View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by D-Scholarship@Pitt

FEUDAL NETWORKS FOR HIERARCHICAL REINFORCEMENT LEARNING REVISITED

by

Alexander Scott Augenstein

Bachelor of Science in Electrical Engineering

University of Pittsburgh, 2015

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Alexander Scott Augenstein

It was defended on

March 26, 2019

and approved by

Zhi-Hong Mao, Ph.D., Professor
Department of Electrical and Computer Engineering
Department of Bioengineering

Ahmed Dallal, Ph.D., Assistant Professor
Department of Electrical and Computer Engineering

Liang Zhan, Ph.D., Assistant Professor
Department of Electrical and Computer Engineering

Thesis Advisor: Zhi-Hong Mao, Ph.D., Professor
Department of Electrical and Computer Engineering
Department of Bioengineering

Copyright © by Alexander Scott Augenstein

2019

FEUDAL NETWORKS FOR HIERARCHICAL REINFORCEMENT LEARNING REVISITED

Alexander Scott Augenstein, M.S.

University of Pittsburgh, 2019

Hierarchical Reinforcement Learning (RL) has gained popularity in recent years in designing RL algorithms that converge in complex environments. Convergence of RL algorithms remains an active area of research, and no single approach has been found to work for all RL applications. Feudal networks (FuNs) are a hierarchical RL technique attempting to address portability and other problems by defining an internal structure for an RL agent using a Manager-Worker hierarchy. A Manager is that portion of the system utilizing a low temporal resolution component for setting goals to maximize rewards from the environment, while the Worker utilizes a high temporal resolution component for selecting among action primitives to maximize rewards from the Manager. This thesis provides an overview of reinforcement learning and the FuN architecture, then compares the relative convergence rates of untrained FuNs to FuNs constructed by Workers with different physical embodiments under a trained Manager.

TABLE OF CONTENTS

NOMENCLATURE..IX

ACKNOWLEDGEMENTS XII

1.0 INTRODUCTION..1

2.0 REINFORCEMENT LEARNING 3

2.1 ACTION SPACES 5

2.2 POLICIES AND AGENTS 5

2.3 TRAJECTORIES 6

2.4 REWARD AND RETURN...7

2.5 THE RL PROBLEM 8

2.6 VALUE FUNCTIONS..9

2.7 BELLMAN EQUATIONS 10

2.8 MARKOV DECISION PROCESSES...11

3.0 HIERARCHICAL REINFORCEMENT LEARNING 12

3.1 AN OVERVIEW OF FEUDAL NETWORKS 14

3.2 ELEMENTS OF FEUDAL NETWORKS...15

3.2.1 Convolutional Neural Network..16

3.2.2 Differentiable Embedding...17

3.2.3 Recurrent Neural Network 18

3.2.4 Long-Short-Term Memory 19

3.2.5 SoftMax 21

3.3 UPDATING FEUDAL NETWORKS 22

3.3.1 Backpropagation Through Time 23

3.3.2 Dilated Long-Short-Term Memory 26

3.3.3 Von Mises-Fisher Distribution 26

3.3.4 Cosine Similarity Measure 29

3.3.5 Advantage Actor-Critic Algorithm 30

4.0 METHODS 31

4.1 EXPERIMENT DESCRIPTION 32

4.2 EXPERIMENT ARCHITECTURE 33

5.0 RESULTS 36

5.1 UNTRAINED WORKER AND UNTRAINED MANAGER 36

5.2 UNTRAINED WORKER AND TRAINED MANAGER 39

6.0 DISCUSSION 41

7.0 CONCLUSIONS AND FUTURE WORK 42

APPENDIX. SOFTWARE IMPLEMENTATION 44

BIBLIOGRAPHY 46

LIST OF TABLES

Table 1. Experiment Parameters for Untrained Agent (Manager and Worker)............................ 38

Table 2. Experiment Parameters for Trained Manager under Different Embodiments................ 40

LIST OF FIGURES

Figure 1. FeUdal Network (FuN) Architecture 2

Figure 2. Agent-Environment Interaction Loop 4

Figure 3. Cart-Pole Environment 9

Figure 4. Option-Critic Architecture 13

Figure 5. FuN Comparison to Option-Critic on Zaxxon and Asterix 13

Figure 6. LSTM Detailed Topology 20

Figure 7. Neural Network Detailed Topology 25

Figure 8. Von Mises-Fisher Distribution in Two Dimensions with Zero Mean 28

Figure 9. VPG Algorithm 34

Figure 10. Multiple Cart-Pole Embodiments 35

Figure 11. Baseline Results starting from Untrained Agent 37

Figure 12. Results starting from Trained Manager 39

NOMENCLATURE

Symbol	Description
x_t	Full State
$f^{percept}$	Convolutional Neural Network Function
f^{Mrnn}	Dilated Long-Short-Term Memory Function
f^{Wrnn}	Long-Short-Term Memory Function
z_t	Observed State
s_t	Partial Observed State
g_t	Goal
a_t	Action
w_t	Goal Embedding
U_t	Option Embedding
φ	Goal Embedding Function
R_t	Reward Function
r_t	Reward from Environment
A	Action Space
π	Stochastic Policy
θ	Parameter Vector
π_θ	Parameterized Stochastic Policy

Symbol	Description
μ	Deterministic Policy
μ_θ	Parameterized Deterministic Policy
P	State Transition Probability
τ	Trajectory
ρ_0	Initial State Distribution
t	Time
γ	Discount Factor
J^π	Expected Return under Policy π
V^π	Value Function under Policy π
Q^π	State-Action Value Function under Policy π
π^*	Optimal Policy
V^*	Optimal Value Function
Q^*	Optimal State-Action Value Function
J^*	Optimal Expected Return
A^π	Advantage Function under Policy π
h_t^M, h_t^W	Hidden State Vectors
$f_t, i_t, \tilde{C}_t, o_t$	Neural Network Layers
a^j	Network Layer Input
W_f, W_i, W_C, W_o, w^j	Trainable Network Weights
b_f, b_i, b_c, b_o, b^j	Trainable Network Biases
σ	Activation Function

Symbol	Description
d_{cos}	Cosine Similarity Function
z^j	Linear Map
η	Learning Rate
λ, λ'	Regularization Weights
δ^j	Network Layer Error
C	Cost Function
∇	Gradient
$\in$	Element of Set
$\odot$	Hadamard (Element-wise) Product
$\cdot$	Dot Product
$\|.\|_p$	P-Norm
$\leftarrow$	Update
$:=$	Defined As
$E[.]$	Expectation Functional

ACKNOWLEDGEMENTS

The basis for this research stemmed from my passion for developing safe and robust systems that interact with the real world in interesting ways. As new technologies emerge that further our need for automated systems that can work with and around humans, continually improved algorithms capable of rapidly learning to respond to a myriad of unforeseen challenges prior to deployment must be developed and made accessible to technology leaders. How will we achieve these goals? It is my mission to drive society towards a direction of improved understanding of RL and its applications.

My contribution is a small push in the right direction, as the topic of this thesis was posed as an open research question by leaders in reinforcement learning community. In truth, I could not have achieved this level of success without a strong support group. I would like to thank my family, friends, and partner who all supported me throughout this process with kindness and understanding. I would also like to thank my advisor for his continued encouragement that empowered me to realize the passions that sparked my research interest. My gratitude goes to my colleague Kevin Brodmerkel for his thorough review and insights on this document. Lastly, I would like to thank the committee members for their continued guidance, some of whom I have known for almost a decade. Between work and continued education, this has been a long-anticipated achievement. Thank you all for your unwavering support.

1.0 INTRODUCTION

Reinforcement Learning (RL) is the study of agents and how they learn by trial and error. This field formalizes the idea of rewarding or punishing an agent for its behavior, making it more likely to repeat or forego that behavior in the future. RL algorithms are categorized by the mathematical or algorithmic structure used to design an agent. Hierarchical RL, for example, is a subset of RL algorithms that structures an agent in a way that leverages multiple layers of abstraction.

Unlike supervised or unsupervised learning, which provide a learning agent with labeled or unlabeled training examples, RL agents use trial and error to seek rewards by interacting with the environment. In scenarios where RL algorithms converge, they are commonly considered "data hungry" compared to supervised and unsupervised learning algorithms in the sense that they might require many interactions with the environment to converge [1]. One use-case for hierarchical RL architectures is to attempt to reduce the total number of agent-environment interactions required to solve a problem, thus improving convergence rates.

As recently as 2017 the Option-Critic architecture was considered state-of-the-art, but the FeUdal Network (FuN) architecture shown in Figure 1 below was demonstrated to outperform the Option-Critic architecture in a variety of test scenarios [2][3]. Since the advent of FuN architectures, no hierarchical RL studies have been observed to outperform the FuN architecture in any task, making the FuN architecture the state-of-the-art hierarchical RL technique at the time of publication.

This work provides a contribution towards understanding the FuN architecture. This is accomplished by first providing a solid foundation for RL concepts, using that foundation to thoroughly explore FuNs, and finally leveraging the knowledge of FuNs to define an appropriate sub-problem for experimentation. The sub-problem explored in this thesis responds to the requests of the hierarchical RL community in [2] and [3] by addressing the transferability of a Manager to untrained Workers under different physical embodiments. Specifically, transitional policies of the supervisor (Manager) will be transferred to actors (Workers) with different embodiments under controlled conditions. This will be tested on variations of the cart-pole environment as shown in Figure 3 [4].

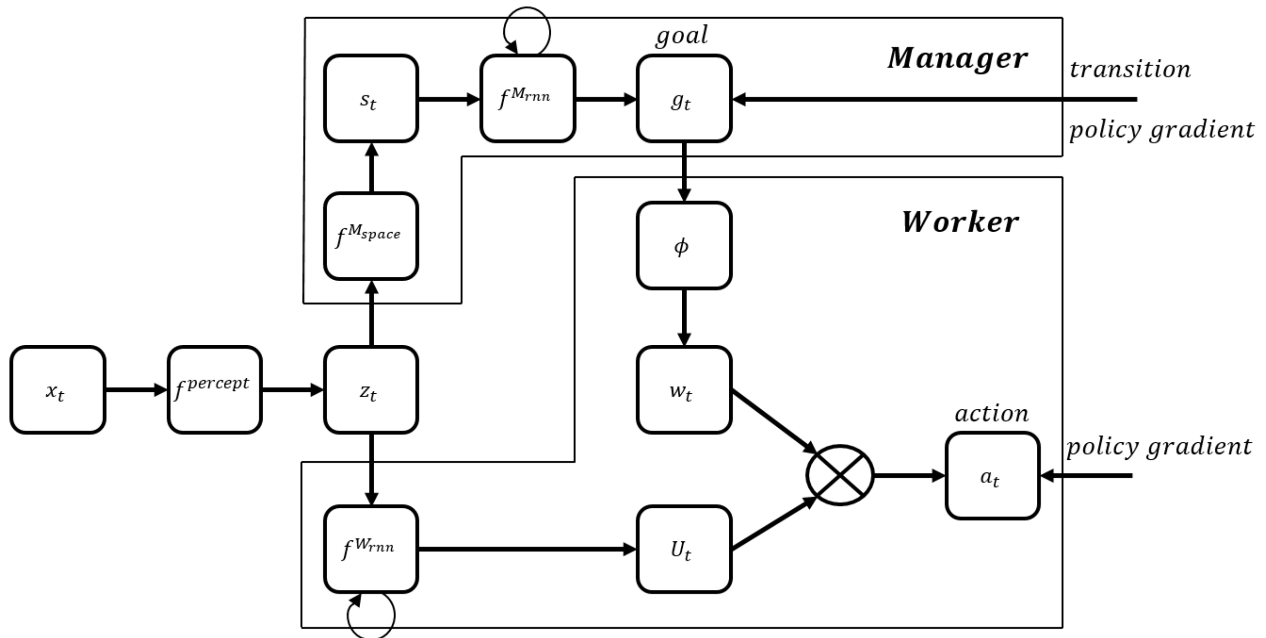

Figure 1. FeUdal Network (FuN) Architecture

2.0 REINFORCEMENT LEARNING

RL is the study of agents and how they learn by trial and error. The field was largely inspired by natural phenomena, and it has been suggested that humans learn using hierarchy [5]. The field of RL at large has recently experienced several notable breakthroughs, including:

- Google Deepmind's algorithm beating the world champion in the (computationally complex, previously considered intractable) game of Go [6]
- OpenAI's algorithm that is currently qualifying for world championship video game tournament participation in the online multiplayer game of DOTA2 [7]

These are only a subset of recent highly publicized results from the field of RL, which remains an active research area [8][9][10]. Because RL formalizes the study of decision making, it could potentially impact any field in which decisions are made.

The agent-environment interaction loop shown in Figure 2 below shows the fundamental progression followed by all RL agents. An agent takes action a_t to modify the state x_t and receives reward through a reward function R_t.

RL notation is not always consistent; however, the work of this thesis was primarily inspired by [3], and the notation in the following sections shall be presented in a way that is consistent with that used in [3].

The objective of an RL agent is to determine an optimal policy. Approaches to this problem are varied and have inspired the discussion in Section 3.0 as to how feudal networks determine an optimal policy. The remainder of this section reviews common terminology from the field of RL to prepare the reader for a thorough discussion of hierarchical RL as it relates to the FuN architecture.

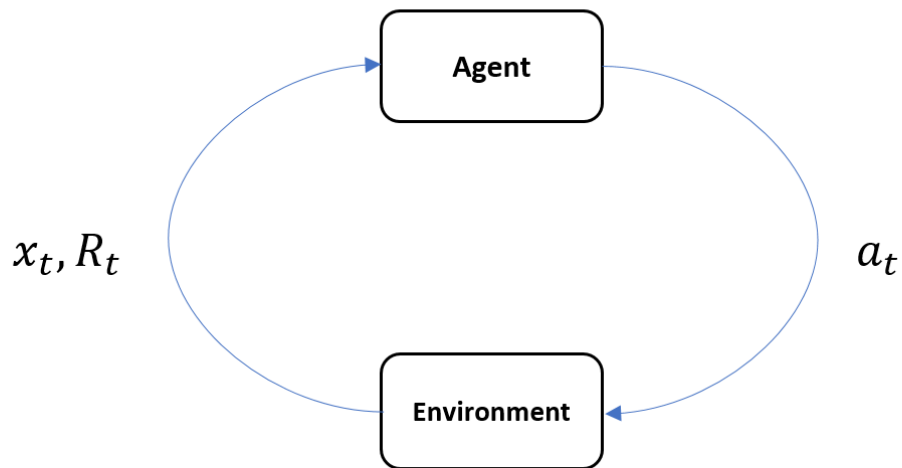

Figure 2. Agent-Environment Interaction Loop

2.1 ACTION SPACES

Different environments enable an agent to take different kinds of actions. The agent is technically not tied to any specific physical embodiment but is responsible for selecting a valid action from the action space $a_t \in A$. Since the action itself is commonly taken by some physical embodiment in the environment (e.g. a robot), it would be appropriate to think of an agent as the brain. An action can be anything that changes the state in some way but is selectable by the agent. Disturbances and noise inherent to the environment don't count as actions. The set of all valid actions, A, in a given environment is called the action space and is the set from which actions, a, are selected by the agent. A time index a_t may be used to denote actions at the current time.

2.2 POLICIES AND AGENTS

A policy is a rule used by an agent to determine what actions to take. The terms "agent" and "policy" are sometimes used as synonyms, but formally the policy is a function within the agent used to select actions. The agent is a structure containing the policy and the ability to sense the environment and collect rewards from it, as well as structure to define and train the policy (e.g. parameters learned through neural networks).

Deterministic and stochastic policies are denoted as:

$$\mu(x_t) = a_t \tag{2-1}$$

$$a_t \sim \pi(.|x_t) \tag{2-2}$$

Policies may be parameterized so that an agent's behavior can be modified using optimization algorithms. The parameters of the policy are denoted by θ. To highlight the connection between these parameters and the policy, policies are sometimes written as $\mu_\theta(x_t)$ or $\pi_\theta(.|x_t)$.

2.3 TRAJECTORIES

A trajectory τ (also known as an episode or rollout) is a sequence of states and actions in the world. The initial state of the world is x_0, and in some problems this is considered to have been drawn from some starting state distribution ρ_0. The transition from one state to the next may not always be deterministic. Even if the policy selects an action with the intention of moving to another state, some unintended state might result. This uncertainty in the state transition can be quantified through a state transition distribution x_{t+1}.

$$\tau = (x_0, a_0, x_1, a_1, \dots) \tag{2-3}$$

$$x_0 \sim \rho_0(.) \tag{2-4}$$

$$x_{t+1} \sim P(.|x_t, a_t) \tag{2-5}$$

2.4 REWARD AND RETURN

The reward function $R(x_t, a_t, x_{t+1})$ is used to define the rewards that the agent can collect from the environment. The goal of the agent is to maximize some notation of cumulative reward over a trajectory. The design of the reward function influences the behaviors expressed by an agent.

Formulations for the reward function include the finite-horizon undiscounted return and the infinite-horizon discounted return.

$$R(\tau) = \sum r_t \, , t \in \{0, \dots, T\} \tag{2-6}$$

$$R(\tau) = \sum \gamma^t r_t \, , t \in [0, \infty) \tag{2-7}$$

The finite-horizon undiscounted return sums up all rewards obtained in a fixed number of steps, and the infinite-horizon discounted return which sums all rewards over time [11]. The infinite-horizon discounted return decreases successive expected rewards by some factor $\gamma^t, \gamma \in (0,1)$, and is used to ensure the infinite sum remains bounded.

2.5 THE RL PROBLEM

The goal of the RL agent is to select a policy which maximizes the expected return when the agent acts according to it. The expected return under a given policy π is denoted as $J(\pi)$. The central optimization problem in RL is to determine the optimal policy π^*, which can be determined from $J(\pi)$.

$$J(\pi) = \underset{\tau \sim \pi}{\mathrm{E}}[R(\tau)] \tag{2-8}$$

$$\pi^* = \underset{\pi}{\operatorname{argmax}} J(\pi) \tag{2-9}$$

One technique used to iteratively update estimates of an optimal policy is the policy gradient method. Policy gradients are discussed in detail in [12], and alternative iterative techniques in [13].

Figure 3 shows a typical environment for the cart-pole problem. A cart on a frictionless track is controlled by forces that can be applied in the left or right directions (along the track). The cart is coupled to an unactuated frictionless rigid pendulum. The objective of this problem is to keep the pole balanced upright and the cart at the center of the track. If this were to be formulated as a reinforcement learning problem, rewards could be collected by an agent for satisfying the above goals. This environment is used in this research to investigate properties of FuNs with minor modifications, as discussed in Section 4.2.

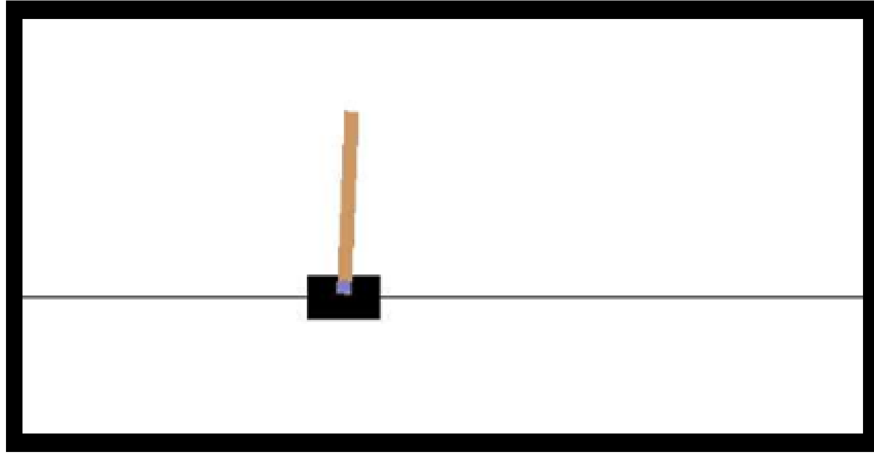

Figure 3. Cart-Pole Environment

2.6 VALUE FUNCTIONS

It can be useful to know how valuable it is to land in a particular state or how valuable it is to take one action (among multiple possible actions) from a given state. A value function $V^{\pi}(x_t)$ quantifies the expected return due to starting in state x_t and acting according to policy π thereafter. An action-value function or Q-function $Q^{\pi}(x_t, a_t)$ quantifies the expected return after taking action a_t from state x_t then always acting according to policy π thereafter. The advantage function quantifies the relative advantage of taking one action over another from a given state and is defined as $A^{\pi}(x_t, a_t)$.

$$A^{\pi}(x_t, a_t) = Q^{\pi}(x_t, a_t) - V^{\pi}(x_t) \tag{2-10}$$

$$a^* = \underset{a}{\operatorname{argmax}}\, Q^*(x_t, a_t) \tag{2-11}$$

These notions are useful for comparing how valuable it is to land in one state versus another, or to quantify how much better it is to take one action compared to another. The optimal value function is denoted $V^*(x_t)$, and the optimal Q-function is denoted $Q^*(x_t, a)$. An optimal policy will select the action that maximizes the expected return starting from x_t. Given the optimal Q-function, the optimal action can be inferred.

2.7 BELLMAN EQUATIONS

Value functions in RL problems obey a set of self-consistency conditions known as the Bellman equations. The basic idea of the Bellman equations can be captured by the notion of cost-to-go. This is the notion that a state's "value" is identical to the reward collected by landing there plus the value of the next state. The Bellman equations can be written as follows [11][14].

$$V^\pi(x_t) = \underset{a\sim\pi,\ x_{t+1}\sim P}{\mathrm{E}}[R(x_t, a_t) + \gamma V^\pi(x_{t+1})] \tag{2-12}$$

$$Q^\pi(x_t, a_t) = \underset{x_{t+1}\sim P}{\mathrm{E}}\left[R(x_t, a_t) + \gamma \underset{x_{t+1}\sim\pi}{\mathrm{E}} Q^\pi(x_{t+1}, a_{t+1})\right] \tag{2-13}$$

$$V^*(x_t) = \max_a \underset{x_{t+1}\sim P}{\mathrm{E}}[R(x_t, a_t) + \gamma V^*(x_{t+1})] \tag{2-14}$$

$$Q^*(x_t, a_t) = \underset{x_{t+1}\sim P}{\mathrm{E}}\left[R(x_t, a_t) + \gamma \max_{a_{t+1}} Q^*(x_{t+1}, a_{t+1})\right] \tag{2-15}$$

2.8 MARKOV DECISION PROCESSES

A Markov Decision Process (MDP) is the formal setting used implicitly in the RL problem definition. RL problems assume the Markov property, which states that state transitions depend only on the most recent state and action [11]. An MDP is a 5-tuple (x, A, R, P, ρ_0) consisting of the state space x, the action space A, the reward function R, the state transition probability P, and the initial state distribution ρ_0 [15].

3.0 HIERARCHICAL REINFORCEMENT LEARNING

Hierarchical RL is a branch of RL that leverages a specific structural assumption for all its algorithms. Specifically, it assumes that problems can be separated into high-level and low-level objectives. Before the introduction of FuN, the dominant architecture was the Option-Critic architecture, shown in Figure 4 below. Intuitively, the Option-Critic architecture demonstrates the notion of hierarchy by appealing to a critic to select the best option (each option defining a policy) that is best for the given context (which includes the current state and rewards collected). Option-Critic architecture trains many policies that are context dependent, allowing for the emergence of complex behaviors by combining learned primitives. In this way, Option-Critic is hierarchical.

A comparative analysis between the performance of FuNs and the Option-Critic architecture was first presented in [3]. The results compare the peak performance of the Option-Critic architecture as published to that of feudal networks across two different tasks. The plots indicate that feudal networks achieved higher scores on these benchmark tasks.

No other hierarchical RL algorithm has since demonstrated superior performance compared to the FuNs approach. While FuNs are state-of-the-art, some properties of this architecture have not yet been thoroughly investigated. Several important quandaries arose within the RL community for further investigation [2][3], one of which is to observe transfer properties of a Manager to different physical embodiments (which are handled by the Worker). This research is focused on the transferability of a trained Manager to different physical embodiments and an in-

depth analysis of each component of the FuN architecture is presented in preparation for breaking the FuN architecture down into a simpler sub-problem, thereby addressing the research question.

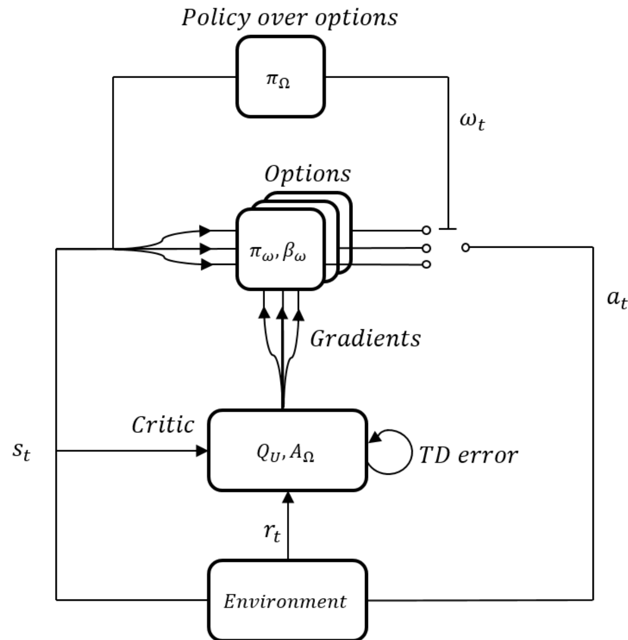

Figure 4. Option-Critic Architecture

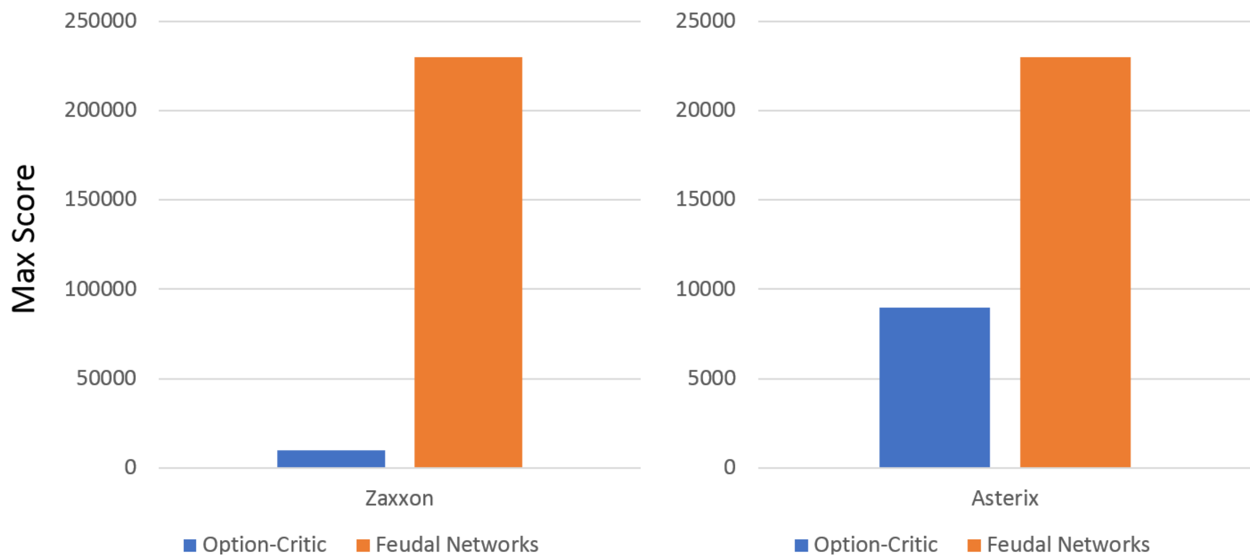

Figure 5. FuN Comparison to Option-Critic on Zaxxon and Asterix

3.1 AN OVERVIEW OF FEUDAL NETWORKS

Figure 1 shows a feudal network as described in [3], based upon the work in [17]. Feudal networks are considered hierarchical since the agent is split into a structure with distinct separation between long-term objectives and short-term actions. In this architecture, the Manager plays the role of a supervisor in that it sets goals for the Worker, but it is not directly responsible for taking actions. The Worker is responsible for selecting actions to achieve the goal set by the Manager. Actions result in a state update. If the action taken by the Worker is aligned with the goal set by the Manager (measured through a cosine similarity), the Worker gets rewarded and as a result the learning step of the algorithm won't substantially change the way the Worker interprets the Manager's goals (e.g. minimal update to U_t).

If the action does not satisfy the goal of the Manager, the Worker gets no reward from the Manager. This more strongly impacts Worker interpretation of the goals set by the Manager. The Manager collects goals directly from the environment, and thus attempts to maximize the collection of these rewards. The Manager rewards the Worker for taking actions that align with the goals even when the Manager sets poor goals (e.g. goals that do not maximize rewards from the environment). The Manager will learn to set better goals when the Worker performs well but the goals aren't achieving the desired result. By setting better goals, the Manager can collect more rewards from the environment. However, the Worker can be trained to follow the goals set by the Manager even when the higher-level objectives aren't being satisfied. In this way, the Worker gets a rich set of reward signals in a potentially sparsely rewarded environment.

After the Manager and the Worker are trained at solving some tasks in tandem (e.g. the algorithm converged), the job of learning how to solve new tasks is essentially a problem for the Manager. The Worker, at this point, learned how to apply action primitives, so the Manager will rely on the Worker to use those primitives to satisfy a new goal.

If the Manager is trained but the Worker is untrained (such as when the physical embodiment of the Worker changes abruptly), faster training times would be expected compared to an untrained network [3].

3.2 ELEMENTS OF FEUDAL NETWORKS

The following discussion steps through the feudal network of Figure 1 to clarify terminology as it relates to functions shown, followed by a detailed discussion of the structure and function of each element in the network.

At the input to the network, the full state of the environment x_t is perceived and mapped via $f^{percept}(.)$ into some potentially informationally lossy state representation z_t shared by both the Manager and the Worker.

This state representation is mapped again by a differentiable embedding $f^{Mspace}(.)$ (e.g. a typical neural network) containing a D-dimensional view of the world for the Manager. The most recent internal state of the Manager h_t^M along with s_t are mapped into a goal g_t at the current timestep through a Recurrent Neural Network (RNN) denoted as $f^{Mrnn}(.)$. A linear transformation without bias φ maps the recent goals set by the manager (over some time horizon $(t-c)$ to (t)) into an embedding vector w_t used to inform the Worker of the Manager's goals.

The shared perception z_t is also mapped to a matrix U_t through an RNN in the Worker portion of the network via f^{Wrnn}. U_t encodes how the Manager's goals might inform the Worker as to which specific action to take. The mapping $SoftMax(U_t w_t)$ produces a policy π_t, a vector of probabilities indicating the likelihood of the Worker selecting one among the available actions given the Manager's goals.

3.2.1 Convolutional Neural Network

The function $f^{percept}(.)$ maps the complete state space x_t to an observation z_t shared by both the Worker and the Manager. This mapping is performed by a Convolutional Neural Network (CNN), which is a class of deep neural networks that can be applied to visual imagery for tasks such as state estimation [18].

Qualitatively, CNN's may be a natural choice as a visual module because they exploit structure in pixel space. In the case of [3], the CNN is used to find an appropriate lower dimensional observation. In this research, since access to full-state is available, a CNN will not be required to estimate the state.

3.2.2 Differentiable Embedding

The FuN architecture uses a differentiable embedding in the design of:

$$f^{Mspace}(z_t) = s_t \tag{3-1}$$

A fully connected neural network is a typical instantiation of an embedding and is differentiable based on appropriate selection of the activation function (e.g. sigmoidal). A neural network is the instantiation used in the FuN architecture.

An embedding in graph applications such as this learning problem, or more generally as defined in [19], is a mapping from discrete objects to vectors of real numbers. Individual dimensions in the vectors may or may not have any inherent meaning, but the overall patterns of location and distance of an embedding are used for efficiently training a network.

The FuN architecture requires a differentiable embedding because the embedding itself is trainable. The training process uses gradient updates and thus the embedding must be at least once-differentiable. Since s_t is the way the Manager sees the world, it may be advantageous for the Manager to change the way they view the world (e.g. update the embedding). This changing of Manager vision may translate to the ability to set better goals. The differentiable embedding f^{Mspace} is on the Manager branch and is thus updated slowly compared to the Worker which operates on a faster temporal resolution.

3.2.3 Recurrent Neural Network

RNN's are used in two locations in the FuN architecture.

$$f^{Mrnn}(s_t) = g_t \tag{3-2}$$

$$f^{Wrnn}(z_t) = U_t \tag{3-3}$$

These RNN's are instantiated by long-short-term memory networks, or LSTM's. One is a standard LSTM and the other is a novel approach presented originally in [3]. Both types are detailed in the following sections but are only a subset of many possible LSTM architectures [20].

RNN's use internal states to process sequences of inputs. This internal state introduces dynamic behavior in time which can have an input response that is either finite-impulse or infinite-impulse. Finite-impulse networks can be used to approximate infinite-impulse networks. This approximation is sometimes used during computation to avoid numeric issues during implementation. Properties of RNNs are addressed at length in [21].

3.2.4 Long-Short-Term Memory

The LSTM consists of several neural networks with different weights, biases, and activations assembled in a specific structure. For a thorough discussion of neural networks, see Section 3.3.1. Consistent with the notation of Figure 6 below, the LSTM equations are as follows [21].

$$f_t = \sigma\left(W_f \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_f\right) \tag{3-4}$$

$$i_t = \sigma\left(W_i \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_i\right) \tag{3-5}$$

$$\tilde{C}_t = \tanh\left(W_C \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_C\right) \tag{3-6}$$

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \tag{3-7}$$

$$o_t = \sigma\left(W_o \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} + b_o\right) \tag{3-8}$$

$$h_t = o_t \tanh(C_t) \tag{3-9}$$

The cell state C_t and the internal state h_t are contain the memory of the cell, which are iteratively modified by their most recent value and the current state. $f_t, i_t, \tilde{C}_t$, and o_t are network layers with trainable weights W_f, W_i, W_C, W_o, trainable biases b_f, b_i, b_c, b_o, with either sigmoidal $\sigma(.)$ or hyperbolic tangent $\tanh(.)$ activations. All multiplications shown are matrix multiplications.

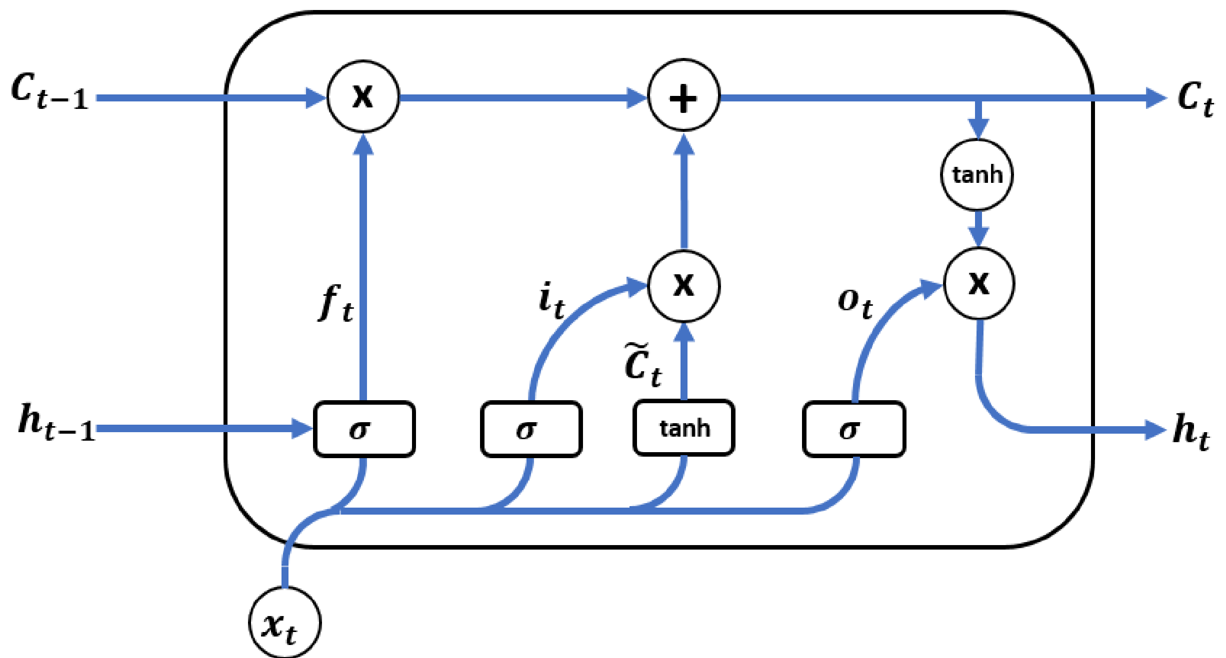

Figure 6. LSTM Detailed Topology

3.2.5 SoftMax

The SoftMax function is defined as follows.

$$softmax(\beta) = \frac{\exp(\beta_j)}{\sum \exp(\beta_k)},\ k = \{1,2,\ldots K\},\ j \in k \tag{3-10}$$

$$\sum softmax(.) = 1 \tag{3-11}$$

This is used in the FuN architecture to define a discrete probability distribution over a finite set of actions (infinite action spaces being quantized). This is done by defining the policy as:

$$softmax(U_t w_t): A \rightarrow \pi \tag{3-12}$$

Since the policy is 1-dimensional, the vector output of the SoftMax function translates to indexes and scalar probabilities. This implies:

$$a_t \sim \pi \tag{3-13}$$

The actions selected by feudal networks are defined by the output of a SoftMax function, because the output of the SoftMax function defines the policy.

3.3 UPDATING FEUDAL NETWORKS

After the Worker selects an action from the policy, the state is updated to reflect any changes. Based on the outcome, the Worker receives a reward that impacts the policy gradient and ultimately the LSTM parameters defining U_t. A policy gradient is a measure of error incorporating the current policy, the reward collected from the most recent action, and some estimate of what the reward could have been if the optimal action was taken. The policy gradient calculation is dependent on the training algorithm used, as discussed in 3.3.5.

It would be possible to train feudal networks using a single policy gradient backpropagated through the network from the output of Worker, but this approach is not used by the architecture in [3]. As discussed in the cosine similarity Section 3.3.4, this design choice allows the goal to retain semantic meaning (e.g. it corresponds to a direction in state space) as opposed to becoming a hard-to-interpret internal latent variable. The training of the Manager and the training of the worker is somewhat decoupled by intentionally not passing a gradient between the two.

Instead of passing the policy gradient, the Manager branch updates according to:

$$\nabla g_t = A_t^M \nabla_\theta d_{cos}\left(s_{t+c} - s_t, g_t(\theta)\right) \tag{3-14}$$

d_{cos} is the cosine similarity function and A_t^M is the Manager advantage function. The policy gradient update relies on value function estimates made by an internal critic as described in Section 3.3.5.

3.3.1 Backpropagation Through Time

RNN's are updated using a process known as backpropagation through time. The standard backpropagation algorithm used to update a feedforward neural network is discussed before extending to the process of BPTT.

The structure of a typical neural network includes an input layer, one or more hidden layers, and an output layer. Each layer contains a specified number of nodes, and each node performs a scalar mapping according to an "activation function" σ. The nodes from one layer to the next are fully connected through a weight matrix w, and offset at each layer by a bias b. The vector output of a layer is called a^j, which maps the vector input z^j through an activation function a^j as:

$$z^j \coloneqq w^j a^{j-1} + b^j \tag{3-15}$$

$$a^j = \sigma^j\left(z^j\right) \tag{3-16}$$

The superscripts indicate the layer used for the calculation. In this section, the convention used throughout follows the specification shown in Figure 7.

Activations for all output nodes a^3 can be calculated by presenting a feature vector x_i at the input of the neural network. Through the following manipulations, the neural network will forward propagate the information at the input and produce some combination of activations at the output:

$$\sigma = \sigma^2(z) = \sigma^3(z) \coloneqq (1 + \exp(-z))^{-1} \tag{3-17}$$

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big) \tag{3-18}$$

$$a^1 = x_i \tag{3-19}$$

$$z^2 = w^2 a^1 + b^2 \tag{3-20}$$

$$a^2 = \sigma(z^2) \tag{3-21}$$

$$z^3 = w^3 a^2 + b^3 \tag{3-22}$$

$$a^3 = \sigma(z^3) \tag{3-23}$$

After forward propagation, network performance can be evaluated against known training output y with deviations accounted for according to some cost function C.

Classification or regression accuracy improvements are achieved by adjusting the network parameters to reduce the cost. By gradient descent, the gradient of the cost function can be used to update network parameters in a direction of reduced cost.

The following are the backpropagation algorithm calculations corresponding to Figure 7:

$$\delta^3 = \nabla_a C \odot \sigma'(z^3) \tag{3-24}$$

$$\delta^2 = ((w^3)^T \delta^3) \odot \sigma'(z^2) \tag{3-25}$$

$$\lambda \coloneqq \eta \frac{\lambda'}{n} \tag{3-26}$$

$$b^2 \leftarrow b^2 - \eta \left(\frac{\partial C}{\partial b^2}\right)^T = b^2 - \eta\delta^2 \tag{3-27}$$

$$b^3 \leftarrow b^3 - \eta\delta^3 \tag{3-28}$$

$$w^2 \leftarrow (1 - \lambda)w^2 - \eta \left(\frac{\partial C}{\partial w^2}\right)^T = (1 - \eta\lambda)w^2 - \eta\delta^2(a^1)^T \tag{3-29}$$

$$w^3 \leftarrow (1 - \lambda)w^3 - \eta\delta^3(a^2)^T \tag{3-30}$$

η is the learning rate, λ' is a regularization weight, and δ is layer error. The last four lines show the update step of the standard backpropagation algorithm.

BPTT is nearly identical but is used to update a recurrent network represented by approximating it as a finite-response network through the process of unfolding. That is to say that if all the states and actions over some time horizon are stored in memory, there exists an equivalent feedforward network that represents the recurrent network over that period. Unfolding is the process of representing the recurrent network as a feedforward network over the stored timeframe (information that is not stored in memory might reasonably be neglected as in [22]). BPTT is then implemented by applying the standard backpropagation algorithm analogous to the discussion above with additional layers. After all the errors have been calculated, the network is folded back up by summing all suggested updates at each point in time for each weight and bias.

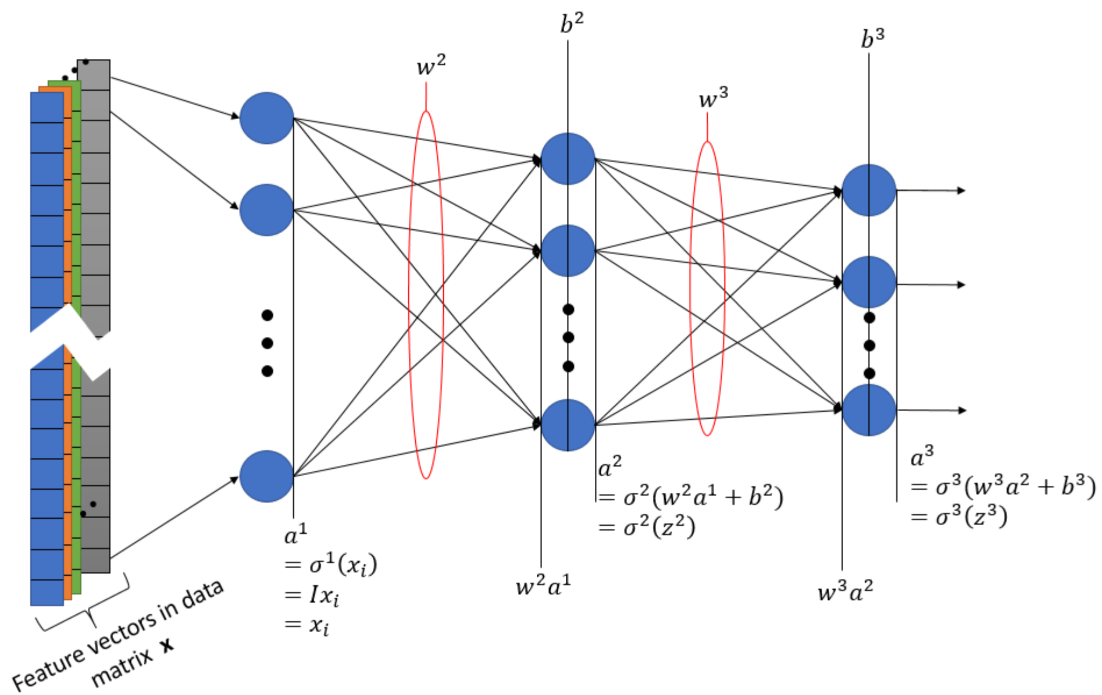

Figure 7. Neural Network Detailed Topology

3.3.2 Dilated Long-Short-Term Memory

Dilated LSTM (dLSTM) is a novel architecture introduced in [3] and is used for the mapping function:

$$f^{Mrnn}(s_t) = g_t \tag{3-31}$$

The only difference between the dLSTM and a standard sequence of LSTM cells is the update rule. For standard LSTM cells, every cell would update with every backward pass through the network. In the case of the dLSTM, batches of cells are frozen while only every r^{th} cell is updated. Reference [3] claims that this modification helps the LSTM improve its ability to retain long term information. Ablative analysis suggested that the lowered temporal resolution was useful for the Manager but not for the Worker, which justified the use of a standard LSTM for the Worker.

3.3.3 Von Mises-Fisher Distribution

The FuN architecture assumes that the direction in state space, $s_{t-c} - s_t$, follows a Von Mises-Fisher distribution. One question that is not directly addressed in [3] is whether this is a reasonable assumption. Some insights can be gained by inspecting properties of the distribution in two dimensions.

Figure 8 shows the Von Mises-Fisher distribution in two dimensions. General properties of the distribution are discussed in [23]. The distribution is defined on an angular interval, in this case $(-\pi, \pi]$. The distribution has a single maximum for all $\kappa > 0$. When $\kappa = 0$, samples from this distribution are uniformly random over the interval. Assuming this distribution describes the state transition $s_{t-c} - s_t$, the situation when $\kappa = 0$ is equivalent to not knowing an expected direction through state space. As κ increases, the distribution begins to concentrate on a single mean value. In the case when $\kappa \to \infty$, the state transition will be in the direction of the mean value with probability approaching 1.

Consider a grid-world where the state transition describes movement from one location on the grid to another. In this scenario the Von Mises-Fisher distribution has very clear semantic meaning, as state transitions are expected to move in the distribution's mean direction along the grid. The Von Mises-Fisher distribution in this case is like a compass, and whichever direction it points in is the direction the state is most likely to transition. Good goals would then provide a strong sense of direction (large κ), since the intrinsic reward is defined by the alignment between the goal and the state transition and these will more likely align when the goal (a known direction) is highly likely to agree with the state transition (large κ in the same direction as the goal). Conversely, poor goals would contain almost no directional information (small κ), as this would shape the distribution as uniform over all angles and thus make it very unlikely that the state transition will align with the direction of the goal.

The Von Mises-Fisher distribution in higher dimensions retains an analogous sense of semantic meaning in terms of setting a direction in state space transitions. The state space itself may or may not have spatial significance but the grid-world example above provides insight into the types of problems which a Von Mises-Fisher distribution can reasonably be expected to model.

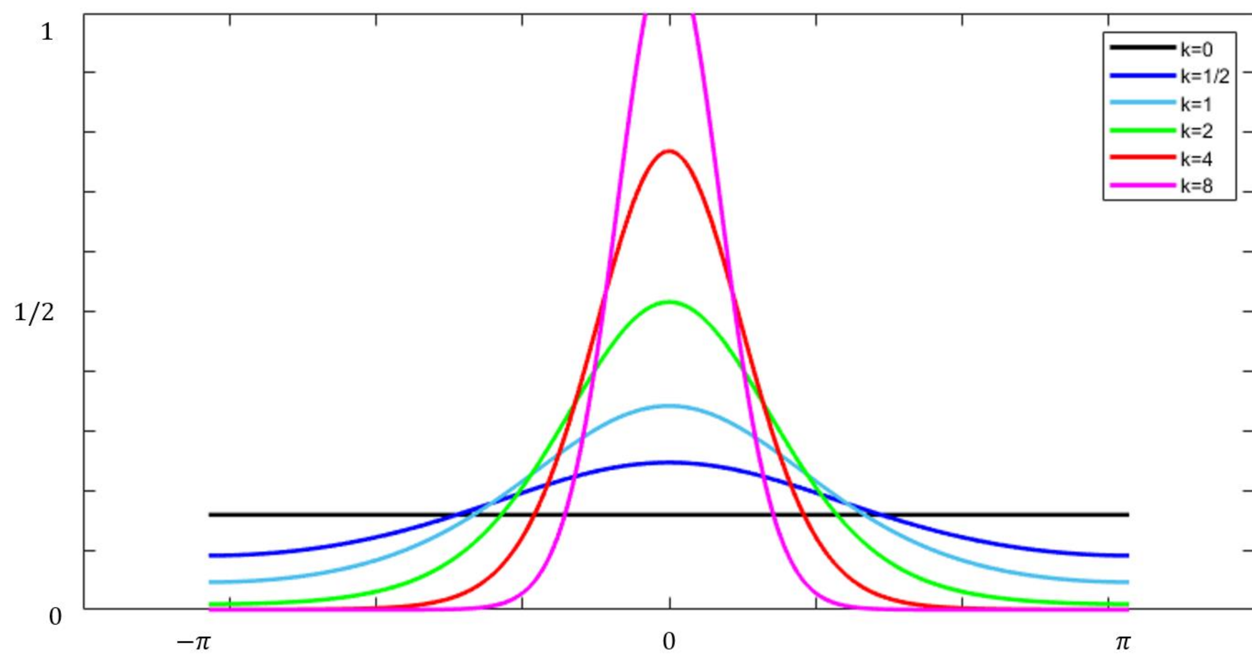

Figure 8. Von Mises-Fisher Distribution in Two Dimensions with Zero Mean

3.3.4 Cosine Similarity Measure

The Worker gets internal rewards for taking actions aligned with the Manager's goals. The Worker never directly observes the Manager's goals but is instructed through the vector w_t. The goals themselves retain semantic meaning due to the Von Mises-Fisher distribution assumed on the state transitions. The state transitions $s_{t-c} - s_t$ will update in a way that tends to progress along the same heading. The states will move in a certain direction for a while without substantially changing direction. The direction is provided by the Manager through g_t, but the Worker is directly responsible for taking actions, and thus determines the way the states transition. The goal g_t is a vector as is the state transition $s_{t-c} - s_t$. If the goal and the state transition align this statement is equivalent to them pointing in the same direction.

The cosine similarity measure is a way of quantifying how closely vectors are aligned:

$$d_{cos}(\alpha, \beta) = \frac{\alpha \cdot \beta}{\|\alpha\|_2 \|\beta\|_2} \tag{3-32}$$

The result is bounded in the interval $[-1, 1]$. If the result is 1, the vectors are pointing in the same direction with arbitrary positive magnitude. If the result is -1, the vectors are pointing in opposite directions with arbitrary positive magnitude. Any value in between means the vectors are misaligned. The key takeaway is that the cosine similarity only compares the heading, not the magnitude. In this way, the goal retains a semantic meaning. The goal vector points in the physical direction in state space the Manager wants to move. If the state transitions are going in this direction, regardless of how fast or slow, the cosine similarity will be close to unity and the Worker will receive a large intrinsic reward.

3.3.5 Advantage Actor-Critic Algorithm

The Worker's policy π is trained using the Asynchronous Advantage Actor-Critic (A3C) algorithm as defined in [24], but the authors in [3] note that any off-the-shelf deep reinforcement learning algorithm can be substituted for updating the policy. This informs the experiment design in this research since it implies that any policy gradient algorithm can be used in place of A3C without loss of generality.

A synchronous version of this algorithm was discovered shortly after A3C, and empirically it has been shown to improve sample efficiency, as compared to A3C [25]. This class of algorithms have come to be known as Advantage Actor-Critic algorithms.

It is called "advantage" actor-critic is due to the estimate of the advantage function used in the policy updates:

$$\nabla \pi_t = A_t^D \nabla_\theta \log\big(\pi(a_t | x_t; \theta)\big) \tag{3-33}$$

The advantage is defined as:

$$A_t^D = \big(R_t + \alpha R_t^I - V_t^D(x_t; \theta)\big) \tag{3-34}$$

The reward function R_t is the reward from the environment. The intrinsic reward function R_t^I comes from the alignment between the state update. The goal, α, is a positive constant that amplifies or attenuates the effect of intrinsic rewards. $V_t^D(x_t; \theta)$ is an estimate of the value function from an internal critic.

4.0 METHODS

Since FuN defines a clear separation between Manager and Worker, the Manager is expected to be capable of learning a transition policy independently of the primitive actions the Worker uses to enact these transitions. The hypothesis of this experiment is that the transition policy of the Manager is transferrable between agents with different embodiments.

If the transition policy is transferable, convergence for an agent given a trained Manager and untrained Workers of different physical embodiments should require less time than the case of an untrained Manager and untrained Workers of different physical embodiments.

Preliminary evidence in support of this hypothesis was presented in [3] by modifying temporally sensitive aspects of previously trained agents. Training these agents versus an untrained control group demonstrated that learning with this specific set of prior information significantly outperformed other methods. This might imply reduced training times under trained Managers when formally testing untrained Workers of different physical embodiments. Results addressing this are presented in Section 5.0.

4.1 EXPERIMENT DESCRIPTION

The question concerns only the relative convergence rate of the Worker under a trained Manager given different physical embdiments. In this section, based on the properties of the FuN architecture, a justification for a simplified network is suggested for testing only the Worker branch of the architecture under controlled conditions. The network is as that studied in [18].

The CNN used in the original FuN paper was used to find an appropriate low dimensional observation of the full state. In this experiment the full state is observable, implying:

$$f^{percept}(x_t) = x_t = z_t \tag{4-1}$$

The goal of a trained Manager will be fixed in steady-state by making some assumptions on the timescale of interest. For this problem, it is asserted that the Manager's goal is "far away". This implies that g_t is constant over the last c timesteps if c is small compared to the number of steps required to approach the goal. Thus, g_t will not change over the next c timesteps regardless of the actions of the Worker. The instructions the Manager provides to the Worker, represented as $\phi(g_{t-c}, \dots, g_t) = w_t$ will then be constant. For this experiment, w_t will be set manually. This is reasonable because the output of the dLSTM will produce entries for the goal vector bounded by $[-1, 1]$, the goals all align for all c, and in the most extreme scenario all of the goals will point in a direction such that the linear projection $\phi(g_{t-c}, \dots, g_t)$ will not lose any information in the projected direction.

For the untrained Manager the linear projection $\phi(g_{t-c}, \ldots, g_t)$ could project in any direction (including any orthogonal dimensions). In the most extreme cases, most of the information will be lost during projection. Based on this, for the untrained Manager a vector w_t of small positive magnitude ϵ in a random direction will be used. In the case of both the trained or untrained Manager, the Worker will still be able to calculate an intrinsic reward and thus have access to the requisite information needed for updates during training.

4.2 EXPERIMENT ARCHITECTURE

OpenAI Gym [4] is a publicly available codebase used in this experiment. This resource is intended to standardize the process of setting up and testing RL algorithms by providing a framework for observing and acting in a variety of environments. Using Gym or some other standardized environment also makes these results more accessible for other RL researchers familiar with Gym. Thus Gym is used in this research in an attempt to make the results more transparent.

Spinning Up in Deep RL [15] is another public code repository made available by OpenAI. This codebase allows for simple implementation of standard RL algorithms in Gym environments. The Spinning Up code implements several algorithms relevant to this research. One such algorithm is the Vanilla Policy Gradient (VPG), which will be used to generate policy updates for our network. In Section 3.3.5 it was discussed that any off-the-shelf deep RL algorithm can be used to implement the policy gradient in the FuN architectureand, since VPG is available for off-the-shelf use from the Spinning Up codebase, VPG is the method of choice for this research.

Figure 9 outlines the pseudocode for the VPG algorithm. This will be used to update the Worker branch of the FuN network as constructed from Section 4.1. Implementation details shown in the Appendix were used in conjunction with the Spinning Up and Gym code repositories to implement the Worker branch for this experiment.

The Worker branch has access to the full state; thus, a CNN was omitted from the shared perceptual module. The setup in Section 4.1 eliminates the need to explicitly define a network for the Manager between z_t and w_t, where w_t is manually defined. For the cart-pole problem, the goal can be quantified by a scalar (negative for positioning the cart to the left and positive for positioning the cart to the right). Examples of different physical embodiments from the experiment are shown in Figure 10.

Algorithm 1 Vanilla Policy Gradient Algorithm

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}
6: Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log\big(\pi_\theta(a_t|s_t)\big)|_{\theta_k} \hat{A}_t$$

7: Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k$$

or via another gradient ascent algorithm like Adam.
8: Fit value function by regression or mean-squared error:

$$\phi_{k+1} = argmin_\phi \left(\frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \big(V_\phi(s_t) - \hat{R}_t\big)^2 \right)$$

typically via some gradient descent algorithm.
9: **end for**

Figure 9. VPG Algorithm

In this experiment, $c = 200$ (corresponding to the episode length), w_t was set according to Table 1 and Table 2, and α (the intrinsic reward weight defined on $[0,1]$) was set to 1 for all experiments to maximize the effect of intrinsic reward. The goals for the Manager were held constant throughout an episode due to the assertions in Section 4.1. For example, the trained Manager was simulated by setting a constant goal of 1, corresponding to continuous rightward motion (so long as the pendulum remains within 15 degrees from vertical, which if not satisfied will end the episode).

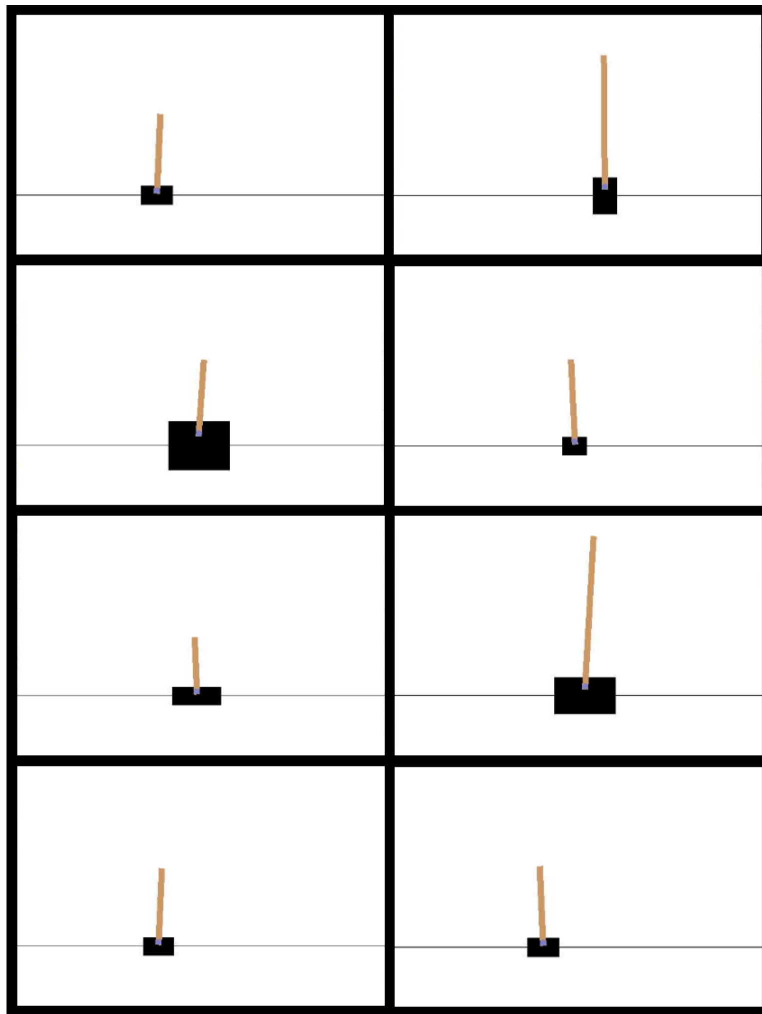

Figure 10. Multiple Cart-Pole Embodiments

5.0 RESULTS

Baseline performance for an untrained agent is presented in Section 5.1. Section 5.2 shows training times for the case of the untrained Worker with different embodiments. Goals are manually adjusted as shown in Table 1 and Table 2. The cart-pole environment is used to train the Worker. Different embodiments arise from different initializations of the cart and pole masses and the pendulum length.

5.1 UNTRAINED WORKER AND UNTRAINED MANAGER

Figure 11 aggregates training times for an initially untrained agent across the combinations of random seeds and cart-pole parameters as shown in Table 1 below.

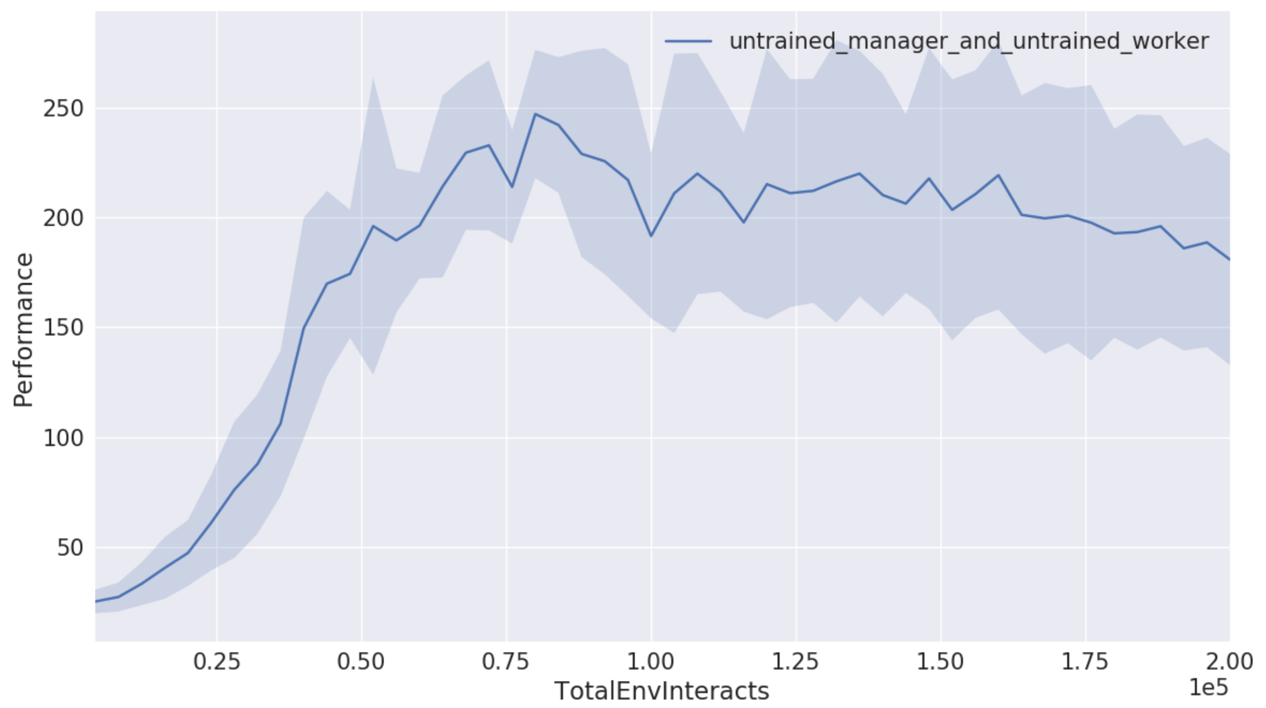

Figure 11. Baseline Results starting from Untrained Agent

Table 1. Experiment Parameters for Untrained Agent (Manager and Worker)

w_t	g_t	**Random Seed**	**Cart Mass (kg)**	**Pole Mass (kg)**	**Pole Length (m)**
-10^{-5}	1	0	1	1	1
10^{-5}	1	1	1.5	1.5	1.5
-10^{-5}	1	2	6	1	1
10^{-5}	1	3	0.75	1	1
-10^{-5}	1	4	1.25	0.5	0.5
10^{-5}	1	5	4	2	2
-10^{-5}	1	6	1	1	1
10^{-5}	1	7	1	1.1	1.1

5.2 UNTRAINED WORKER AND TRAINED MANAGER

Figure 12 shows training times for an agent initialized with a trained Manager and different embodiments across the combinations of random seeds and cart-pole parameters as shown in Table 2 below.

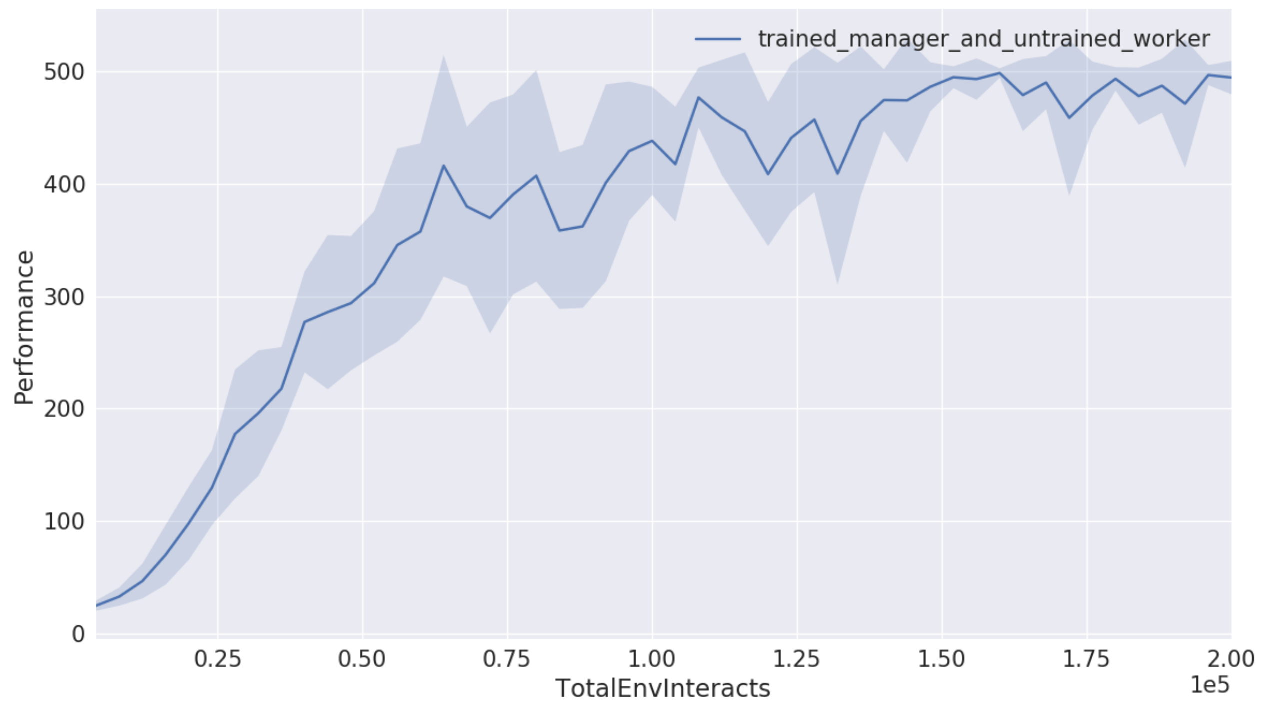

Figure 12. Results starting from Trained Manager

Table 2. Experiment Parameters for Trained Manager under Different Embodiments

w_t	g_t	Random Seed	Cart Mass	Pole Mass	Pole Length
1	1	0	1	1	1
1	1	1	1.5	1.5	1.5
1	1	2	6	1	1
1	1	3	0.75	1	1
1	1	4	1.25	0.5	0.5
1	1	5	4	2	2
1	1	6	1	1	1
1	1	7	1	1.1	1.1

6.0 DISCUSSION

The plots aggregated in Figure 11 correspond to the case of the untrained Manager and untrained Worker. This result uniformly converges in fewer total environment interactions than in Figure 12, representing the case of an untrained Worker across different embodiments under a trained Manager. The baseline experiment exhibits a lower peak performance than that of the simulation incorporating a trained Manager. This implies that an agent exhibits faster overall convergence rates given a trained Manager and different embodiments of the Worker. This result might have been anticipated from an information theoretic perspective, given that an agent constructed from a trained Manager and untrained Worker is initialized with more task specific knowledge than in the case where both the Manager and Worker are untrained.

The sub-problem studied in this experiment was deconstructed into that of training LSTM based networks using policy gradient methods, and the results presented here remain consistent with existing research on similar networks such as that presented in [18].

This experiment supplies test data that corroborates with what the authors in [3] assumed to be a property of the FuN architecture. The result turned out to be consistent with expectation, thus the data from this research enables future work on the FuN algorithm to progress with the confidence that it achieves the objective of portability across physical embodiments by leveraging hierarchy.

7.0 CONCLUSIONS AND FUTURE WORK

This research has illustrated that the FuN architecture exhibits faster overall convergence given a trained Manager and different embodiments of the Worker. This was demonstrated by observing the result that low-information goals (small w_t) inhibit the worker from collecting intrinsic or external rewards, while informative goals support the mutual success of the Worker and the Manager in the sense that the latter test setup led to faster convergence.

To compare training times for the Worker under varying levels of prior knowledge of the Manager, a testbed was constructed by customizing existing open-source code. This approach was used to improve the portability of these findings for the RL community.

The architecture for this experiment minimized sensitivity to potential interactions between training of the Manager and the Worker by making a slowly-time-varying assumption. This temporal decoupling of the training of the Manager from the training of the Worker resulted in a network that could be trained through a single policy gradient algorithm to capture all relevant aspects of Worker training.

This demonstrated that an agent containing an untrained Worker under different embodiments can be trained in less time under a trained Manager than an untrained one, which agreed with the hypothesis of the experiment (and thus agreed with expectations).

Open questions regarding the FuN architecture include setting goals at multiple timescales, scaling agents to large environments, and nesting feudal networks in such a way that hierarchies can include multiple levels of Managers operating on different timescales. The field of hierarchical RL does not necessarily stop with feudal networks. It is possible that completely novel architectures can become the state-of-the-art. Hierarchical RL is by no means considered a solved problem and research in this area continues to be performed.

APPENDIX. SOFTWARE IMPLEMENTATION

The following python code defines the network topology used to implement the Worker branch:

```
import tensorflow as tf

def policy_network(observations,
                   init_states,
                   seq_len,
                   lstm_unit_size,
                   num_layers,
                   num_actions,
                   embedding_vec):
    with tf.variable_scope("rnn"):
        lstm_cell = tf.contrib.rnn.LSTMCell(lstm_unit_size)
        lstm_cells = tf.contrib.rnn.MultiRNNCell([lstm_cell] * num_layers)
        output, final_state = tf.nn.dynamic_rnn(lstm_cells, observations,
                initial_state=init_states, sequence_length=seq_len)

        output = tf.reshape(output, [-1, lstm_unit_size])
        output = tf.matmul(output, embedding_vec)

    with tf.variable_scope("softmax"):
        w_softmax = tf.get_variable("w_softmax",
            shape=[lstm_unit_size, num_actions],
            initializer=tf.contrib.layers.xavier_initializer())
        b_softmax = tf.get_variable("b_softmax", shape=[num_actions],
            initializer=tf.constant_initializer(0))

    logit = tf.matmul(output, w_softmax) + b_softmax
    return logit, final_state
```

The reward function in the Spinning Up implementation of vpg.py was modified by using the above code for the network definition, passing w_t as embedding_vec and appending a $d_{cos}(s_{t+c} - s_t, g_t)$ term to the rewards definition with g_t hard coded as a 1.

The environment selected was the cartpole environment. Repository code was manually modified in the __init__ section of cartpole.py from the Gym source code (which defines the entire cartpole environment). Experiments were run by manually modifying the masscart, masspole, and length parameters in Gym and passing different random seeds through the Spinning Up UI.

BIBLIOGRAPHY

[1] Y. LeCun, *“Predictive Learning”*, Neural Information Processing Systems, 2016, [online], Available: https://www.youtube.com/watch?v=Ount2Y4qxQo

[2] V. Mnih, *“Frontiers Lecture I: Recent Advances, Frontiers and Future of Deep RL”*, Deep RL Bootcamp, August 2017, [online], Available: https://sites.google.com/view/deep-rl-bootcamp/lectures

[3] A.S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, K. Kavukcuoglu, *“FeUdal Networks for Hierarchical Reinforcement Learning”*, eprint arXiv: 1703.01161v2, March 2017.

[4] OpenAI et al. *“OpenAI Gym”,* [online], Available: https://gym.openai.com/

[5] M.M. Botvinick, Y. Niv, A.C. Barto, *“Hierarchically Organized Behavior and its Neural Foundations: A Reinforcement Learning Perspective”*, Cognition, vol. 113, no. 3, pp. 262-280, December 2009.

[6] OpenAI, *“OpenAI Five”*, 2018, [online], Available: https://blog.openai.com/openai-five/

[7] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G.V.D. Driessche, T. Graepel, D. Hassabis, *“Mastering the Game of Go without Human Knowledge”*, Nature, vol. 550, pp. 354-359, October 2017.

[8] J. Sanito, R. Fernandez., A. Swaminathan, K. Tran, K. Hofmann, M. Hausknecht, *“Reinforcement Learning Explained”*, Microsoft Research, 2019, [online], Available: https://www.edx.org/course/reinforcement-learning-explained-3

[9] D. Silver, *“Introduction to Reinforcement Learning”*, COMPM050/COMPGI13: Advanced Topics, 2015, [online], Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

[10] R. Dror and A. Ng, *“Reinforcement Learning and Control”*, CS229: Machine Learning, 2018, [online], Available: http://cs229.stanford.edu/notes/cs229-notes12.pdf

[11] R.S. Sutton and A.G. Barto, *“Reinforcement Learning: An Introduction”* (2nd Edition). Cambridge, USA: The MIT Press, 2018.

[12] R.S. Sutton, D. McAllester, S. Singh, Y. Mansour, *"Policy Gradient Methods for Reinforcement Learning with Function Approximation"*, Advances in Neural Information Processing Systems, vol. 12, pp. 1057-1063, 2000.

[13] C. Szepesvari, *"Algorithms for Reinforcement Learning",* San Rafael, USA: Morgan & Claypool Publishers, 2009.

[14] H. Yu and D. Bertsekas, *"Weighted Bellman Equations and their Applications in Approximate Dynamic Programming"*, Laboratory for Information and Decision Systems Report 2876, October 2012

[15] OpenAI et al. *"OpenAI Spinning Up"*, 2018, [online], Available: https://spinningup.openai.com/

[16] P.L. Bacon, D. Precup, J. Harb, *"The Option-Critic Architecture"*, eprint arXiv: 1609.05140, September 2016.

[17] P. Dayan and G.E. Hinton, *"Feudal Reinforcement Learning"*, Advances in Neural Information Processing Systems, vol. 5, pp. 271-278, December 1992.

[18] T.N. Sainath, O. Vinyals, A. Senior, H. Sak, *"Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks"*, IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 4580-4584, April 2015.

[19] H.Y. Cai, V.W. Zheng, K.C.C. Chang, *"A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications"*, IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 9, pp. 1616-1637, September 2018.

[20] K. Greff, R.K. Srivastava, J. Koutnik, B.R. Steunebrink, J. Schmidhuber, *"LSTM: A Search Space Odyssey"*, IEEE Transactions on Neural Networks and Learning Systems, vol. 28, no. 10, pp. 2222-2232, October 2017.

[21] A. Sherstinsky, *"Fundamentals of Recurrent Neural Network (RNN) and Long-Short-Term Memory (LSTM) Network"*, eprint arXiv: 1808.03314v4, November 2018.

[22] G. Hinton, *"Recurrent Neural Networks"*, CSC2535: Advanced Machine Learning, 2013, [online], Available: https://www.cs.toronto.edu/~hinton/csc2535/notes/lec10new.pdf

[23] *"Von Mises-Fisher Distribution"*, January 2019, [online], Available: https://en.wikipedia.org/wiki/Von_Mises-Fisher_distribution

[24] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, *"Asynchronous Methods for Deep Reinforcement Learning"*, eprint arXiv: 1602.01783, February 2016.

[25] J. Schulman, X. Chen, and P. Abbeel, *“Equivalence Between Policy Gradients and Soft Q-Learning”* eprint arXiv: 1704.06440, April 2017.