[76, 120, 79]. In addition, by training in this way, the state representation ϕ_t can learn features that are both a good predictor of the reward r_t and useful in discriminating between states [79].

6.3 Experiments

This section examines Axes, a navigation task, Reacher, a robotic control task built using the MuJoCo engine [135], and the 3D maze *Doom* game engine. The environments are shown in Figure 6.2, and act as test beds for the S2F method, clearly showing that the second-order function provides additional representation capacity to the reward model. They each contain tasks specified by goal location and are split between training and test distributions, with the exception of Doom. The environments were chosen to guarantee comparability and task reproducibility within previous studies on SFs [128].

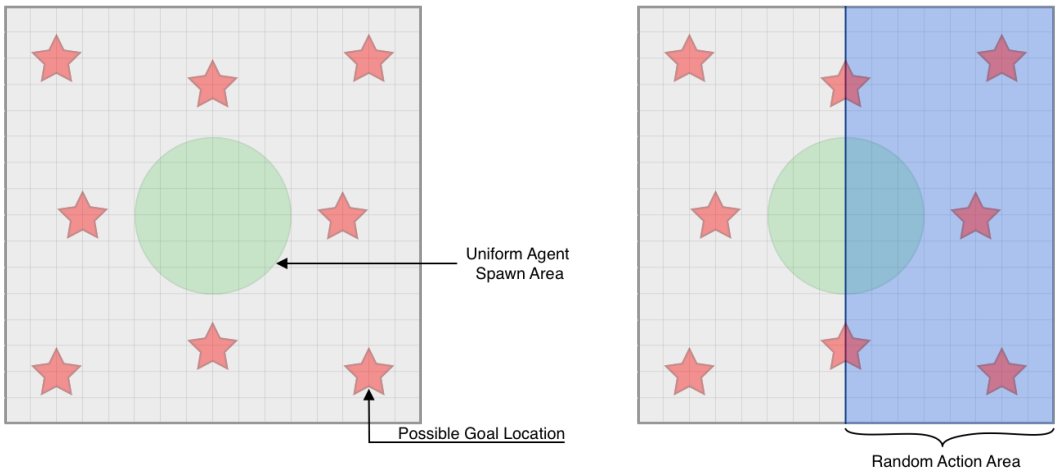


Figure 6.3: A graphical representation of the standard Axes environment, on the left, and its half-random variant on the right. The spawn location of the agent is uniformly sampled from within the green circle. The agent must traverse to a goal location, which defines the specific task, this is visualized by the star symbols. On the right hand side the half-random variant is shown, with the random action area shaded in blue. Within the random action area, the actions of the the agent are randomly perturbed by a fixed probability.

6.3.1 Axes

In this environment, shown in Figure 6.3, the agent, spawned at a uniform location within the green circle, must traverse the map to reach a goal location using four actions: *up*, *down*, *left*, and *right*. Eight separate goal locations exist. An episode ends when either the agent reaches the goal or more than 25 steps have elapsed. The agent's starting location is randomly sampled from a grid of 3×3 step units, centered at $(0, 0)$.

Within the Axes environment, two variants exist, *easy* and *hard*, where each refers to the difficulty of using the state for reward prediction. The reward function takes the form

$$d(p_a, p_{g_k}) = \sum_k \sqrt{\left(p_a^{(1)} - p_{g_k}^{(1)}\right)^2 + \left(p_a^{(2)} - p_{g_k}^{(2)}\right)^2 + \cdots + \left(p_a^{(n)} - p_{g_k}^{(n)}\right)^2}, \tag{6.7}$$

where $d(p_a, g_{g_k})$ is the distance between the agent position $p_a \in \mathbb{R}^n$ and each k^{th} goal position $p_{g_k} \in \mathbb{R}^n$ within the environment. For Axes, $n = 2$, meaning positions are in 2D.

In the *easy* variant, which is commonly used within other SF studies [128], the state ϕ_t is represented as the distance between the agent position p_a and each possible goal position p_{g_k} within the environment. Note that because the agent receives a reward equal to the negative distance between itself and the target goal at each step, reward prediction can be easily accomplished by using a 1-hot encoded reward vector. Therefore, the state s_t is a vector of distances between the agent and all eight available goals, such that $s_t \in \mathbb{R}^8$.

On the other hand, in the *hard* variant, the state is simply the location of the agent and the location of the current goal. Therefore, because the reward involves non-linear functions, a square root and square powers as in (6.7), the linear variant will have trouble modelling the reward of *hard* environments with a sub-optimal state representation. The state s_t is the agent's and the current active goal's 2D coordinates, such that $s_t \in \mathbb{R}^4$. With this state space the agent must learn a reward function that can approximate the distance between itself and the goal location, which is a non-linear function.

6.3.2 Reacher

The Reacher environment also serves as a test-bed to examine the representational property of the second-order function and represents a control task defined in the MuJoCo physics engine [135], as shown in Figure 6.2(b). This study has used a modified version of the robotic model provided by Metaworld [136], which has been chosen to show that the S2F model can scale to difficult control tasks. In this environment, the agent must move a simulated robotic arm to a specific 3D point in space by activating four torque-controlled motors.

As in the Axes environment, the *easy* and *hard* variants exist. In both variants, the reward function takes the form of (6.7), where for Reacher, $n = 3$, meaning positions are 3D. Therefore, in the *easy* variant, the state s_t is a set of distances between the agent and all available goals, such that $s_t \in \mathbb{R}^8$. While in the *hard* variant, the state s_t is the agent's and the current active goal's coordinates, such that $s_t \in \mathbb{R}^4$.

In the Reacher environment, an episode ends when 150 steps have elapsed or the top of the arm, controlled by the agent, is within 7cm of the goal. Again, the agent receives a reward equal to the negative distance between the end-effector and the current target goal at each step.

Because the models can be used only with discrete actions, we have discretized the actions such that the agent has nine discrete actions that control the arm's movements. Therefore, the four-dimensional continuous action space $\mathcal{A}$ was discretized using two values per dimension: the maximum positive and maximum negative torque for each actuator. An all-zero option was included that applies zero torque along all actuators, resulting in a total of nine discrete actions.

6.3.3 Doom

To demonstrate the general applicability of the S2F model to complex environments, we evaluate it on a 3D navigation task in the Doom environment from raw pixels. This task is challenging because the model must learn a state representation that is adequate for both reward prediction and for use in the SF branch from raw pixels.

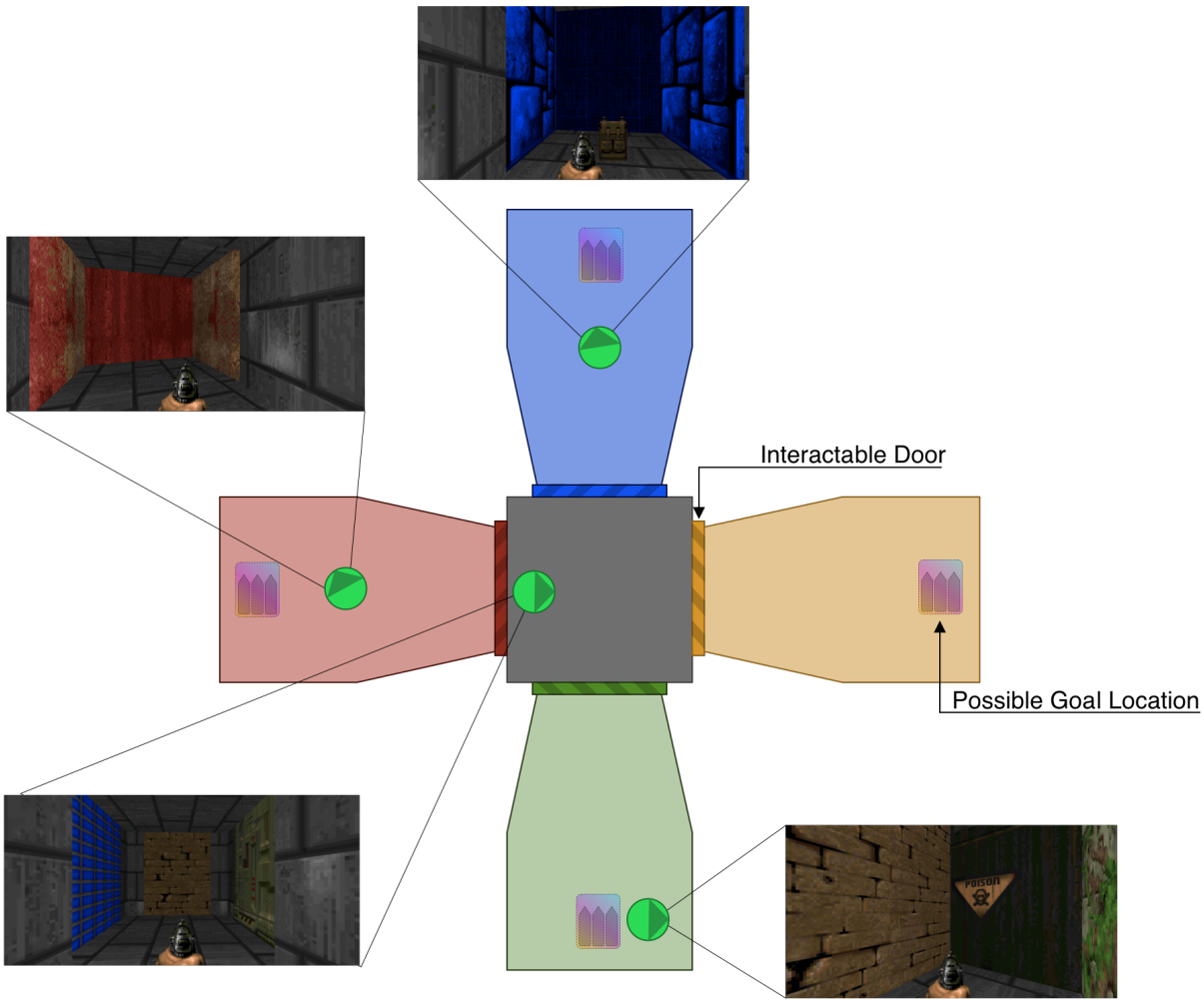


Figure 6.4: Overview of the Doom environment. Several agent views, with the agent shown as a green circle, throughout the map. The possible goal locations are marked with ammo boxes; while the corresponding interactable doors are marked as dark blocks leading to each colored room.

In this environment, shown in Figures 6.4, the agent must navigate among four rooms, trying to collect an item from one of the rooms, after which the episode ends. This setup and map are identical to that of Kulkarni *et al.* [79].

The rooms are separated by doors that the agent must manually open. At each step, the agent receives a small negative reward of -0.01, and upon finding the goal it receives +50. The agent perceives the state and the four stacked RGB frames of shape (3, 84, 84), corresponding to color channels, width, and height. The agent has four actions available: forward, rotate left,

rotate right, and activate door. One difference in the present implementation is that a consistent action repeat is used across all available actions; this differs from the original implementation, where a different action repeat was used on a per-action basis, e.g., forward did not repeat, but rotation movements repeated five times

6.3.4 Experiments

This section describes in detail the experiments used to validate and test the proposed S2F model deployed on Axes, Reacher, and Doom.

For both the Axes and Reacher environments, the primary set of experiments focused on examining and comparing the performance between the S2F model and baselines over the training distribution tasks. The eight available goals were split between training and test distributions. To strongly demonstrate our claims for the improved representation capacity of the second-order reward model, a weak encoder was purposely used for the Axes and Reacher environments, represented by a single hidden layer, which could learn only a sub-optimal state representation.

Because transfer to related tasks is a core benefit of the SF framework, we also evaluate how well the models transfer to unseen tasks from the test distribution on the Axes and Reacher environments.

Finally, we use a modified version of Axes, referred to as *half-random*, shown on the right side Figure 6.3, to assess and investigate the new second order term Λ_t. This version was identical in all aspects to the base version except for a location-based conditional probability that affects the agent's actions. If the agent was within the positive x quadrant of the map, $x > 0$, then actions are randomly perturbed with a fixed probability. Otherwise, they were fully deterministic. This experiment helped identify what Λ_t learns.

For the Doom environment, the primary set of experiments also examined the performance differences between the proposed S2F model and the baselines over the training distribution tasks. S2F method was compared with the SF framework with a linear reward model; the

architecture was similar to that of Kulkarni *et al.* [79]. This baseline was identical in all ways to the second-order model for $\beta = 0$ in (6.3) and (6.6). More precisely, the linear baseline and the proposed S2F model use the *exact same code* with *only* the β and z hyperparameters adjusted.

Because this experiment was solely intended to compare the performance of the second-order model and that of the linear baseline in the Doom environment, a train and test split was not used; instead, the experiment relied on the randomness of the item and agent locations. We follow by checking our assumption on the representation capacity of the S2F model by evaluating the performance of the linear agent with an increasingly stronger encoder. Finally, this environment was also used to understand the Λ_t term that appears after the derivation of the proposed S2F model. We examine the learned Λ_t function to understand if it captures environmental stochasticity and evaluate different guided exploration strategies using the Λ_t term on new tasks.

It is important to state that as the S2F model extends the SF framework, the focus remains on keeping the scope of comparisons to baselines, and that ϵ-greedy approach was chosen, because it is robust and presents few confounding factors.

6.4 Results & Discussion

6.4.1 Performance

The primary point of comparison was between the proposed S2F model and the original formulation of the *Successor Feature* framework, which can be recovered by setting $\beta = 0$ in (6.3) and (6.6) of the proposed S2F model. The result of these experiments are shown in Table 6.1 for both the Axes and Reacher environments, and reported as the average rewards over the last 1000 steps of training. Note that, in both environments, the *easy* task is solved by both the linear and second-order variants. This is expected because the reward, in the case of the linear variant, can be recovered using a 1-hot encoded reward vector, and even with a weaker state

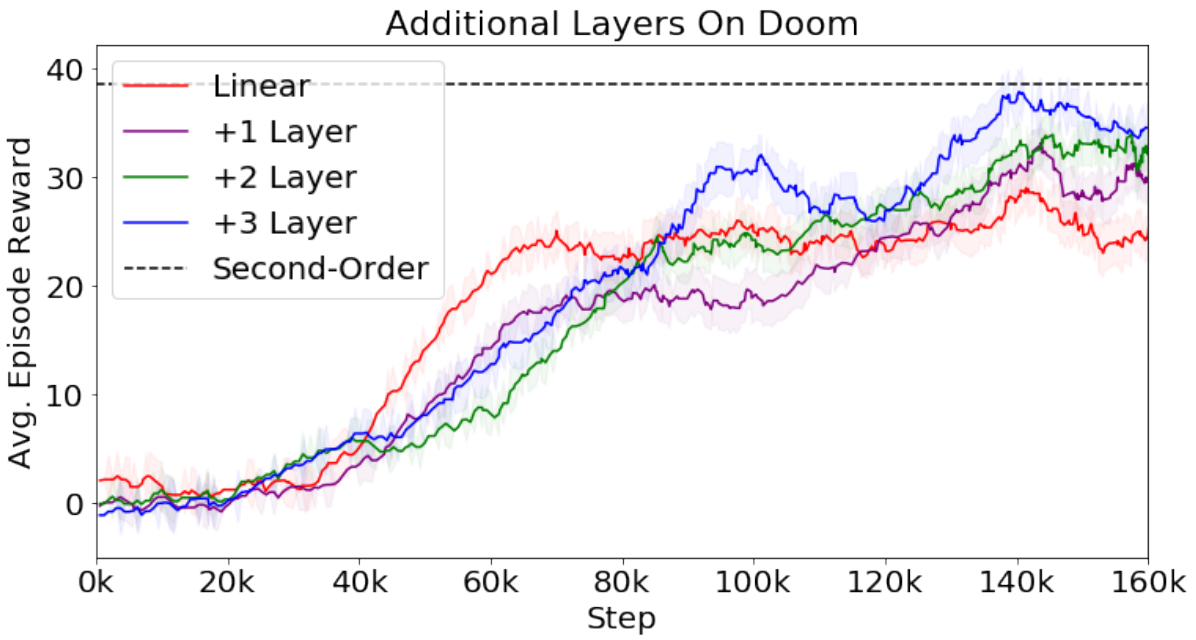


Figure 6.5: Performance of the baseline linear variant and variants of it where we gradually increase the expressive strength of the state model. The average performance of the S2F model over the last 1000 steps of training is included as a horizontal dashed line. The linear model has a total of 343M parameters while the S2F model only has 276M, approximately 20% fewer parameters, while showing stronger performance.

representation the linear reward model can learn to accurately predict future reward. However, this does not hold true for the *hard* task, because the linear variant has weaker performance than the second-order model.

Clearly, the second-order variant provides extra representational capacity to the reward model so that it can compensate on its own for a non-ideal state representation, which is shown by its better performance on the *hard* tasks. One explanation for this is that the linear variant cannot appropriately model the reward structure because, the reward is a non-linear function of the state, in this case the agent's coordinates and the current goal location.

The earlier experiments revealed that the features learned by the encoder are insufficient for reward prediction. Therefore, we examine the performance of the model in the Doom environment with additional encoder layers [2]. As shown in Figure 6.5, the additional encoding layers do indeed help improve the performance of the base model. However, even with three

[2]Each additional layer has the same number of parameters with z hidden units

	Axes				**Reacher**			
	Easy		Hard		Easy		Hard	
Model	Train	Transfer	Train	Transfer	Train	Transfer	Train	Transfer
Random	-3.43	-3.43	-3.43	-3.43	-78.48	-78.48	-78.48	-78.48
Linear SF	**-1.28**	**-1.29**	-1.58	-1.61	**-6.29**	**-6.25**	-10.25	-10.15
Second-Order SF	-1.3	-1.45	**-1.47**	**-1.48**	-6.31	-6.33	**-6.87**	**-7.29**

Table 6.1: Performance (average rewards) in the Axes and Reacher environments during training and transfer over the last 1000 steps. Both variants could solve the *easy* environment with essentially equal performance. In the *hard* environment the second-order model had higher performance than the linear model and was much closer to the *easy* variant score.

additional layers it could not quite match the S2F model we propose. This implies that the second-order reward model uses parameters with greater efficiency than the linear model.

From the result within the Doom environment, as shown in Figure 6.6, it is clear that the proposed S2F model has outperformed the baseline SF implementation. Indeed, S2F was near the ceiling performance of the environment, and converged rapidly. In comparison, the baseline method has failed to achieve similar performance and converged at a much slower rate. From the difference in learning curves, one can conclude that the extra representational power of the reward model in S2F has a dramatic impact on performance. Because both variants have roughly equal parameters, but with the linear variant containing more, one can conclude that the extra representational power is of better use in the reward component of the model than in the encoder.

6.4.2 Task Transfer

An important property of the SF framework is the ability to adapt rapidly to new tasks within the same environment. In this study, transfer to new tasks was performed only after training has been completed. Adaptation, or transfer, is accomplished by freezing the model's state-dependent components, such as ψ_t, and quickly learning just the reward parameter w [128, 79].

In practice, this is accomplished by training the reward vector w on rewards from a new out-of-sample task using the loss in (6.5), but with respect to the parameters $\mathbf{o}$ and $\mathbf{A}$ only. Doing so ensures that the other model parameters, such as those involved in predicting the SFs,

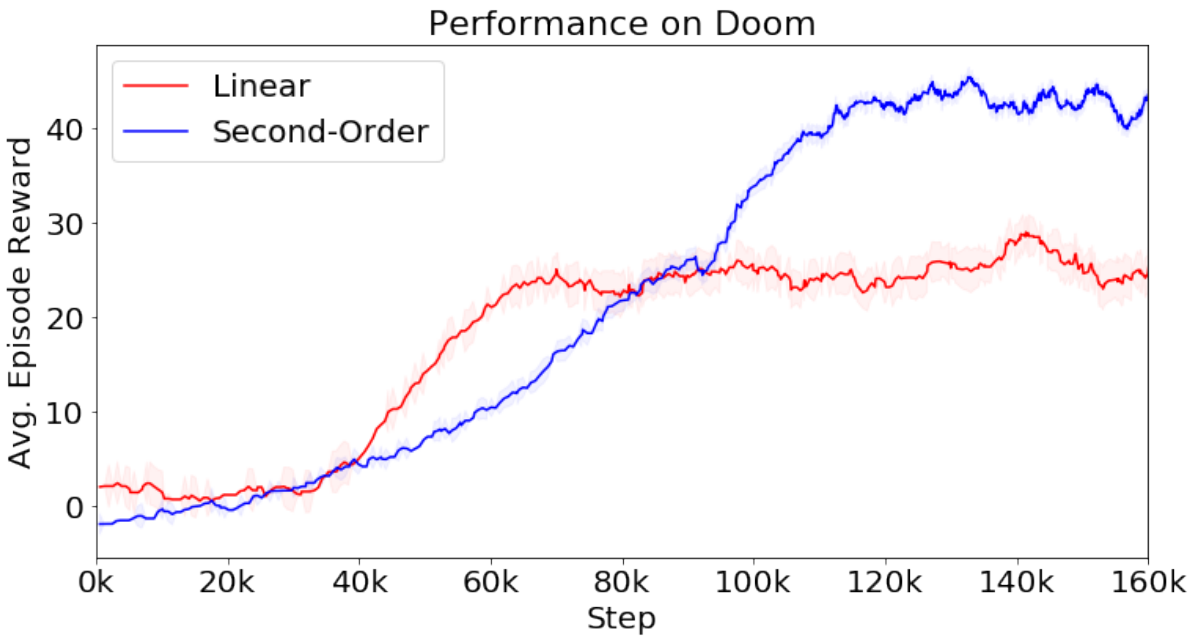


Figure 6.6: Performance of the baseline linear variant and the S2F model in the Doom environment. Each model's mean performance is given as the average over three runs with varied seeds. Standard deviation over all runs are represented as a shaded area. S2F model has outperformed the baseline SF implementation and converged rapidly, while the baseline method has failed to achieve similar performance and converged at a much slower rate.

are not updated, but instead only vector w is updated – enabling fast task adaptation.

We evaluated the transfer performance of the proposed S2F model within the Axes environment on the hard task configuration. The environment was chosen because it was easy to generate and switch between training and test tasks after a certain number of training steps had been accrued. Table 6.1 includes a summary of the transfer results and Figure 6.7 provide learning curves during training and transfer.

Figure 6.7 shows that after the abrupt change in tasks, signified by the dashed vertical line, both baseline and S2F model performance changed sharply. Both models reached previous levels of performance on the new tasks and did so at a faster rate than their original training periods, doing so in under 100k steps. This implies that the models can leverage previously learned information as transfer to the new task occurs at a faster rate than in the original training phase. Furthermore, the additional parameters involved in the second-order reward structure

did not hinder learning speed or performance and we see might provide a stronger basis for transfer; as after the abrupt task change we see the S2F model had higher initial performance than the linear variant.

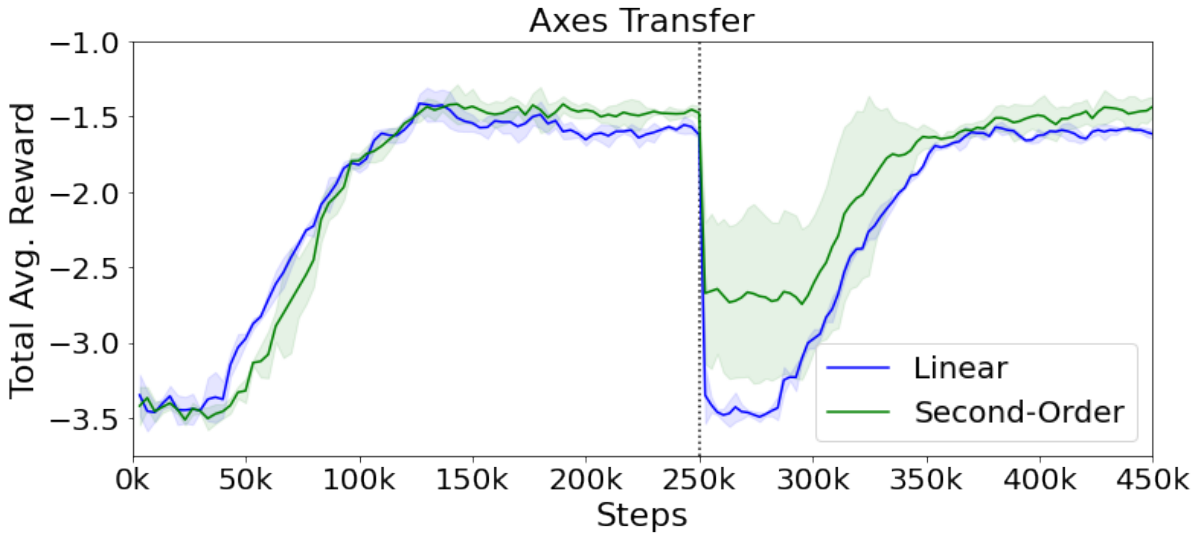


Figure 6.7: Transfer performance on the Axes environments. Each model's mean performance is reported on all plots as the average over three runs with varied seeds, which also includes the standard deviation over all runs as a shaded area. The additional parameters of the S2F model do not hinder its ability to transfer. The models larger capacity appears to give a stronger prior base, such that after abrupt task changes it retains more of its previous performance abilities.

6.4.3 Λ

After training to convergence on the Axes half-random variant, we examine the Λ_t function that the model has learned. From (6.4) it is known that Λ_t is trained to approximate the discounted future states of $\phi\,\phi^\top$, which is the auto-correlation matrix of the learned state features. According to the definition of the auto-correlation matrix, the diagonal terms in Λ_t are the correlations between states, which is shown in Figure 6.8. Throughout this work, we noticed that this new term might provide a powerful mechanism to represent the stochasticity of an environment.

From Figure 6.8, the visualizations of Λ_t on both variants of Axes show significant differences. In the case of no additional randomness, that is the top row of Figure 6.8, the Λ_t value

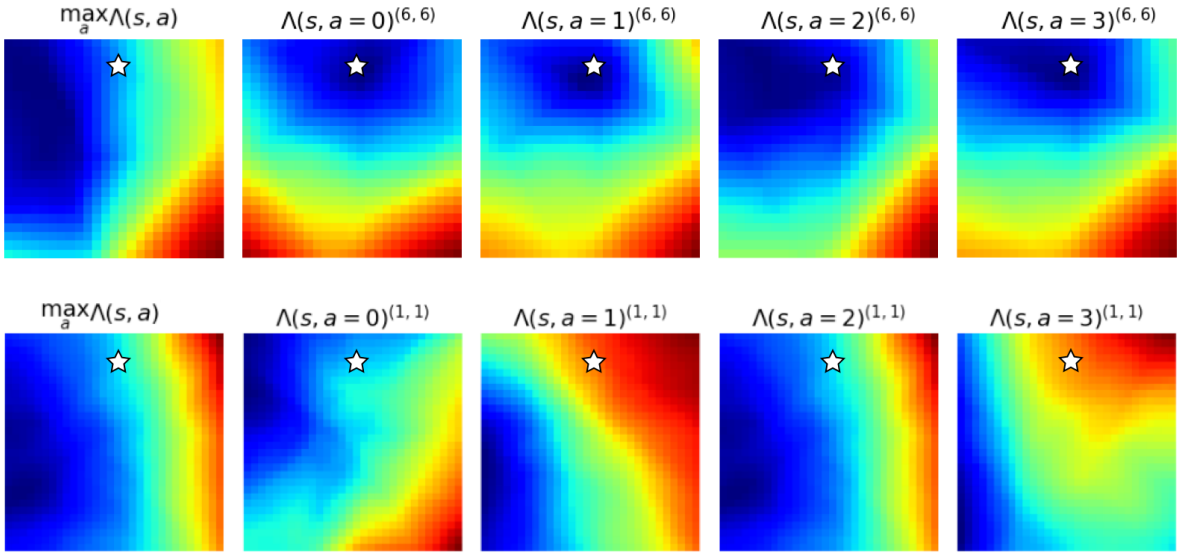


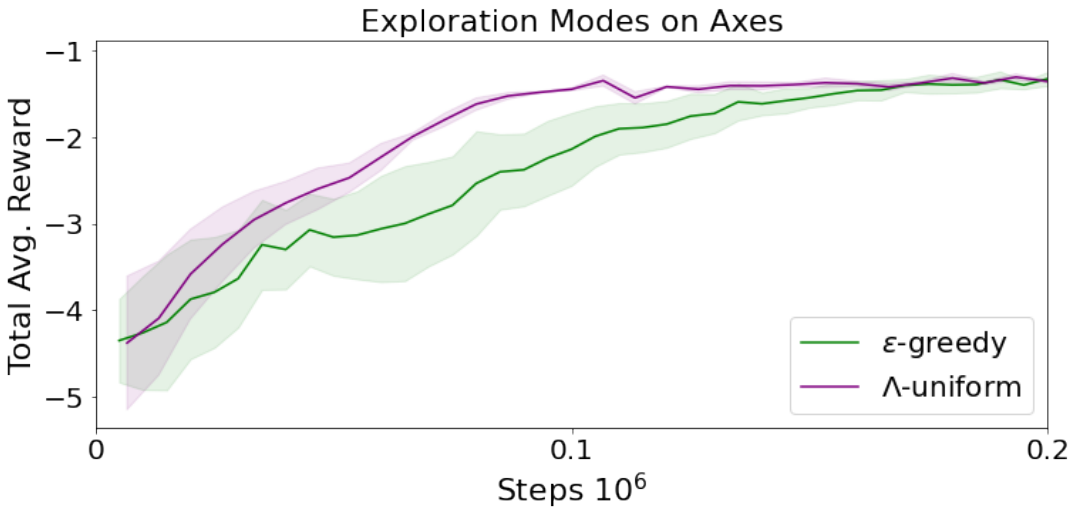
Figure 6.8: The learned expected future correlation of a feature along Λ_t's diagonal is visualized over the entire state space in the Axes environment. Blue and red signify the minimum and maximum values seen. The goal the agent was trained to find is shown as a star. The first column is the max value of Λ_t over the actions. The remaining columns, from left to right, correspond to each action: *left*, *up*, *right*, and *down*. The *top row* is the Λ function learned over the standard Axes environment with no randomness. The *bottom row* is the Λ function learned over the half-random Axes environment variant.

has captured that there is lower volatility nearer to the goal and higher volatility away from the goal. This is consistent across all possible actions. In comparison, the second row of Figure 6.8 shows the learned Λ_t value under random actions over the right side of the map. From the bottom row of 6.8 we can clearly see the higher volatility areas, that is the "hotter" colors, are biased to the area of Axes with high probability of random actions. This implies that the Λ_t value is able to model the stochasticity of the environment to some degree.

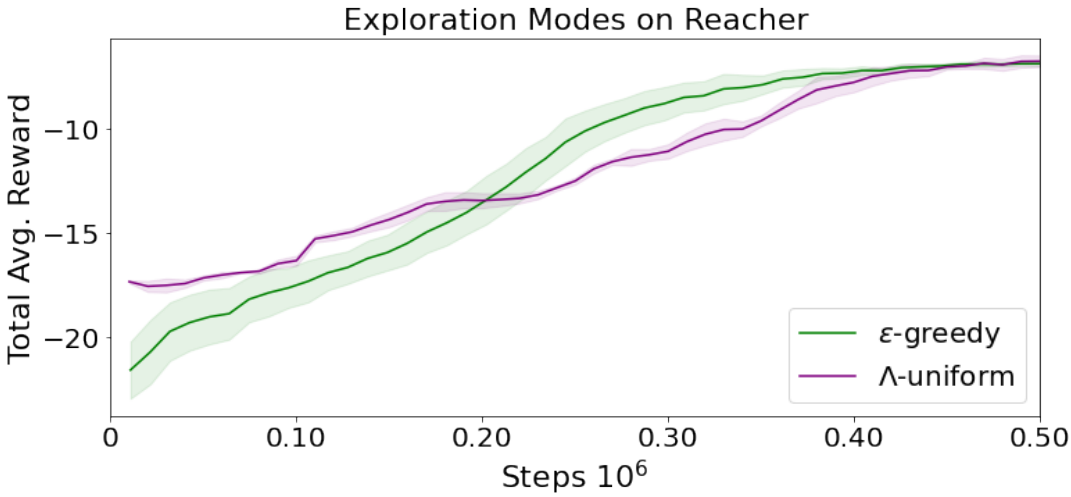
6.4.4 Guided Exploration With Λ

Here we examine whether it is possible to use the Λ-function for guided exploration during transfer within the Axes and Reacher environments.

The SFs can be interpreted as predicting the future expected path taken by the policy π in an environment. Under this interpretation, ψ can be seen as capturing the expected features of



(a) Axes Environment.



(b) Reacher Environment.

Figure 6.9: Guided Exploration: The Λ component of the proposed model is used to guide exploration during transfer. By using Λ the agent explores in directions with large variance in the state space.

the states and Λ the expected variance between state features along these pathways. Adding noise to the Λ component would then perturb around the expected path. Therefore, instead of using ϵ-greedy exploration, it is possible to add noise to Λ during transfer, such that $\hat{\Lambda}(s,a) = \Lambda(s,a) + \epsilon\Lambda(s,a)$, where ϵ is sampled from some distribution. During learning, the variance of the sampling distribution, controlled by α, can be annealed to some final value. The actions are then sampled from the model at time t as:

$$a_t = argmax_{a^*}\{\psi(s_t, a^*)^\top o + tr(\mathbf{A}\hat{\Lambda}(s_t, a^*))\}$$

From Figure 6.9, we see that using Λ for guided exploration is indeed a possible alternative to ϵ-greedy. Additionally, we found that using a scalar value sampled from uniform noise, that is $\epsilon \sim \mathcal{U}(-\alpha, \alpha)$ where $\epsilon \in [-\alpha, \alpha]$, provides the best performance.

6.5 Summary

In this chapter, a novel formulation for the SF framework with a second-order non-linear reward function has been derived, which predicts rewards as a non-linear combination of state features. Experimentally, we have shown that the agent can perform well with the second-order reward structure, providing extra flexibility to the reward model and empowering state representation and policy transfer. Furthermore, the new second-order term Λ_t has been investigated and confirmed to capture environmental dynamics closely. This result has significant implications, given that the state representation within the original SF framework may not have enough representational power for good state and reward reconstruction tasks, for example, in highly complex environments.

Chapter 7

Dynamic Successor Features

This chapter introduces the Dynamic Successor Features (DynSF) model, a formulation of the SF framework in which the RL problem can be disentangled as two supervised learning problems greatly increasing flexibility. This chapter builds on the published work [137], and contributes to improving task transfer efficiency in DRL.

The chapter is arranged as follows in the sections below. Section 7.1 provides an introduction to the chapter, DynSF is introduced in Section 7.2, a brief review of the environments and experiments used to validate the DynSF model is presented in Section 7.3, Section 7.5 contains experimental results supporting the flexible representation of the DynSF model and a discussion of these results, and finally the chapter concludes in Section 7.6 with a final discussion on the contributions of this research and possible avenues for future work.

7.1 Introduction

The performance of RL, which consists of learning through goal-oriented interactions of an agent with the environment, has improved by leaps and bounds over the past few years, outperforming human experts on gaming tasks such as Go [36], Poker [138], and Atari [139, 41]. RL has also been successfully used to tackle problems in robotics [140, 141], fluid mechanics [142, 143], energy [144, 145], chip placement [146], and other areas, extending to applications

outside the computer science and engineering fields.

A central question in RL is the transfer of knowledge between tasks in an environment when only the reward specification changes, but other environment characteristics remain fixed. The SF framework presented by [128] computes a representation of the environment that can be transferred across different reward functions. In the SF framework, the state-action value function is expressed as the dot product between a vector of expected discounted future-state occupancies (the Successor Features) and another vector representing the immediate reward in each of those successor states [77, 147, 148, 19, 76, 120, 121, 78, 122, 123, 124]. During transfer, the SFs are held frozen, and only the parameters from the reward component are trained, which requires fewer samples.

The SF framework was developed to tackle task transfer, but also brings light to the mathematical formulation of RL by partially disentangling the environment from task-related rewards. By taking advantage of the SF model, this chapter explores the SF framework's mathematical formulation to facilitate understanding of what is learned. The SF framework formulation has been extended to yield the Dynamic Successor Feature (DynSF). The process of doing this extension sheds light on what the original SF framework learns, showing mathematically that the original SF representation is tied to marginally learning a specific policy, which governs the SFs under a fixed discount factor.

In fact, after training, the SFs are tied to the optimal policy learned. However, when transferring to another task, the ongoing optimal policy might not be the same [149]. For the SF framework, a successful transfer depends on how drastically the reward function changes between tasks. If a slight change in rewards occurs, then the optimal policy change is not drastic, and relearning can converge within a good response time. Otherwise, if a significant change occurs, then the original SF framework can indeed fail to transfer.

In particular, it will be shown that through DynSF, SFs can be modelled using a state-transition model. In this newly proposed model, by using rollouts of state embeddings that are learned and output by the state-transition model, to induce a policy, it is possible to dynamically

adjust an agent. This induced policy and the discount factor can be parameterized on the fly during a rollout. Dynamic adjustment of the discount factors and acting policy is possible because they are treated as functional parameters of the state-transition model. This novel formulation makes it possible to use different policies and discount factors during training and testing.

The proposed approach can be related to model-based RL, where an environment model is learned to improve action efficiency. In such model-based approaches, action efficiency is improved by using the environment model to simulate future actions and optimize the chosen action [10, 12, 150]. The DynSF framework proposed in this study is most similar to this action planning approach, but with the induced policy and discount factor being dynamically created, which brings flexibility and efficiency to training and transfer. The DynSF framework combines supervised learning of a state-transition model and the reward component, enabling several supervised machine learning techniques. As a result, the DynSF formulation improves the fidelity of the captured state dynamics over those of SFs through a more expressive state-transition model. Now, novel algorithms for supervised learning can be used to learn the rewards correctly. The aforementioned flexibility in the use of different techniques and dynamic adjustment of parameters during learning the SFs helps illuminate another central discussion to RL: the importance of correctly modelling the rewards. In [130], the authors claim the hypothesis that intelligence arises from reward maximization. Therefore, it is critical for learning agents to well model task-specif rewards. In the original SF framework, the encoder is often trained to output a dense vector representation of the state that helps minimize the least-squared error loss for both state reconstruction and reward prediction [79, 128, 81, 122]. If the encoder learns a sub-optimal state representation for reward prediction, then the reward model may be unable to compensate in terms of reward prediction, given the limited set of model parameters. In contrast, the DynSF representation expands the SFs through a state-transition model, where more degrees of freedom can be adjusted during training and novel algorithms for supervised learning can be used to learn the rewards correctly.

The contribution of this chapter can be summarized as follows:

- DynSF enables a state-transition model to be used for state rollouts where the discount factor and policy can be set on the fly, which is advantageous compared to common model-based approaches.
- The chapter examines how rollout length and policy affect the performance of DynSF to compare its performance to the original framework and baselines.
- Through experiments, the flexibility of DynSF is analyzed, where policy and discount factors can be parameterized dynamically and where different supervised ML algorithms can be used to learn the state transition model.
- A significant task change is used to show that DynSF performs better during task transfer than SF. DynSF's learned state model is shown to offer a lesser bias than the previously learned policy.

7.2 Model

This section presents the proposed variant of the SF framework, which uses a state-transition model to dynamically bootstrap the SFs.

7.2.1 Model Architecture

The definition of the SFs ψ_t^π is as follows:

$$\psi_t^\pi = \mathbb{E}^\pi[\phi_t + \gamma\,\phi_{t+1} + \gamma^2\,\phi_{t+2} + \dots], \tag{7.1}$$

where the visited latent states within the expectation depend on the particular policy π. From this, (7.1) can be rewritten in terms of a chain of state transitions dictated by the stated policy. Defining (7.1) in terms of state transitions requires access to a model of the environment that

can predict the next latent state ϕ_{t+1} given the current state ϕ_t and an action a. The assumption is made that ϕ_t, and therefore the reward r_t, are predicted solely from the current state s_t. This study refers to $\mathcal{M}$ as the latent-state transition model and defines it as:

$$\mathcal{M} : (\phi_t \in \Phi) \times (a_t \in \mathcal{A}) \mapsto (\phi_{t+1} \in \Phi),$$

where Φ is the set of all encoded valid states in $\mathcal{S}$. Following this definition, (7.1) can now be rewritten with the latent state-transition model $\mathcal{M}$ as follows:

$$\psi(s, a, \bar{\pi}, \gamma) = \phi_t + \gamma \mathcal{M}(\phi_t, \bar{\pi}(\phi_t)) + \ldots, \tag{7.2}$$

where ψ_t^π now loses the superscript π because it accepts additional arguments such as any policy $\bar{\pi}(\phi_t)$, working on the latent space representation, and the discount factor γ. Rewriting (7.2) with a summation and a k-length rollout yields:

$$\psi(s, a, \bar{\pi}, \gamma) = \phi_t + \gamma \mathcal{M}(\phi_t, \bar{\pi}(\phi_t)) + \sum_{j=1}^{k-1} \gamma^{j+1} \mathcal{M}(\hat{\phi}_{t+j}, \bar{\pi}(\hat{\phi}_{t+j})) \tag{7.3}$$

where $\hat{\phi}_{t+1} = \mathcal{M}(\phi_t, \bar{\pi}(\phi_t))$ and $\hat{\phi}_{t+j+1} = \mathcal{M}(\hat{\phi}_{t+j}, \bar{\pi}(\hat{\phi}_{t+j}))$. Equation (7.3) implies that the SFs can be computed dynamically given a latent state-transition model, a discount factor γ, and a policy $\bar{\pi}$. The SF function $\psi(s, a, \bar{\pi}, \gamma)$ is bootstrapped before each step in the environment by gradually unrolling the latent space with $\mathcal{M}$, as shown in (7.3). Comparing (7.1) and (7.3) reveals that the original function formulation of ψ_t^π we see it is learning an amortized version of this rollout under a specific policy $\pi = \bar{\pi}$ and discount factor γ.

The final Q-Learning function can now be dynamically inferred based on $\bar{\pi}$ and the discount factor γ at each roll-out by $\mathcal{M}$ for k steps:

$$Q(s, a, \bar{\pi}, \gamma) = \{\phi_t + \gamma \mathcal{M}(\phi_t, \bar{\pi}(\hat{\phi}_t)) + \sum_{j=1}^{k-1} \gamma^{j+1} \mathcal{M}(\hat{\phi}_{t+j}, \bar{\pi}(\hat{\phi}_{t+j}))\}^\top \cdot w. \tag{7.4}$$

This contrasts with the traditional phrasing of the SF framework, and some important considerations of this new formulation are made next. It is now clearly shown that the RL problem can be cleanly decomposed into two supervised learning tasks without explicit need for temporal difference learning.

Rollout Length

Given the DynSF formulation, note that the original SF function learns to combine information about the policy, discount factor and state transitions in some marginalized form. It is possible to claim a mathematical equivalence between the original SF framework and DynSF, given that an equivalent fixed policy π is used throughout the rollouts in (7.4), and that the number of rollouts matches the task state horizon. Although DynSF brings flexibility, the chosen rollout length certainly affects performance, as shown and discussed in Section 5.

Rollout Policy

When the SF function $\psi(s, a, \bar{\pi}, \gamma)$ is being dynamically computed, as described in (7.3), an acting policy $\bar{\pi}$ is required. The policy can be passed in as a function argument, providing immense flexibility. An experiment involving the deployment of different acting policies is discussed in Section 5. This can have significant implications, for example, for guided and safe exploration/ At its core is an optimization formulation in which the constraints restrict attention to a subset of safe policies and state visitations with some well-controlled probability [151]. However, because the final policy is derived from $\psi(s, a, \pi, \gamma)$ after k iterations and is dependent on the *current* state, it cannot be used during rollouts.

Therefore, a locally defined policy, such as random, greedy, or ϵ-greedy based on the reward component w, is used instead. Where *local* refers using the information available in the current state. Here, a locally greedy policy selects the state with the greatest predicted reward using: $\phi_t^\top w$.

7.2.2 Model Learning

The proposed model is composed of the following parts: an encoder $\mathrm{E}(s) : S \rightarrow \Phi$, a decoder $\mathrm{D}(\phi) : \Phi \rightarrow S$, a state-transition model $\mathcal{M}(\phi, a)$, and a reward vector w.

The state-transition model is trained to predict the next latent state ϕ_{t+1} given the current state ϕ_t and action a_t and minimizes:

$$\mathcal{L}_{\mathcal{M}} = \mathbb{E}[(\phi_{t+1} - \mathcal{M}(\phi_t, a_t))^2],$$

where the target variable ϕ_{t+1} is produced by encoding the following state. State grounding, as used by [10], is used to ensure that the latent representation ϕ_t contains meaning about the environment and fits naturally into the framework. The reward prediction step keeps the same form used in the SF framework, which minimizes

$$\mathcal{L}_r = \mathbb{E}[(r_t - \phi_t^\top w)^2].$$

There is also a reconstruction loss where the model must learn to reconstruct the state s_t from a latent representation ϕ_t using the decoder $D(\phi_t)$:

$$\mathcal{L}_{ED} = \mathbb{E}[(s_t - D(\phi_t))^2].$$

By summing the losses together, the following sum of mean-squared error losses is minimized:

$$\mathcal{L} = \mathcal{L}_{\mathcal{M}} + \mathcal{L}_r + \mathcal{L}_{ED}.$$

Unlike [147], all model components together are minimized at once and end-to-end instead of requiring interleaved updates.

Because the training formulation is quite general, any state-transition model can be dropped into place with its respective losses. This implies the flexibility to use any supervised learning

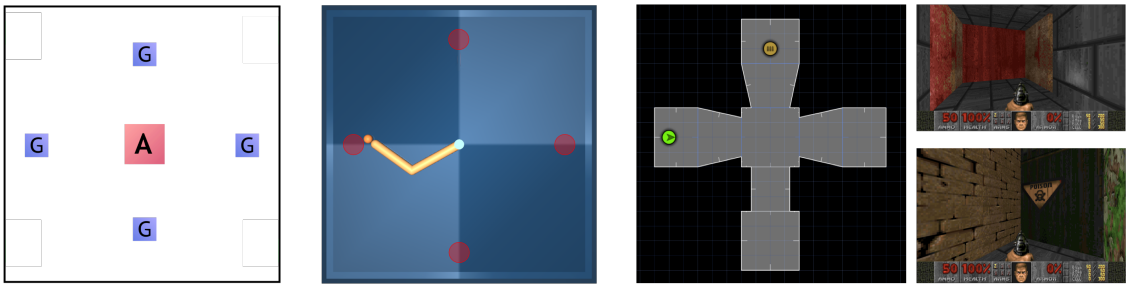


Figure 7.1: Visualizations of the environments used in this study. From left to right: Axes environment, Reacher, Doom. **Axes (left):** The agent shown in red must traverse to the various goals shown as colored "G"s. **Reacher (center):** The agent controls the robotic arm shown and must move the arm's end to a goal position shown by the colored circles. **Doom (right):** The agent must traverse between rooms looking for a goal point. The map layout is shown on the right, and images of the environment are on the far right.

model, which is considered, together with the acting policy and discount factor flexibility on the fly, to be one of the advantages of this new formulation compared to other model-based learning approaches.

7.3 Environments

This section provides the reasoning behind the choice of the environments.

The environments were chosen to guarantee comparability and research reproducibility within previous studies on SFs [128]. The following subsection describes Axes, a navigation task, Reacher, a robotic control task built using the MuJoCo engine [135], and the 3D maze *Doom* game engine. Figure 7.1 shows these environments, which act as test beds for the DynSF method. Each contains tasks specified by goal location and split between training and test distributions.

7.3.1 Axes

In this environment, shown in Figure 7.1, the agent, spawned at a uniform location, must navigate the map to reach a goal location in as few steps as possible by moving in each of the

cardinal directions: *up*, *down*, *left*, and *right*. Four separate goal locations exist. An episode ends when either the agent reaches the goal or moves more than 25 steps in the environment. The agent's initial position is randomly placed in a grid of 3×3 step units, centered at $(0, 0)$.

In the commonly used version of this environment also used within other SF studies [128], the reward the agent sees is the negative squared distance between its location and the target goal. The reward function takes the form

$$-d(p_a, p_g) = -\sqrt{\left(p_a^{(1)} - p_g^{(1)}\right)^2 + \left(p_a^{(2)} - p_g^{(2)}\right)^2 + \cdots + \left(p_a^{(n)} - p_g^{(n)}\right)^2}, \tag{7.5}$$

where $d(p_a, p_g)$ is the distance between the agent position $p_a \in \mathbb{R}^n$ and the goal position $p_g \in \mathbb{R}^n$ within the environment. For Axes, $n = 2$, meaning that positions are in 2D.

Note that the agent receives a reward equal to the negative distance between itself and the goals at each step. The state space of the environment is $s_t \in \mathbb{R}^4$, and reward prediction can be accomplished by using a 1-hot encoded reward vector.

Given the 2D characteristic, this environment provides visual access to the state-value over the entire state space to understand how the agent perceives the environment.

7.3.2 Reacher

This environment consists of a robotic control task defined in the MuJoCo physics engine [135], as shown in Figure 7.1 (center). The agent must move a robotic arm to a specific point in space by activating four torque-controlled motors. Because this is a continuous control task, the actions were discretized so that the agent had nine discrete actions to control the arm movements. Therefore, the four-dimensional continuous action space $\mathcal{A}$ was discretized using two values per dimension: the maximum positive and maximum negative torque for each actuator. An all-zero option that applies zero torque along all actuators was included, resulting in a total of nine discrete actions.

As in the Axes environment, the reward function takes the form of (7.5), where for Reacher,

$n = 3$, meaning that positions are 3D. Therefore, the agent receives a reward equal to the negative distance between the end-effector and the target goal. Hence, the state s_t is a set of distances between the agent and the target goals, such that $s_t \in \mathbb{R}^4$. In the Reacher environment, an episode ends when 150 steps have elapsed or the top of the arm, controlled by the agent, is within 7cm of the goal.

7.3.3 Doom

Doom is a 3D navigation task, shown in Figure 7.1 (right), where the agent must navigate among four rooms, trying to collect an item from one of the rooms, after which the episode ends.

Doors separate the rooms, which the agent must manually open. At each step, the agent receives a small negative reward of -0.01, and upon finding the goal, it receives +50. The agent perceives the state and the four stacked RGB frames of shape (3, 84, 84), corresponding to color channels, width, and height. The agent has four actions available: forward, rotate left, rotate right, and activate door.

The episode ends when the agent reaches the item or exceeds 1250 steps. This setup and map are identical to that of [79], except that a consistent action repeat was used across all available actions; this differs from the original implementation, where a different action repeat was used on a per-action basis, with some actions repeating once only and others several times, e.g., forward movements did not repeat, but rotation movements repeated five times.

7.4 Experiments

This section provides details on the environments against which the proposed methodology was tested, the reasoning behind the choice of the environments, and a description of the experimental methodology.

This section describes the experiments used to validate and test the DynSF model on Axes,

Reacher, and Doom. Based on the original SF framework, all baseline models used in the experiments had encoder, decoder, reward vector, and successor feature components, whereas the DynSF agents had an encoder, decoder, reward vector, and state-transition model.

7.4.1 Rollout Policy

Through the rollout policy experiment, the model's flexibility in using different policies for state visitation was examined. An evaluation of the different rollout policies used during the creation of $\psi(s, a, \bar{\pi}, \gamma)$ was performed, as defined in (7.3). Specifically, greedy, random, and ϵ-greedy policies were evaluated in the Reacher environment.

This simple experiment supports a more comprehensive use of flexible rollout policies for tasks such as safe RL and others on model-based environments.

7.4.2 Rollout Length

As the DynSF's policy was created dynamically with a variable rollout, the impact of the rollout length on the dynamic $\psi(s, a, \bar{\pi}, \gamma)$ was investigated. These experiments evaluated rollout lengths of $\{0, 2, 4, 8\}$ in the Axes environment. The aim was to understand how the hyperparameters affect the state-value function and whether a shorter rollout length can be used.

This analysis was performed using two variants of DynSF. The first variant used a fixed a priori state-transition model $\phi(s, a, \bar{\pi}, \gamma)$, which had already captured the optimal policy, called here Oracle DynSF. Therefore, Oracle DynSF had the perfect state transition model and had only to learn the reward vector. The other variant was that DynSF learned end-to-end, where either the state-transition model or the reward vector was learned. A state-value map for the Axes environment was produced, along with the path the agent took to reach the goal position.

7.4.3 Transfer with Significant Task Change

A significant task change was used to further assess whether the flexibility offered by DynSF can better tackle transfer between different domains. The original SF framework algorithm and DynSF were both evaluated on the same grid world of Axes. The same reasoning was followed as in [152], causing the reward function to change drastically by changing the task, which meant that the goal was relocated to an opposite corner. The agent's start location was considered fixed for both tasks. The episode length and transfer capability of the agents were compared. Sufficient exploration was ensured by using a different ϵ-annealing approach than the original SF algorithm to make the comparison fair and to give the agents a chance to explore.

7.4.4 Performance Evaluation

In the Reacher and Axes environment, DynSF was compared with the Q-learning algorithm and random baselines. In the Doom environment, the proposed DynSF was used to confirm that the method can handle an increasingly complex environment.

7.5 Results & Discussion

This section presents the results of the experiments detailed in the previous section.

7.5.1 Rollout Policy

Figure 7.2 (left) shows the results of the various local policies evaluated on the Reacher environment: *greedy*, *random*, and ϵ-*greedy*. This experiment shows that the greedy and ϵ-greedy policies gave the best performance (close to 100%) in this environment. Although the random policy performed worse than the greedy and ϵ-greedy policies, it still delivered reasonable performance on the tasks. The completely random policy can be seen as forcing additional

exploration because the resulting $\psi(s, a, \bar{\pi}, \gamma)$ function does not attempt to maximize reward during action selection. Each model's mean performance is reported on all plots as the average over runs with varied seeds, which also includes the standard deviation over all runs as a shaded area. From the standard deviation bands around each curve, it is also evident that the other two policies had a much smaller variance in performance, which could have been expected because they attempted to select the optimal state at each step.

Because the agent was trained with a decaying exploration rate, where the action from (7.3) was randomly selected with probability ϵ, equal performance of the greedy and ϵ-greedy policies could have been expected. Both the greedy and ϵ-greedy policies explored at roughly the same rate under this scheduling. However, using an ϵ-greedy policy requires less computing because only the next latent state resulting from random actions needs to be predicted, instead of all future proceeding states. This contrasts with the greedy policy, which expands each future state across the actions using the state-transition model. Therefore, the ϵ-greedy policy was used for the other experiments unless stated otherwise.

This experiment showed that any policy can be chosen during training at each rollout. Standard policies have been used here to show this flexibility. However, a policy can be tailored to accommodate constraints, such as safe explorations.

7.5.2 Rollout Length

Figure 7.2 (right) shows the state-value maps produced by varying the rollout length k in the Axes environment for $k = \{0, 2, 4, 8\}$. The map was created by taking the maximum value produced by (7.3) for the k steps under a local ϵ-greedy policy over all states in the Axes environment.

The reward function produced a clean gradient on a state-value map flowing between the agent and the goal. Note that the agent could solve the given tasks with up to $k = 4$ steps. After this, noise began to dominate the state-value estimates, and the agent could no longer solve the

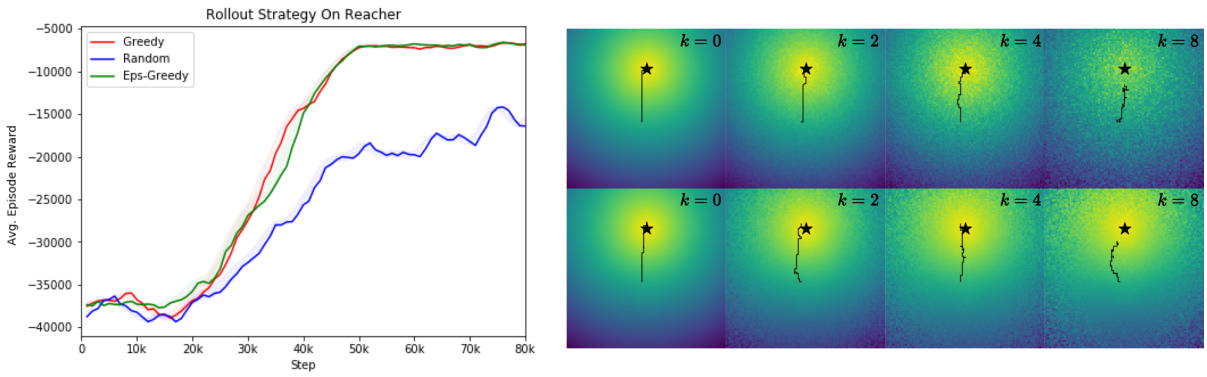


Figure 7.2: **Left**: Learning curves on the Reacher environment with various rollout policies. **Right**: State-value maps of the Axes environment. The goal is shown as the black star and the path taken by the agent from the origin as a black line. *Top row*: Oracle state-transition model and reward vector. *Bottom row*: Learned state-transition model and reward vector.

task. Interestingly, the learned state-transition model appears to have produced better quality estimates in the central area of the map, as evidenced by the smoother gradient. The oracle state-transition model did not have this smoothness around the most visited areas. Because the reward vector was identical, the authors believe that the latent state representation produced by the state-transition model is more resilient to the added noise of further rollout steps.

It can be concluded from these experiments that using many rollout steps might not be required and could decrease agent performance by introducing too much additional noise. Furthermore, there is a computational trade-off to be made because each added step forward requires additional prediction by the state-transition model. Therefore, the optimal number of rollout steps is the smallest number that does not degrade performance.

7.5.3 Transfer with Significant Task Change

Figure 7.3 shows the performance when transferring DynSF and the original SF framework within a substantial task change in Axes. The target goal changed drastically in location, from North(0,1) to South(0,-1). The agent's initial position was unchanged.

The DynSF framework achieved better results, which can be explained by analyzing the mathematical formulation that DynSF contributes. The SF and DynSF frameworks were de-

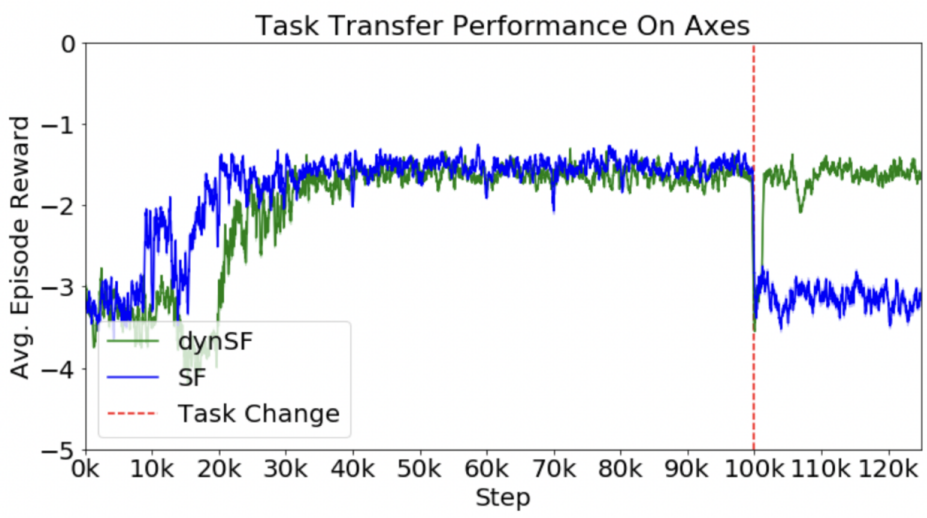


Figure 7.3: DynSF and SF framework compared in a drastic task change. DynSF was able to transfer, but the original SF framework struggled to solve the new task. It is hypothesized here that this performance is the result of a more generalized state representation learned by the DynSF model, which has not specialized to a specific task.

rived by expanding the mathematical definition of Q-Learning. The next step is to acknowledge that a Q-Learning agent will nearly always have fewer parameters than an SF-based agent on the same task, implying that the Q-Learning agent must be learning some compressed form of the task structure. Most likely, the Q-Learning model is learning a representation that is tuned to the particular task's reward structure, discount factor, and state dynamics, leading to a compressed representation where important information related to task change is tracked. This is a clear trade-off because the tracked information will be useful for related tasks. This is hypothesized here to be what leads the SF and DynSF models to generalize much better given a task change: the learned representations from the expanded formulation of such frameworks are much more general. The DynSF model takes this a step further by disentangling the future discounted reward, the discount factor, the acting policy, and the state dynamics to the task-related weights through the supervised learning state representation model, implying much greater generalization capabilities.

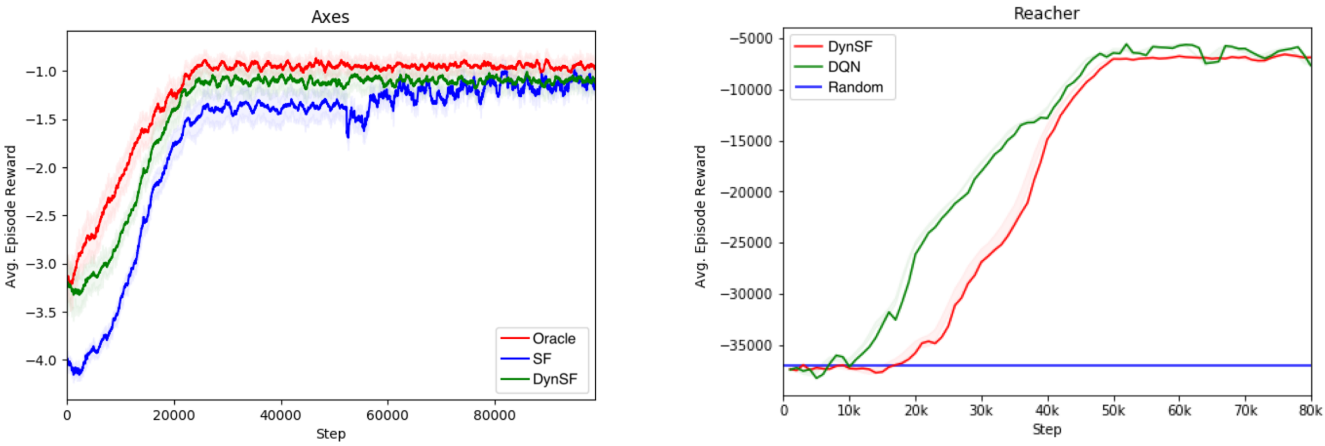


Figure 7.4: **Left**: Learning curves on the Axes environment of an oracle model, the original SF formulation, and the proposed dynamic variant DynSF. **Right**: Learning curves in the Reacher environment of the proposed dynamic variant DynSF compared against DQN and random baselines.

7.5.4 Performance evaluation

Axes

Figure 7.4 (left) shows the results of DynSF against the original formulation of the SF framework and an oracle. The oracle is the DynSF algorithm but with a perfect environment model, hence the oracle label. Therefore, the rollouts will have no additive error. This shows the possible performance if DynSF learns a perfect state-transition model. The results were that all models solved the environment, but the dynamic variant did so faster and with less variance.

Reacher

Figure 7.4 (right) shows the results of the proposed model against a DQN and a random baseline. Both the proposed model and the DQN could solve the environment. Although both models converged to the final solution, the proposed DynSF did so and remained very stable after convergence.

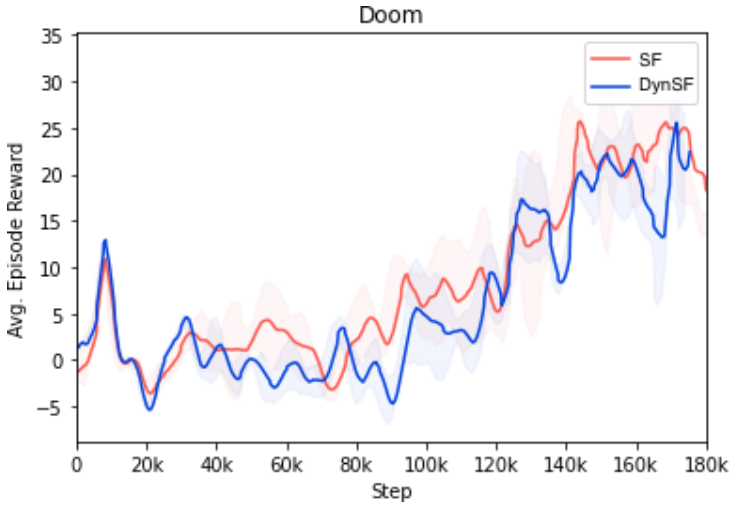


Figure 7.5: Learning curves on the Doom environment of the proposed dynamic variant DynSF compared against the original SF formulation.

Doom

Figure 7.5 shows the performance of the proposed model against the original formulation of the SF framwork in the Doom environment. Figure 7.5 illustrates that the proposed model can successfully navigate the environment to obtain the reward and is competitive with the original formulation of the SFs. Furthermore, it shows that the proposed method can scale up to complex environments and can do so from raw pixels.

7.6 Summary

This chapter has derived and presented a dynamic rollout representation of the SF framework called DynSF, which uses a state-transition model. The DynSF framework enables the setup of dynamic policies and discount factors, which brings flexibility to the original SF framework.

Mathematically, the new formulation brings a clearer insight into what is learned by the SF and DynSF frameworks. Experimentally, it has been shown that different rollout policies yield different performances. This supports a further investigation on DynSF to be used for guided exploration and Safe RL. Furthermore, the effects of rollout length were analyzed. Short rollouts are possible with reasonable performance, but lengthy rollouts can introduce

noise to the training. With comparisons to baselines, it has been demonstrated that RL can be performed through the DynSF framework with two supervised learning tasks, one for the state and another for reward prediction. Finally, given a drastic change in tasks, the DynSF framework adapts better to transfer than the original SF framework.

This result has significant implications, given that the learned state-model representation with dynamic policies and discount factor sets up an on-the-fly flexible successor feature model and enables better task transfer by enabling different policies.

A limitation of the DynSF approach is the need for additional forward inference steps during the bootstrapping process of ψ. The is a clear trade off between using a fixed amortized functional approximation of ψ, as used by the classic SF framework, and our proposed method that gains additional flexibility due to its dynamic nature.

Chapter 8

Conclusions

In the last half-decade, RL has been successfully applied to complex and large-scale tasks. The key drivers of this success are immense amounts of computational power and DNNs as non-linear function approximators. However, the resulting methods could be more effective across essential categories such as task transfer, action efficiency, and sample efficiency. While modern approaches have been suggested, that improve on those essential categories, they themselves are not without issue. The research presented in this thesis provides improvements across task transfer, action efficiency, and sample efficiency. This thesis proposed the S2F and DynSF algorithms in the task transfer category. The S2F algorithm models the agents' rewards with a second-order function providing a stronger guarantee of the agent performance due to its non-linear representational structure and extra parameters. The DynSF model enables the setup of dynamic policies and discount factors, which brings flexibility to the original SF framework. In action efficiency, a planning-centric model, DPN, was presented. DPN uses a planner and agent working in tandem. The planner is optimized to maximize a pseudo-reward, the utility provided to the agent, balancing exploitation and exploration during planning.
Finally, for sample efficiency, Noisy Importance Sampling Actor-Critic (NISAC) has been introduced; a fully *off-policy* actor-critic algorithm that learns from stored off-policy trajectories.

Below, in Section 8.1, a final discussion of the contributions of the thesis is given. Finally,

Section 8.2 presents possible directions for future work.

8.1 Contributions

8.1.1 Noisy Importance Sampling Actor-Critic

This thesis has proposed Noisy Importance Sampling Actor-Critic (NISAC), a fully *off-policy* actor-critic algorithm that learns from stored off-policy trajectories. We have proven, experimentally, that NISAC improves upon the performance and sample efficiency of A2C [41], an *on-policy* actor-critic, and truncated importance sampling [16], an *off-policy* algorithm. NISAC nears the performance of ACER [17], a SOTA *off-policy* actor-critic method, on several environments while completing a training session in 40% less time and being significantly easier to implement. We have analyzed the effect of additive action space noise, identified the Gumbel distribution as the most performant variant, and examined where the noisy policy can be used within the importance sampling weight ρ and policy gradient update. Our analysis shows that additive action space noise fundamentally changes the distribution of importance sample weights ρ during training. Moreover, we have shown that each component in NISAC contributes to its improved performance over the baseline methods, and even with additive action space noise, the learned policies are stable.

8.1.2 Dynamic Planning Networks

This thesis has proposed DPN, a new architecture for DRL that uses a planner and agents working in tandem. The planner is optimized to maximize a pseudo-reward, the utility provided to the agent, balancing exploitation and exploration during planning. We have demonstrated that DPN outperforms the model-free and planning baselines in the Multi-Goal Gridworld and Push environments while using ~2x fewer environment samples. Furthermore, the ability of DPN to learn a dynamic planning style enables it to achieve much greater efficiency in

terms of the state transitions required; this is especially evident when comparing TreeQN, with a fixed planning style, and DPN, with a dynamic planning style. By letting the planner learn its planning style, we see evidence of emergent planning patterns, such as breadth-first search. In the Push environment, we see DPN achieves greater or equal performance to TreeQN while requiring 96% fewer applications of the state-transition model. Taken all together, DPN, compared to other architectures, reduces the computational requirements to reach a similar level of performance. Finally, we have shown that allowing the planner to select *where* to plan from helps avoid sub-optimal trajectories. Our studies have provided evidence that the triplet previous, current, reset provides the greatest performance. In future work, we plan to examine how structured memory can help improve this dynamic planning process.

8.1.3 Second-Order Successor Features

A novel formulation for the Successor Feature framework with a second-order non-linear reward function has been derived, which predicts rewards as a non-linear combination of state features. Experimentally, we have shown that the agent can perform well with the second-order reward structure, providing extra flexibility to the reward model and empowering state representation and policy transfer. The method also shows greater performance compared to the baseline SF method on the Doom environment, providing strong evidence that it can scale quickly to complex environments. Furthermore, the new second-order term Λ_t has been investigated and confirmed to capture environmental dynamics closely. This result has significant implications, given that the state representation within the original Successor Feature framework may not have enough representational power for good state and reward reconstruction tasks, for example, in highly complex environments. Finally, we have provided evidence that the Λ_t term can also be used for exploration by the agent during task transfer.

8.1.4 Dynamic Successor Features

This thesis has derived and presented a functionally equivalent dynamic rollout representation of the SF framework called DynSF, which uses a state-transition model. The DynSF framework enables the setup of dynamic policies and discount factors, which brings greater flexibility to the original SF framework. Indeed, with comparisons to baselines, it has been demonstrated that through the DynSF framework, RL can be performed with two supervised learning tasks, one for the state and another to reward prediction. The use of supervised learning in an RL setting provides immense flexibility in the choice of each learners architecture. Experimentally, it was shown that the DynSF model has equal performance on baseline tasks. However, given a drastic change in tasks, the DynSF framework adapts better to transfer than the original SF framework.

The proposed formulation brings a clearer insight into what the SF and DynSF frameworks learn. Through experimental results, the effect of different rollout policies and rollout lengths was shown to have a tremendous impact on performance. Indeed, shorter rollouts provide good performance, while lengthy rollouts can introduce noise to the training.

This result has significant implications, given that the learned state-model representation with dynamic policies and discount factor sets up an on-the-fly flexible successor feature model and enables better task transfer by enabling different policies.

8.2 Future Work

This thesis has explored various paths for improvement in DRL across sample efficiency, planning, and task transfer on numerous environments and tasks. Directions for possible future research related to the algorithmic improvements, organized by contribution, include:

8.2.1 Dynamic Planning Network

The DPN architecture showed immense flexibility in its planning ability. However, the planner has finite memory and must compress the current plan into the hidden state of the RNN. A natural direction of improvement would involve extending the memory of the planner, similar to Neural Turing Machines [153], such that the agent has a perfect recall and can readily read and write as needed. Information could be used across multiple bouts of planning, such that the planner can integrate previous plan information – possibly bootstrapping subsequent evaluations. This would allow longer planning sequences by the planner unit and access to higher fidelity information as the plan grows. Further, if the memory system is discrete, that is not summarized into an embedding, the planner would have access to exact state. A planner possessing such properties would be able to generate higher quality plans and that gives greater informational context to the acting agent, leading to greater rewards.

8.2.2 Noisy Importance Sampling Actor-Critic

Within sample efficiency and our proposed NISAC model, an exciting future could include evaluating suitable distributions compatible with continuous action spaces. As NISC is currently only compatible with discrete action spaces, a compatible distribution increases the applicability of the NISAC model to domains such as robotics that heavily rely on continuous control. Further investigation into possible annealing schedules for the clamping constant c could yield an increase in performance by respectively helping improve the speed of convergence to an optimal policy and further reducing instability during learning.

8.2.3 Second-Order Successor Features

As shown in Section 6.4, the Λ function can capture the environment's stochasticity. Indeed, during transfer, it was as competitive as the standard exploration methods such as ϵ-greedy. A deeper evaluation of the learned Λ function could yield exciting results in future work, particu-

larly by examining the application of different noise distributions to sample actions for directed exploration. If an agent can exploit previously learned information, such as that learned by the Λ function, it could improve the convergence speed to an optimal policy. Further, by having additional context around the current state, via the Λ function, the agent might be able to smartly explore, improving the exploration-exploitation dilemma.

8.2.4 Dynamic Successor Features

The DynSF model's flexibility, with its mathematical specification and architectural simplicity, leaves room for natural extensions in future work.

It can be hypothesized that the induced policy's generalization, and therefore quality, could be significantly improved by looking to larger state-transition models such as Dreamer [154] or other generative models [155, 156, 157]. Given its large set of parameters, such a model could be generalized across many tasks in complex environments. Atari is an example of an environment where the dynamics of specific tasks, such as Space Invaders and Demon Attack, are nearly identical.

Another line of future work, in a nearly opposite direction to that described in the previous paragraph, is the exploitation of state-transition model prediction noise to enhance the exploration ability of the agent. Indeed, the rollout length experiment described in Section 5.2 revealed that the compounding error causes the agent to explore *along* its path through the environment. By doing so the exploration efficiency could be improved, leading the agent to require fewer samples. The hypothesis is that this noise could be further enhanced by using it as an exploration bonus during learning, much as other studies have used state-prediction or reconstruction as an exploration bonus for the agent [158, 159].

Bibliography

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

[2] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[3] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[4] O. Jangmin, J. Lee, J. W. Lee, and B.-T. Zhang, "Adaptive stock trading with dynamic asset allocation using reinforcement learning," *Information Sciences*, vol. 176, no. 15, pp. 2121–2147, 2006.

[5] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[8] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[9] *Deep Learning RL = AI?* [Online]. Available: https://www.coursera.org/lecture/prediction-control-function-approximation/david-silver-on-deep-learning-rl-ai-xZuSl

[10] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning," *arXiv preprint arXiv:1710.11417*, 2017.

[11] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li *et al.*, "Imagination-augmented agents for deep reinforcement learning," *arXiv preprint arXiv:1707.06203*, 2017.

[12] J. Oh, X. Guo, H. Lee, R. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," *arXiv preprint arXiv:1507.08750*, 2015.

[13] J. Oh, S. Singh, and H. Lee, "Value prediction network," *arXiv preprint arXiv:1707.03497*, 2017.

[14] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *arXiv preprint arXiv:1205.4839*, 2012.

[15] T. Jie and P. Abbeel, "On a connection between importance sampling and the likelihood ratio policy gradient," *Advances in Neural Information Processing Systems*, vol. 23, pp. 1000–1008, 2010.

[16] P. Wawrzyński, "Real-time reinforcement learning by sequential actor–critics and experience replay," *Neural Networks*, vol. 22, no. 10, pp. 1484–1497, 2009.

[17] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[18] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[19] P. Dayan, "Improving generalization for temporal difference learning: The successor representation," *Neural Computation*, vol. 5, no. 4, pp. 613–624, 1993.

[20] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. Van Hasselt, and D. Silver, "Successor features for transfer in reinforcement learning," *arXiv preprint arXiv:1606.05312*, 2016.

[21] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[22] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[23] P. Kidger and T. Lyons, "Universal approximation with deep narrow networks," in *Conference on learning theory*. PMLR, 2020, pp. 2306–2327.

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[25] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," *CoRR*, vol. abs/2102.12092, 2021. [Online]. Available: https://arxiv.org/abs/2102.12092

[26] 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.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*. Springer, 2016, pp. 630–645.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[29] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer, "Scaling vision transformers," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 12 104–12 113.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[32] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[33] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[36] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[37] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[38] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[39] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[40] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPs*, vol. 99. Citeseer, 1999, pp. 1057–1063.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[42] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[43] V. Behzadan and A. Munir, "Whatever does not kill deep reinforcement learning, makes it stronger," *arXiv preprint arXiv:1712.09344*, 2017.

[44] T. Rückstiess, F. Sehnke, T. Schaul, D. Wierstra, Y. Sun, and J. Schmidhuber, "Exploring parameter space in reinforcement learning," *Paladyn*, vol. 1, no. 1, pp. 14–24, 2010.

[45] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, "Parameter space noise for exploration," *arXiv preprint arXiv:1706.01905*, 2017.

[46] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin *et al.*, "Noisy networks for exploration," *arXiv preprint arXiv:1706.10295*, 2017.

[47] S. Chiappa, S. Racaniere, D. Wierstra, and S. Mohamed, "Recurrent environment simulators," *arXiv preprint arXiv:1704.02254*, 2017.

[48] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[49] G. Z. Holland, E. J. Talvitie, and M. Bowling, "The effect of planning shape on dyna-style planning in high-dimensional state spaces," *arXiv preprint arXiv:1806.01825*, 2018.

[50] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine *et al.*, "Model-based reinforcement learning for atari," *arXiv preprint arXiv:1903.00374*, 2019.

[51] D. Ha and J. Schmidhuber, "World models," *arXiv preprint arXiv:1803.10122*, 2018.

[52] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 2.

[53] N. Meuleau, L. Peshkin, L. P. Kaelbling, and K.-E. Kim, "Off-policy policy search," *MIT Articical Intelligence Laboratory*, 2000.

[54] S. Levine and V. Koltun, "Guided policy search," in *International conference on machine learning*. PMLR, 2013, pp. 1–9.

[55] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[56] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," *arXiv preprint arXiv:1606.02647*, 2016.

[57] J. Schmidhuber, "On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models," *arXiv preprint arXiv:1511.09249*, 2015.

[58] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised learning of video representations using lstms," *CoRR*, vol. abs/1502.04681, 2015. [Online]. Available: http://arxiv.org/abs/1502.04681

[59] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel, "Value iteration networks," *arXiv preprint arXiv:1602.02867*, 2016.

[60] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine learning proceedings 1990*. Elsevier, 1990, pp. 216–224.

[61] D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto *et al.*, "The predictron: End-to-end learning and planning," *arXiv preprint arXiv:1612.08810*, 2016.

[62] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 2011, pp. 465–472.

[63] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International Conference on Machine Learning*, 2016, pp. 2829–2838.

[64] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia, "Learning model-based planning from scratch," *arXiv preprint arXiv:1707.06170*, 2017.

[65] A. Vezhnevets, V. Mnih, J. Agapiou, S. Osindero, A. Graves, O. Vinyals, K. Kavukcuoglu *et al.*, "Strategic attentive writer for learning macro-actions," *arXiv preprint arXiv:1606.04695*, 2016.

[66] A. Guez, T. Weber, I. Antonoglou, K. Simonyan, O. Vinyals, D. Wierstra, R. Munos, and D. Silver, "Learning to search with mctsnets," *arXiv preprint arXiv:1802.04697*, 2018.

[67] J. Oh, S. Singh, and H. Lee, "Value prediction network," *CoRR*, vol. abs/1707.03497, 2017. [Online]. Available: http://arxiv.org/abs/1707.03497

[68] V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau, "Combined reinforcement learning via abstract representations," *arXiv preprint arXiv:1809.04506*, 2018.

[69] C. Finn and S. Levine, "Deep visual foresight for planning robot motion," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 2786–2793.

[70] A. Srinivas, A. Jabri, P. Abbeel, S. Levine, and C. Finn, "Universal planning networks," *arXiv preprint arXiv:1804.00645*, 2018.

[71] M. Henaff, W. F. Whitney, and Y. LeCun, "Model-based planning with discrete and continuous actions," *arXiv preprint arXiv:1705.07177*, 2017.

[72] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[73] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[74] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *arXiv preprint arXiv:1611.05763*, 2016.

[75] 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.

[76] M. C. Machado, C. Rosenbaum, X. Guo, M. Liu, G. Tesauro, and M. Campbell, "Eigenoption discovery through the deep successor representation," *arXiv preprint arXiv:1710.11089*, 2017.

[77] P. Dayan, "Improving generalization for temporal difference learning: The successor representation," *Neural Comput.*, vol. 5, no. 4, p. 613–624, Jul. 1993.

[78] A. Barreto, D. Borsa, J. Quan, T. Schaul, D. Silver, M. Hessel, D. Mankowitz, A. Zidek, and R. Munos, "Transfer in deep reinforcement learning using successor features and generalised policy improvement," in *International Conference on Machine Learning*. PMLR, 2018, pp. 501–510.

[79] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman, "Deep successor reinforcement learning," *arXiv preprint arXiv:1606.02396*, 2016.

[80] C. Szepesvári, "Reinforcement learning algorithms for mdps," 2009.

[81] S. Hansen, W. Dabney, A. Barreto, T. Van de Wiele, D. Warde-Farley, and V. Mnih, "Fast task inference with variational intrinsic successor features," *arXiv preprint arXiv:1906.05030*, 2019.

[82] N. Tasfi and M. Capretz, "Noisy importance sampling actor-critic: An off-policy actor-critic with experience replay," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–8.

[83] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[84] OpenAI, "Openai five," https://blog.openai.com/openai-five/, 2018.

[85] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[86] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[87] E. J. Gumbel, *Statistical theory of extreme values and some practical applications: a series of lectures*. US Government Printing Office, 1948, vol. 33.

[88] C. J. Maddison, D. Tarlow, and T. Minka, "A* sampling," in *Advances in Neural Information Processing Systems*, 2014, pp. 3086–3094.

[89] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[90] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," *arXiv preprint arXiv:1611.00712*, 2016.

[91] P. Thomas, "Bias in natural actor-critic algorithms," in *International conference on machine learning*, 2014, pp. 441–448.

[92] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[93] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[94] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," 2012.

[95] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," *GitHub, GitHub repository*, 2017.

[96] N. Tasfi and M. Capretz, "Dynamic planning networks," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–9.

[97] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[98] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[99] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, second edition, in progress ed. MIT press, 2017.

[100] J. Peng and R. J. Williams, "Efficient learning and planning within the dyna framework," *Adaptive Behavior*, vol. 1, no. 4, pp. 437–454, 1993.

[101] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems*, 2014, pp. 1071–1079.

[102] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[103] A. Deac, P. Veličković, O. Milinković, P.-L. Bacon, J. Tang, and M. Nikolić, "Xlvin: executed latent value iteration nets," *arXiv preprint arXiv:2010.13146*, 2020.

[104] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. P. Singh, "Action-conditional video prediction using deep networks in atari games," *CoRR*, vol. abs/1507.08750, 2015. [Online]. Available: http://arxiv.org/abs/1507.08750

[105] M. Guzdial, B. Li, and M. O. Riedl, "Game engine learning from video," 2017.

[106] J.-B. Grill, F. Altché, Y. Tang, T. Hubert, M. Valko, I. Antonoglou, and R. Munos, "Monte-Carlo tree search as regularized policy optimization," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 3769–3778. [Online]. Available: http://proceedings.mlr.press/v119/grill20a.html

[107] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," *Advances in neural information processing systems*, vol. 34, pp. 15 084–15 097, 2021.

[108] N. Chentanez, A. G. Barto, and S. P. Singh, "Intrinsically motivated reinforcement learning," in *Advances in neural information processing systems*, 2005, pp. 1281–1288.

[109] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[110] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "https://github.com/oxwhirl/treeqn/blob/master/treeqn/envs/push.py", 2017.

[111] A. Graves, "Adaptive computation time for recurrent neural networks," *arXiv preprint arXiv:1603.08983*, 2016.

[112] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[113] K. Arulkumaran, N. Dilokthanakul, M. Shanahan, and A. A. Bharath, "Classifying options for deep reinforcement learning," *arXiv preprint arXiv:1604.08153*, 2016.

[114] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[115] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[116] C. Tessler, S. Givony, T. Zahavy, D. Mankowitz, and S. Mannor, "A deep hierarchical approach to lifelong learning in minecraft," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[117] S. Sohn, J. Oh, and H. Lee, "Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[118] M. Eppe, P. D. Nguyen, and S. Wermter, "From semantics to execution: Integrating action planning with reinforcement learning for robotic causal problem-solving," *Frontiers in Robotics and AI*, p. 123, 2019.

[119] B. Wu, J. K. Gupta, and M. Kochenderfer, "Model primitives for hierarchical lifelong reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 34, no. 1, pp. 1–38, 2020.

[120] J. Zhang, J. T. Springenberg, J. Boedecker, and W. Burgard, "Deep reinforcement learning with successor features for navigation across similar environments," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 2371–2378.

[121] L. Lehnert and M. L. Littman, "Successor features combine elements of model-free and model-based reinforcement learning." *J. Mach. Learn. Res.*, vol. 21, pp. 196–1, 2020.

[122] M. Abdolshah, H. Le, T. K. George, S. Gupta, S. Rana, and S. Venkatesh, "A new representation of successor features for transfer across dissimilar environments," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 1–9. [Online]. Available: https://proceedings.mlr.press/v139/abdolshah21a.html

[123] H. Liu and P. Abbeel, "Aps: Active pretraining with successor features," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6736–6747.

[124] L. Szoke, S. Aradi, T. Bécsi, and P. Gáspár, "Skills to drive: Successor features for autonomous highway pilot," *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[125] S. J. Gershman, "The successor representation: its computational logic and neural substrates," *Journal of Neuroscience*, vol. 38, no. 33, pp. 7193–7200, 2018.

[126] W. de Cothi and C. Barry, "Neurobiological successor features for spatial navigation," *Hippocampus*, vol. 30, no. 12, pp. 1347–1355, 2020.

[127] I. Momennejad, E. M. Russek, J. H. Cheong, M. M. Botvinick, N. D. Daw, and S. J. Gershman, "The successor representation in human reinforcement learning," *Nature human behaviour*, vol. 1, no. 9, pp. 680–692, 2017.

[128] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. P. van Hasselt, and D. Silver, "Successor features for transfer in reinforcement learning," in *Advances in neural information processing systems*, 2017, pp. 4055–4065.

[129] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *arXiv preprint arXiv:1802.06070*, 2018.

[130] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," *Artificial Intelligence*, vol. 299, p. 103535, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370221000862

[131] D. Borsa, B. Piot, R. Munos, and O. Pietquin, "Observational learning by reinforcement learning," *arXiv preprint arXiv:1706.06617*, 2017.

[132] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[133] Z. Yu, J. Yu, J. Fan, and D. Tao, "Multi-modal factorized bilinear pooling with co-attention learning for visual question answering," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 1821–1830.

[134] A. Fukui, D. H. Park, D. Yang, A. Rohrbach, T. Darrell, and M. Rohrbach, "Multimodal compact bilinear pooling for visual question answering and visual grounding," *arXiv preprint arXiv:1606.01847*, 2016.

[135] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[136] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning," *arXiv preprint arXiv:1910.10897*, 2019.

[137] N. Tasfi, E. Santana, and M. Capretz, "Policy agnostic successor features," in *Advances in Neural Information Processing Systems*, 2020.

[138] M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[139] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[140] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[141] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[142] S. Verma, G. Novati, and P. Koumoutsakos, "Efficient collective swimming by harnessing vortices through deep reinforcement learning," *Proceedings of the National Academy of Sciences*, vol. 115, no. 23, pp. 5849–5854, 2018.

[143] P. Garnier, J. Viquerat, J. Rabault, A. Larcher, A. Kuhnle, and E. Hachem, "A review on deep reinforcement learning for fluid mechanics," *Computers & Fluids*, vol. 225, p. 104973, 2021.

[144] Q. Li, X. Meng, F. Gao, G. Zhang, W. Chen, and K. Rajashekara, "Reinforcement learning energy management for fuel cell hybrid system: A review," *IEEE Industrial Electronics Magazine*, 2022.

[145] X. Hu, T. Liu, X. Qi, and M. Barth, "Reinforcement learning for hybrid and plug-in hybrid electric vehicle energy management: Recent advances and prospects," *IEEE Industrial Electronics Magazine*, vol. 13, no. 3, pp. 16–25, 2019.

[146] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, A. Babu, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean, "Chip placement with deep reinforcement learning," 2020.

[147] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. Gershman, "Deep successor reinforcement learning," *ArXiv*, vol. abs/1606.02396, 2016.

[148] C. Ma, D. R. Ashley, J. Wen, and Y. Bengio, "Universal successor features for transfer reinforcement learning," *arXiv preprint arXiv:2001.04025*, 2020.

[149] L. Lehnert and M. L. Littman, "Successor features support model-based and model-free reinforcement learning," *CoRR abs/1901.11437*, 2019.

[150] N. Tasfi and M. Capretz, "Dynamic planning networks," *arXiv preprint arXiv:1812.11240*, 2018.

[151] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016.

[152] L. Lehnert, S. Tellex, and M. L. Littman, "Advantages and limitations of using successor features for transfer in reinforcement learning," *arXiv preprint arXiv:1708.00102*, 2017.

[153] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[154] D. Hafner, T. P. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *CoRR*, vol. abs/2010.02193, 2020. [Online]. Available: https://arxiv.org/abs/2010.02193

[155] S. M. A. Eslami, N. Heess, T. Weber, Y. Tassa, K. Kavukcuoglu, and G. E. Hinton, "Attend, infer, repeat: Fast scene understanding with generative models," *CoRR*, vol. abs/1603.08575, 2016. [Online]. Available: http://arxiv.org/abs/1603.08575

[156] T. Wang and P. Isola, "Understanding contrastive representation learning through alignment and uniformity on the hypersphere," *CoRR*, vol. abs/2005.10242, 2020. [Online]. Available: https://arxiv.org/abs/2005.10242

[157] J. Robinson, C. Chuang, S. Sra, and S. Jegelka, "Contrastive learning with hard negative samples," *CoRR*, vol. abs/2010.04592, 2020. [Online]. Available: https://arxiv.org/abs/2010.04592

[158] P.-Y. Oudeyer, J. Gottlieb, and M. Lopes, "Intrinsic motivation, curiosity, and learning: Theory and applications in educational technologies," *Progress in brain research*, vol. 229, pp. 257–284, 2016.

[159] R. Simmons-Edler, B. Eisner, D. Yang, A. Bisulco, E. Mitchell, S. Seung, and D. Lee, "Qxplore: Q-learning exploration by maximizing temporal difference error," 2019.

Curriculum Vitae

Name: Norman L. Tasfi

Education:

Ph.D Software Engineering
University of Western Ontario
London, ON
Jan 2018 - Dec 2022

B.E.Sc Electrical Engineering
University of Western Ontario
London, ON
Sept 2011 - May 2016

Related Work Experience:

Teaching Assistant
University of Western Ontario, London, Ontario, Canada
Jan 2018 - Dec 2021

Research Assistant
University of Western Ontario, London, Ontario, Canada
Sept 2017 - Dec 2017

Inference Engineer
Scale Inference, Palo Alto, California, USA
June 2016 - Aug 2017

Applied AI Researcher
Flipboard, Palo Alto, California, USA
June 2014 - May 2015

Publications:

- N. Tasfi and M. Capretz, "Dynamic Planning Networks," 2021 International Joint Conference on Neural Networks (IJCNN), 2021, pp. 1-9.

- N. Tasfi, E. Santana, and M. Capretz, “Policy Agnostic Successor Features” 2020 Advances in Neural Information Processing Systems (NIPs) Preregistration Experiment, 2020.
- N. Tasfi and M. Capretz, “Noisy Importance Sampling Actor-Critic: An Off-Policy Actor-Critic With Experience Replay,” 2020 International Joint Conference on Neural Networks (IJCNN), 2020, pp. 1-8.