Fall 2022

Multi-step Prediction using Tree Generation for Reinforcement Learning

Kevin Prakash
San Jose State University

Multi-step Prediction using Tree Generation for Reinforcement Learning

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Kevin Prakash

December 2022

The Designated Project Committee Approves the Project Titled

Multi-step Prediction using Tree Generation for Reinforcement Learning

by

Kevin Prakash

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2022

Dr. Genya Ishigaki	Department of Computer Science
Dr. Fabio Di Troia	Department of Computer Science
Dr. Guangliang Chen	Department of Mathematics and Statistics

ABSTRACT

Multi-step Prediction using Tree Generation for Reinforcement Learning

by Kevin Prakash

The goal of reinforcement learning is to learn a policy that maximizes a reward function. In some environments with complete information, search algorithms are highly useful in simulating action sequences in a game tree. However, in many practical environments, such effective search strategies are not applicable since their state transition information may not be available. This paper proposes a novel method to approximate a game tree that enables reinforcement learning to use search strategies even in incomplete information environments. With an approximated game tree, the agent predicts all possible states multiple steps into the future and evaluates the states to determine the best action sequences with the highest return. Our proposal differs from deep reinforcement learning in that it uses deep learning for not only the state evaluation but also game tree approximation. This allows it to perform better in completing complex tasks as well as learning in sparse reward environments.

ACKNOWLEDGMENTS

I'd like to thank my family first and foremost for always supporting my academic endeavours. I'd also like to thank Professor Ishigaki for guiding me through the process of writing this. Finally, I'd like to thank my friends for always being entertaining and hanging out with me.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

1.1 Reinforcement Learning

Reinforcement learning is a subsection of machine learning in which an agent learns to interact with its environment to perform certain tasks. This learning is done by giving the agent a numerical reward for actions it takes (positive reward for making positive progress or completing a targeted task and negative reward for failure or making negative progress). The agent attempts to maximize the total reward that it receives by adjusting its actions. This system of environment-agent interaction is modeled in a Markov Decision Process.

Many breakthroughs have been made in this field on ways to train agents. One example deep Q-learning [1] with utilizes deep neural networks to more accurately model the state-action function. Another example is the integration of multi-faceted neural networks to learn complex tasks in an environment with spatial and non-spatial information [2]. Additionally, there have been breakthroughs in ways to analyze the state-action tree to predict the best action to take [3], which allows agents to plan out their actions when the transition function of the environment is known. This paper aims to combine these techniques to integrate a deep learning model that enables reinforcement learning agents to learn state-action trees and plan a sequence of actions by looking multiple steps ahead in the tree.

1.2 Game Playing AI

One of the main uses for reinforcement learning is to learn how to play games. For example, AlphaZero [3] utilizes deep state evaluation and Monte-Carlo Tree Search to learn to play Chess, Shogi and Go at superhuman levels. OpenAI Five [4] was able to beat the 2019 world champion human team in the game DotA 2 by developing a deep Long Short-Term Memory (LSTM) actor-critic architecture.

Combining reinforcement learning with computer vision, agents were able to learn to play Atari games by looking at the rendered screen of the game [5], which was done by the use of convolutional neural networks (CNNs).

The latter two examples of game playing AI function of looking only at the current state could potentially be enhanced with state-action search, which looks several steps ahead to evaluate an action sequence. Although it may seem restricted in application, learning to play games demonstrates the planning and control capabilities of reinforcement learning, which can be applied to other fields such as network management and robotics.

1.3 Challenges for Reinforcement Learning AI

Major challenges of reinforcement learning are sparse rewards and the long or complex action sequences. Sparse rewards are a situation where the environment does not provide sufficient feedback to the agent. Long or complex action sequences are difficult to learn without some ability to make long-term plans.

Methods to dealing with sparse rewards include manually shaping rewards [6, 7] to deal with the sparsity or leveraging learning from a knowledgeable agent that already knows how to do the task [8, 9]. However, in situations where it is not possible or desirable to use either method, it becomes a very difficult problem to deal with sparse rewards because the agent gets very little meaningful feedback on what actions make progress.

This also makes it difficult to calibrate the agent because it may need to take a complex or long series of actions to complete a task. The total number of possible action sequences grows exponentially as the sequence length increases, but there may only be a small fraction of sequences that lead to a desirable end state. This means that the vast majority of explored action sequences lead to undesirable states which

may involve no or negative rewards, making it more difficult for the agent to learn.

One method to mitigate this is to search ahead and examine which sequence of actions will lead to a desirable outcome because an agent needs only to lead itself to a state in which (through search) it can find a path to a desired terminal state. This set of penultimate states is a larger set of states compared to directly trying to lead itself to a desired terminal state. In some environments, with well-defined and known transition rules (such as chess), this is possible and has shown good performance [3], however, this is often not possible as the state transitions may not be known or may be too complex to calculate efficiently. This is what this paper attempts to address.

1.4 Project Statement

The proposed solution of this paper is to learn an approximation of the transition function of a Markov Decision Process (as described in Section 2.1) to generate a state-action tree that can be searched and evaluated to effectively find the optimal action sequences. This should help an agent learn sparse reward environments as well as environments that require complex action sequences to complete a task. This will be accomplished via a neural network architecture based on graph networks.

To test this, the agent will be run on four environments: two of which are sparse reward environments (described in Sections 4.4 and 9) and three of which require complex action sequences to learn (described in Sections 4.4, 4.6, and 4.7). This will demonstrate our agent's capabilities to solve or mitigate some of the challenged described in the previous section.

CHAPTER 2

Background

2.1 Markov Decision Process

A Markov Decision Process (MDP) is a 4-tuple of the form:

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle. \tag{1}$$

- $\mathcal{S}$ is the state space, the set of all possible states.
- $\mathcal{A}$ is the action space: the set of all possible actions. $\mathcal{A}_s \subseteq \mathcal{A}$ denotes the action space for state s.
- $\mathcal{P}$ is a transition function $\mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$. $\mathcal{P}_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the transition probability of going from state s to state s' by taking action a.
- $\mathcal{R}$ is a reward function $\mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. $\mathcal{R}_a(s, s')$ is the numerical reward for reaching state s' from state s by taking action a. In cases in which the reward is only dependent on the state reached, we can denote this $\mathcal{R}(s')$, which is assumed to be the same no matter what values s and a take on.

Because the state inputs of $\mathcal{P}$ and $\mathcal{R}$ are only dependent on a single state, the current state is assumed to encompass the entire information of all past states. This means that no matter what actions were taken or states were visited prior, the information contained in the MDP is the same for a given state.

The goal of a reinforcement learning agent operating on a Markov Decision Process is to find an optimal policy $\pi(s, a) = \Pr(a_t = a | s_t = s)$ of which action a to take given a current state s to maximize the cumulative reward, which is written as a weighted (potentially infinite) sum of immediate rewards. We will call this the return, R.

$$R_t = \sum_{i=t}^{\infty} \gamma^{i-t} \mathcal{R}_{a_i}(s_i, s_{i+1}). \tag{2}$$

From this we define two metrics for evaluating the state and actions taken. The state-value function (or V value) calculates the expected return to be gained by being in a certain state and following a given policy:

$$V_\pi(s) = \mathbb{E}_\pi[R_t|s_t = s]$$

While the state-action function (or Q value) calculates the expected return to be gained by taking a certain action in a certain state and then following a given policy:

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a]$$

2.2 Optimizing Action Policy

There are many methods of finding an optimal policy $\pi*$, but the ones that will be focused on in this paper are Q-learning, policy learning and Monte-Carlo Tree Search.

2.2.1 Q-Learning

Q-learning was first introduced in 1992 [10] as a dynamic programming method for learning optimal behaviour in Markovian domains. The goal is to learn a function $Q_\pi(s, a)$ such that taking action a at state s based on policy π maximizes the expected return. This Q function serves as a proxy for maximizing the state-value function $V_\pi(s)$ which determines the maximum future discounted reward received by being in state s based on the current policy π.

This is done by value adjustments according to:

$$Q_\pi(s_t, a_t) = (1 - \alpha_t)Q_\pi(s_{t-1}, a_{t-1}) + \alpha[r_t + \gamma V_\pi(s_{t-1})], \tag{3}$$

based on the original paper [10], where r is the immediate reward at time step t, α is a learning factor, and γ is a reward discount factor.

When we define loss function $\mathcal{L}$ based on the squared error of our Q-value:

$$\mathcal{L} = \frac{1}{2} \left(Q_\pi(s_t, a_t) - R_t \right)^2, \tag{4}$$

the gradient for backpropagation calculations after an episode completes:

$$\frac{\partial \mathcal{L}}{\partial Q_\pi(s_t, a_t)} = Q_\pi(s_t, a_t) - R_t, \tag{5}$$

where $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$.

2.2.2 Policy Learning

The goal of policy learning is to learn a probabilistic model of bevahiour for the agent, with preferred actions having a higher probability. It can be modeled as:

$$\pi(s, a) = \Pr(a_t = a | s_t = s).$$

There are many ways to optimize a policy, such as Proximal Policy Optimization [11], REINFORCE algorithm [12], and Advantage Actor-Critic [13].

In this paper, we focus on Advantage Actor-Critic. This algorithm consists of two parts: 1) The critic learns the state value $V(s)$ function, and 2) based on the critic's evaluation of the state, the performance of the agent is determined and adjusted by the actor according to:

$$\mathcal{L} = -(R_t - V(s_t)) \cdot \log \pi(s_t, a_t), \tag{6}$$

where R_t is defined the same as in Sec 2.2.1.

This loss $\mathcal{L}$ punishes the agent for performing worse than the critic expects and rewards it for outperforming the critic's expectation. This is also scaled based on how likely the agent was to take the action it took. Combining these factors, the agent learns to take better actions with higher probability.

2.2.3 Monte-Carlo Tree Search

There are many methods for tree searching, but the general approach, as outlined in [14], is to build up a tree by finding the most promising leaf node and then executing from there using a default policy. Upon reaching a terminal state, pass the gradient calculated from the episode back through the chosen leaf node and its parents recursively. This adjusts the value of the nodes in the path from the root node to the leaf node.

2.3 Non-Reinforcement Learning Inspirations

The existing tools in reinforcement learning tend to focus on learning state evaluations, but there is very little work on state or transition approximation. For this we need to look to other machine learning fields for tools to use. Graph networks can assist us in generating the state-action tree as trees are a type of graph. Additionally, the field of computer vision has tools for dealing with high depth models, which is very important if we are to generate high depth trees.

2.3.1 Graph Networks

Graph networks are neural networks designed for sharing information between nodes (and possibly edges) in a graph [15]. The formulation of the message passing is:

$$s_{l+1} = \Psi(s_l, \Theta(\forall_{n \in neighbours} \Phi(s_l, n_l)), \tag{7}$$

where Ψ and Φ are functions for processing two nodes, and Θ is an aggregator function. Φ computes a latent value for the information shared between the current node's state s_l and a neighbour node's state n_l. This value is then aggregated for all neighbours, using Θ. Finally, Ψ calculates the new state of the current node based on the current node's current state and the aggregated latent value of the information shared by the neighbours.

In an updated calculation of this message passing system, used by convolutional

graph networks [16], the node processing functions are linear, and the aggregation is calculated via a normalized adjacency matrix:

$$S_{l+1} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}S_l W_l\right), \tag{8}$$

where S_l is a matrix of the current state of all the nodes at layer l, W_l is the weights for layer l, A is the adjacency matrix of the graph, $\tilde{A} = A + I$, $\tilde{D}$ is the node degree matrix computed from $\tilde{A}$, and σ is an activation. This computation is a fast approximation of the traditional message passing function.

Graph networks of depth d collect information for each node from nodes d steps away. This causes nearby nodes to get similar information, causing regional similarities, known as smoothing. This is desirable when the information of a node is positionally dependent.

2.3.2 Residual Connections

In networks that are extremely deep, gradients can often get crushed because of the stacked activation networks. Later layers also often lose the information of earlier layers because of all the transformations the data has been through.

One of the significant steps in the progression for depth of convolutional networks was the introduction of residual connections [17], which allowed image recognition networks to go from about 20 layers to over 100 layers without deterioration of performance. This residual connections works by adding (or concatenating) the input to a block of functions to the output of those blocks:

$$h(x, \phi) = x + \phi(x). \tag{9}$$

This allows the network to keep information from higher layers as well as directly propagate gradients, allowing them to flow better without getting crushed by numerous activation functions. The proposed agent of this paper utilizes recurrent connections in the tree generation network to stabilize deep state predictions.

CHAPTER 3

Multi-step Prediction using Tree Generation

This chapter explains the process of a Tree Generation agent selecting which action sequence to take from a starting state. The Section 3.1 is the crux of the contributions of this paper as it defines how the agent generates the state-action tree by approximating the transition function for the environment and gives an example network. Section 3.2 explains how a state-evaluation model can take the generated state-action tree and evaluate the leaf nodes with an example network. Section 3.3 describes how to take these state evaluations and find the desired action sequence to take as well as provide an example algorithm to do so. Finally, Sections 3.4 and 3.5 explain how to store the experiences of the agent and then train on them, respectively.

3.1 Tree Generation

The Tree Generation portion of the agent is where it looks ahead (by approximation or definition) to future states by generating a state-action tree. Starting from the current state (the root node), all possible (one per possible action) subsequent states are found (the child nodes). From these child nodes, we find each of their child nodes. Completing this process of finding child nodes D times creates a $D + 1$ depth tree. The leaf nodes of this state-action tree will then be processed by the State Evaluation portion of the agent.

3.1.1 State-Action Tree

In a Markov decision process with deterministic transition rules, it is possible to generate a state-action tree (effectively a generalized game tree) that can predict the state where the agent will be at time $t + d$ after following action sequence $a_1, a_2, ..., a_d$. This is accomplished by generating a tree with the current state, s_t as the root node. Then, for every possible action from the root node, a child node can be generated representing each possible child state s_{t+1}. Note that this is true because the transition

rules are deterministic. This process can be repeated on the leaf nodes of the tree to generate a new set of leaf nodes, thereby increasing the depth of the tree by 1. By starting at the root node and following the actions taken at each node, you can reach the leaf node which will represent the terminal state that will be reached by taking the given action sequence from the current state. These terminal states can be evaluated to determine which action sequence is best.

3.1.2 Relation to Graph Neural Networks

Trees can be thought of as a directed graph with a very specific structure. Notably, each node only has one node that directs towards it (its parent node) and it only is directed towards a small subset of nodes (its children nodes). Due to this, we can heavily simplify the traditional graph network model to work as a message passing system in a tree.

The traditional graph network message passing function [15] is formulated as:

$$s_{t+1} = \Psi(s_t, \Theta(\forall_{n \in neighbours} \Phi(s_t, n_t)) \tag{10}$$

where Ψ and Φ are learnable functions that take in two state values, s_t and n_t are the state values at the current time step of the current node and a neighbour node, respectively, and Θ is a differentiable aggregator function that takes in a variable number of values (e.g. the sum or average).

Now, using our assessment that a tree is a special kind of graph, we can simplify this equation. First, there is only one (incoming) neighbor, a parent node p, so we can replace the neighbor node n_t with p and get rid of the aggregator function. Second, because a child node represents a subsequent state to the parent node, it can be represented as a function F of the parent node p. Therefore, we can replace x_t with $F(p)$. Now, our message-passing function looks like this:

$$s = \Psi(F(p), \Phi(F(p), p)). \tag{11}$$

This is a function of only p, so we can collapse the entire function into a single learnable function of p. Of note, though is that F will be defined differently for each child of p as, in this context, it represents the effect of taking an action, a, on state p. Therefore, each child (one for each possible action) will need a separate function. Additionally, because this is calculating a subsequent state, this process can be repeated on the new child nodes to generate their child nodes. To succinctly write it, we use the notation of

$$s_{[A_t,a]} = \Omega_a(s_{A_t}), \tag{12}$$

where $A_t = [a_0, a_1, ..., a_t]$ is defined as the sequence of actions to be taken starting from the root state, Ω_a is a learnable transition function defined for action a and s_{A_t} is defined as the (potentially approximated) state that results from taking the sequence of actions A_t from the root state.

3.1.3 Tree Generation Networks

This section demonstrates how our Tree Generation Network is constructed. For simplicity of the discussion, this section assumes an environment with non-spatially organized state information (i.e. not signals or images). Note that the concrete model choice in this section to generate the tree is irrelevant to overall tree-generation agent as a whole, and any model that approximates the future states can replace the one in this section. The model selected in this section is the one used for our evaluation experiments.

The foundation of this model is the ‘‘Dense Normalized’’ layer which is a composite layer consisting of a linear transformation, followed by a ReLU activation, followed by output normalization. This model can be represented, in one equation, as

$$Y = \frac{ReLU(XW + B) - \mu}{\sigma}, \tag{13}$$

where X and Y are the inputs and outputs, respectively, W and B are the learnable weights and biases, respectively, and μ and σ are the observed average and standard deviation of the previous outputs of the layer.

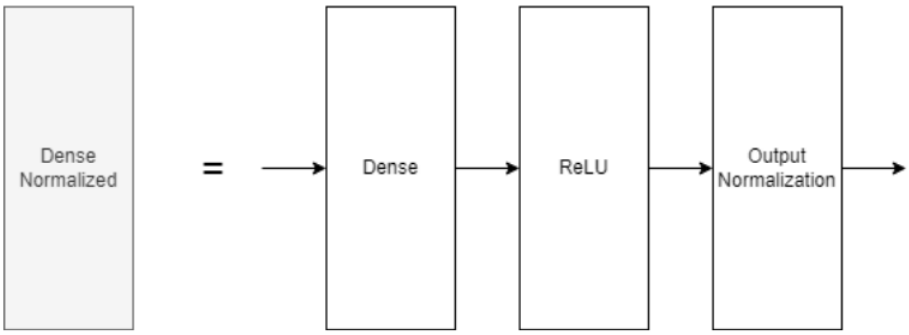

Figure 1: The Dense Normalized Layer

With this building block, we can build the rest of the tree-generation agent. Given the breadth factor B, the number of actions available at state s_t at each time step$t$, the tree generation network predicts the B child state nodes of s_t.

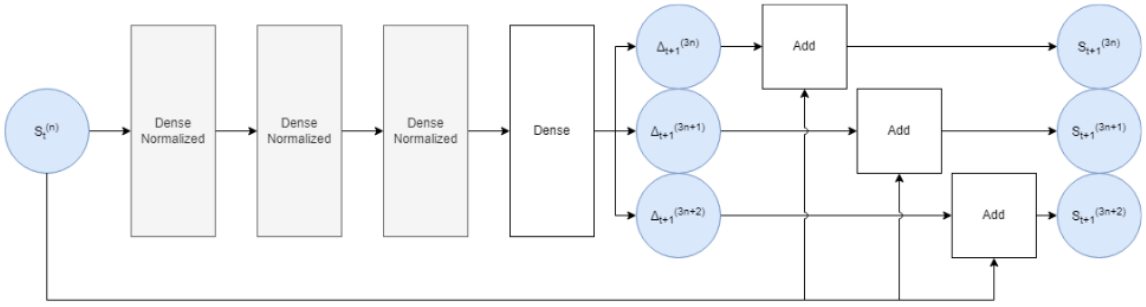

Figure 2: A linear tree generation network with breadth factor = 3

An example of the network is shown in Fig. 2, where the bread factor B is 3. The core of it is three dense normalized layers followed by a dense layer that outputs a delta matrix with size $B \times s$, where B is the breadth factor and s is the size of the input. The input is duplicated B times into a $B \times s$ matrix and then added to the delta matrix to create the $B \times s$ output matrix. The output matrix is then reshaped

into B child state nodes of node $s_t^{(n)}$.

Due to the fact that predictions are compounded, for high depths, the proposed tree generation network is functionally extremely deep. Therefore, it is important to not lose information or crush gradients as depth increases. This method of evaluating the change in state for each child state then adding it to the original state proved more stable and better performing than estimating the child state.

Once these new leaf nodes are computed, we can pass them into the network again to generate the next depth of the tree. Enacting this process D times generates a tree of depth D. Agents that use trees of depth $D-1$ will be referred to as a Depth D model. This slight discrepancy is because the state evaluation predicts the effects of taking an action at the terminal state, which effectively introduces an additional depth. Of note is that Depth 1 models are simply models that use their state evaluation as no state-action tree is generated.

3.2 State Evaluation

The state evaluation portion of the agent is used to analyze which state will lead to the best potential long-term return. The goal is to take in a state and evaluate its value, however that value may be calculated. After calculating the value of all terminal states generated by the Tree Generation portion of the agent, the agent can determine which of the terminal states it will attempt to visit.

Similarly to Sec 3.1.3, the learning algorithms here are examples. Any evaluation algorithm that assigns an actionable value to every action taken from a given state will suffice.

3.2.1 Example 1: Q Estimation Network

The Q estimation of each state is done in parallel by passing every leaf node as a single batch into the state evaluation network. The Q-learning network (as

descibed in Section 2.2.1) used for the environments in this paper consists of three dense normalized layers followed by a single dense layer with a linear activation. The size of output to the final dense layer is equal to the breadth factor B of the environment; thereby, it produces a single $Q(s, a)$ value for every action a that can be taken at the input state s.

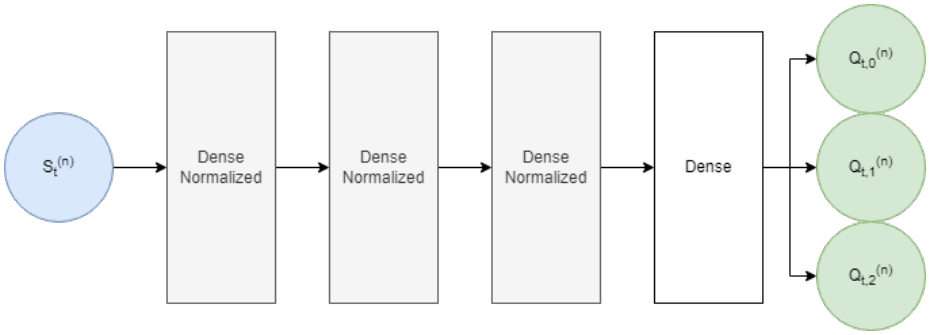

Figure 3: Q Estimation Network with breadth factor of 3

3.2.2 Example 2: Advantage Actor-Critic Network

Since agents using this state evaluation model will be trained using the advantage actor-critic method (as described in Section 2.2.2), the model needs to output both a policy and a value for each terminal state. This model contains three dense layers to compute an intermediate value. This intermediate value is then passed to two separate dense layers. One will generate action weights for the state, and the other will compute a value of the state. The concatenated weights for all the actions of all the leaf states will then be passed through a softmax activation, and this will be used as the policy.

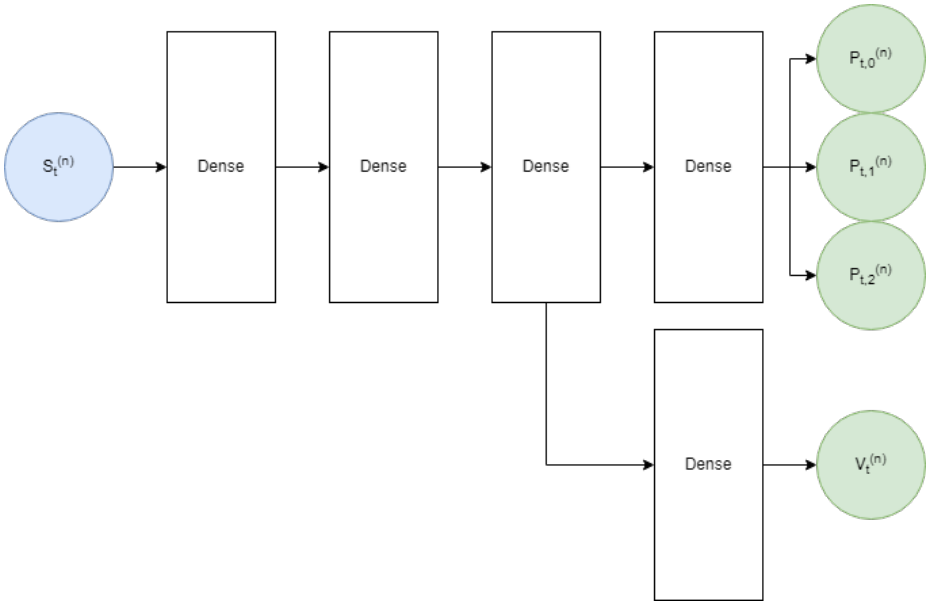

Figure 4: AAC Network with breadth factor of 3

3.2.3 Return Calculation

The value estimation network is trained on the returns calculated for the agent at the respective time step on the states that were visited. To calculate returns, the reward at each time step is recorded and the discounted (possibly normalized) sum of subsequent rewards is used as returns. Normalized rewards, when used, are calculated as:

$$r_t^{(norm)} = \frac{r_t - r^{(min)}}{r^{(max)} - r^{(min)}},$$

where r_t is the reward received at time step t, and $r^{(max)}$ and $r^{(min)}$ are the maximum and minimum possible rewards, respectively. This guarantees that $r_t^{(norm)} \in [0, 1]$.

To avoid temporal bias in truncated episodes, a dummy reward time step, r_{T+1}, is appended to the reward sequence, after normalization, if an episode is truncated prior to reaching a terminal state:

$$r_{T+1}^{(norm)} = \frac{\pm 1}{1 - \gamma}$$

It is either positive or negative depending on what type of reward was being received at truncation. Once normalization (if used) and extra terms (if needed) are added,

the return is calculated with a discount factor γ:

$$R_t = \sum_{i=t}^{t^{max}} \gamma^{i-t} r_i,$$

where r_t is normalized if normalization was done, and t^{max} is either T, the number of time steps in the episode, or $T+1$ if the extra term was added due to truncation. If an extra term was added, the final return time step is not used. To keep the return within a normalized range, we calculate what the max value of a theoretically infinite series of returns would be and divide a return by the bound as follows.

$$R_t^{(norm)} = \frac{R_t^{(norm)}}{\frac{1}{(1-\gamma)}}$$

$$R_t^{(norm)} = R_t^{(norm)}(1-\gamma)$$

3.3 Tree Search

Tree search can be done with any algorithm suitable to the problem environment, such as the one proposed in [18, 3] The search algorithm needs to identify which leaf nodes are desirable to reach.

By tracing back and recording an action sequence from the identified leaf node to the root node, an RL agent can identify which actions were taken at each parent node to eventually reach the best leaf node. Reversing this action sequence will result in the action sequence that the agent needs to take to maximize the return.

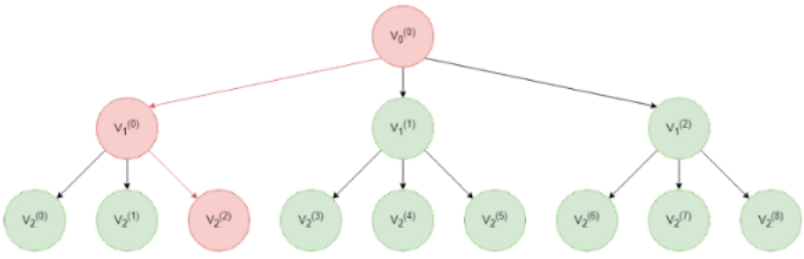

Figure 5: Depth 2 Tree Search

As an example, Figure 5 shows a Depth 2 tree with the path to follow highlighted in red. In this example, from our target terminal node $V_2^{(2)}$, we see that it has a child index of 2 for its parent node $V_1^{(0)}$ which itself has a child index of 0 for its parent node (and root node) $V_0^{(0)}$. This makes our child indices list [2, 0] which we reverse, making our action index list [0, 2]. This means that in the current state, we should take action 0 followed by action 2 to attempt to reach our target terminal state.

3.4 Replay Buffer

To store items to train on, a replay buffer (similar to the one used in [1]) is used. This replay buffer stores replays which are sequences of states, the actions taken at each state, and the return corresponding to the state. If the replay buffer is below max capacity, sequences are appended to the buffer. Otherwise, the buffer is shuffled and the new replays are inserted sequentially into the start of the buffer, as described in Alg. 1.

Algorithm 1 Insertion Into Full Replay Buffer

$Insert \leftarrow [\alpha_0, \alpha_1, \alpha_2, ...]$
$Buffer \leftarrow [\beta_0, \beta_1, \beta_2, ...]$
$i \leftarrow 0$
$l \leftarrow \text{size}(Buffer)$
Shuffle $Buffer$
while i $<$ size($Insert$) **do**
    $Buffer[i \,\%\, l] = \alpha_i$
    $i \leftarrow i + 1$

At the end of each episode, all the states, actions and returns (SAR) are stored or computed. At each time step, a sequence of SAR starts from that time step and includes all SAR's for the next D time steps, where D is the depth of the agent. After generating a complete episode, the SAR sequence is inserted into the buffer. The buffer is then shuffled, and SAR sequences are iteratively used for training. This process applies a recency bias to the agent as they are more likely to be trained on

SAR sequences from newer episodes. This recency bias allows for the agent to learn faster as they are more likely to train on experiences that are from when the agent was better trained, reinforcing the learned behaviour.

3.5 Training

Only the nodes that were actually visited during training, whether intentionally or by random action, are used for training. Given a SAR sequence, we can generate a tree from the initial state using the tree generation network and determine which nodes are utilized for training on that SAR sequence by following the action sequence held in the SAR sequence.

3.5.1 Example 1: Q Learning

Given a SAR tuple with sequence length D, our loss function for the Q-Learning is defined as:

$$\mathcal{L}_{state} = \sqrt{\frac{1}{nD}\sum_{i=1}^{D}\sum_{j=1}^{n}\left(\hat{s}_i^{(j)} - s_i^{(j)}\right)^2}, \tag{14}$$

$$\mathcal{L}_Q = \left|\hat{q}_{s_D,a_D} - q_{s_D,a_D}\right|, \tag{15}$$

$$\mathcal{L} = \mathcal{L}_{state} + \mathcal{L}_Q, \tag{16}$$

where $\hat{s}$ and s represent the predicted and actual state vectors, respectively, $\hat{q}$ and q represent the predicted and actual state-action values, respectively, (j) representing the index of the state vector and i representing the time step relative to the beginning of the SAR sequence. The loss function (Eq. 14) is the sum of the root mean squared error of the state prediction and the absolute error of the Q-value prediction of the terminal state.

3.5.2 Example 2: Advantage Actor-Critic Training

Given a SAR tuple with sequence length D, we can define our loss function for the Advantage Actor-Critic is defined as

$$\mathcal{L}_{state} = \sqrt{\frac{1}{nD}\sum_{i=1}^{D}\sum_{j=1}^{n}\left(\hat{s}_i^{(j)} - s_i^{(j)}\right)^2}, \tag{17}$$

$$Advantage = v_{s_D} - \hat{v}_{s_D}, \tag{18}$$

$$\mathcal{L}_{\pi} = -1 \cdot Advantage \cdot \log \pi(s_D, a_D), \tag{19}$$

$$\mathcal{L}_{value} = 0.5 \cdot Advantage^2, \tag{20}$$

$$\mathcal{L} = \mathcal{L}_{state} + \mathcal{L}_{\pi} + \mathcal{L}_{value}, \tag{21}$$

where $\hat{s}$ and s represent the predicted and actual state vectors, respectively, $\hat{v}$ and v represent the predicted and actual state values, respectively, $\pi(s, a)$ representing the probability of taking action a in state s under the current policy, (j) representing the index of the state vector and i representing the time step relative to the beginning of the SAR sequence.

CHAPTER 4

The Experiments

4.1 OpenAI Gym

To test the agents, OpenAI Gym has been used to provide environments. Gym provides several environments of varying input types, output types and difficulty. Many tasks are difficult even for state-of-the-art reinforcement learning agents, so the testing environments were chosen so as to be dimensionally simple (discrete input and output with only a few variables) and possible for the baseline agent, which is one of the standardized reinforcement learning algorithms discussed in the previous chapter, to complete. These restrictions were satisfied by the environments in the Classic Control subset of gym environments. This allows for a proof of concept to show that tree generation provides benefits even to naive standardized reinforcement learning agents. The environment documentation is available at https://www.gymlibrary.dev/. The specifics of the environments will be described in the following sections.

4.2 General Training Settings

The Adam optimizer [19] with a dynamic learning rate was used. The initial learning rate was set to 0.01, with an additional multiplier of $10^{-k/25}$, where k is the number of the current episode.

The gradient from state loss and value estimation or policy losses are detached. This is to prevent the value estimation or policy networks from causing inaccuracies in the state estimation because of their gradients.

4.3 General Running Settings

For all environments, with the exception of the Mountain Car environment, five agents of each depth (the tree depth of the tree generation algorithm) 1, 5, and 10 are trained for 50 episodes, and the results are averaged over the five agents per episode. Due to the prohibitively long duration of the Mountain Car environment, only one

agent of each depth 1, 7, and 13 is trained for 50 episodes. The higher depth values in this environment are because there is deep transition approximation, which frees up GPU memory space for higher depth predictions. In addition to the raw performance stats, moving averages are presented in the following sections as well. The moving average of k-th episode is computed by averaging the reward from $(k-5)$-th episode to $(k+5)$-th episode.

4.4 Acrobot Environment

Figure 6: Screenshot of the Acrobot Environment

As described in Gym's documentation: "The Acrobot environment is based on Sutton's work in 'Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding' [20] and Sutton and Barto's book[21]." The environment itself consists of two unit-length pendulums which are attached via a connecting join with one being anchored by an anchoring joint. The observation space of the environment is a six-dimensional vector consisting of the sine, cosine and angular velocities of both joints. The goal for the agent is to apply either -1, 0 or 1 [N m] torque to the connecting joint at each time step to attempt to get the non-anchored

tip of the double pendulum above a height of 1, where the anchoring joint is at a height of 0. The episode truncates if 500 time steps have passed.

4.4.1 Observation Preprocessing

To keep all observations in the range [-1, 1], the angular velocity of the anchoring joint is divided by 12.57 and the angular velocity of the connecting joint is divided by 28.27 before passing the observations to the agents. The rest of the observation values are unchanged.

4.4.2 Reward and Return Structure

The reward at each time step is -1. If the episode is truncated an additional reward term is added:

$$r_{T+1} = \frac{-1}{1-\gamma}$$

From this (not normalized) reward, the return is then calculated and normalized in accordance with Section 3.2.3. A value of $\gamma = 0.99$ was used.

4.4.3 Agent Specification

The agents used for testing were Q-learning agents at three depths: 1 (baseline), 5, and 10. In our preliminary experiments, Q-learning was found to be able to learn this environment while naive policy-learning was not.

The agent uses the Tree Generation Network described in Section 3.1.3 to generate the state-action tree. The hidden layer sizes of the Tree Generation Network are 32, 64, and 32 (in order), and the hidden layer sizes of the Q-learning Network are 32, 64, and 32 (in order). An exploration factor of $\epsilon = 0.1$ and a buffer size of 1000 was used.

4.5 Cart Pole Environment

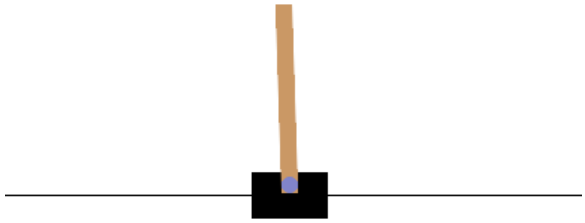

Figure 7: Screenshot of the Cart Pole Environment

As described in Gym's documentation: "[The Cart Pole] environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson in 'Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem' [22]." The environment itself consists of a cart that is constrained to one-dimensional horizontal movement and a pole that is attached to and balanced on the cart. The observation space of the environment is a four-dimensional vector consisting of the cart's horizontal position and velocity, and the pole's angle and angular velocity. The goal of the agent is to push the car either left or right at each time step and keep the pole within a $\pm 12\,\text{deg}$ angle of the upright for as long as possible. The episode truncates if 500 time steps have passed.

4.5.1 Observation Preprocessing

To keep all non-velocity observations in the range [-1, 1], the cart position is divided by 4.8 and the pole angle is divided by 0.42. The cart velocity and pole angular velocity are in the range $(-\infty, \infty)$ so they cannot be linearly mapped to the range [-1, 1], so they were unchanged.

4.5.2 Reward and Return Structure

The reward at each time step is 1. If the episode is truncated an additional reward term is added:

$$r_{T+1} = \frac{1}{1-\gamma}$$

From this (not normalized) reward, the return is then calculated and normalized in accordance with Section 3.2.3. A value of $\gamma = 0.99$ was used.

4.5.3 Agent Specification

The agents used for testing were both Q-learning agents and policy agents, each at three depths: 1 (baseline), 5, and 10.

The agent uses the Tree Generation Network described in Section 3.1.3 to generate the state-action tree. The hidden layer sizes of the Tree Generation Network are 32, 64, and 32 (in order), and the hidden layer sizes of both the Q-learning Network and Policy Network are 32, 64, and 32 (in order). An exploration factor of $\epsilon = 0$ and a buffer size of 1000 were used.

4.6 Pendulum Environment

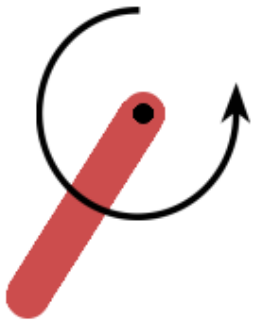

Figure 8: Screenshot of the Pendulum Environment

The pendulum environment consists of a unit-length pendulum hanging downwards about a fixed anchor. The observation space of the environment is a three-dimensional vector consisting of the pendulum's tip's x and y location, and the angular velocity of the pendulum. The goal of the agent is to apply a torque of between -2 and 2 [N m] to the free end of the pendulum to keep it as upright and still as possible. The episode truncates at 200 time steps.

4.6.1 Observation Preprocessing

To keep all observations in the range [-1, 1], the angular velocity of the pendulum is divided by 8. The rest of the observations were unchanged.

4.6.2 Reward and Return Structure

The reward at each time step is calculated as:

$$r_t = -\left(\theta^2 + 0.1\left(\frac{d\theta}{dt}\right)^2 + 0.001\tau^2\right)$$

Where θ is the angle of the pendulum, $\frac{d\theta}{dt}$ is the angular velocity of the pendulum and τ is the torque applied to the pendulum. This reward ends up being in the range $[-16.2736044, 0]$, which was the minimum possible value of the reward to the maximum possible value of the reward. To normalize this to the range $[0, 1]$ all rewards are recalculated to be:

$$r_t^{(norm)} = \frac{r_t + 16.2736044}{16.2736044}$$

Because the episode is always truncated, we add an additional term

$$r_{T+1}^{(norm)} = \frac{r_T^{(norm)}}{1-\gamma}.$$

The return is then calculated and normalized in accordance with Section 3.2.3. A value of $\gamma = 0.99$ was used.

4.6.3 Agent Specification

The agents used for testing were Q-learning agents at three depths: 1 (baseline), 5, and 10. In our preliminary experiment, Q-learning was found to be able to learn this environment while policy-learning was not.

The agent uses the Tree Generation Network described in Section 3.1.3 to generate the state-action tree. Since the action space must be discrete (because the branching method of Tree Generation creates discrete future states, one for each action), the actions were mapped as follows: action $a \in \{0, 1, 2\}$ would be mapped to $(a - 1) \times 2$ in the continuous action space.

The hidden layer sizes of the Tree Generation Network are 32, 64, and 32 (in order), and the hidden layer sizes of the Q-learning Network are 32, 64, and 32 (in order). An exploration factor of $\epsilon = 0$ and a buffer size of 400 were used.

4.7 Mountain Car Environment

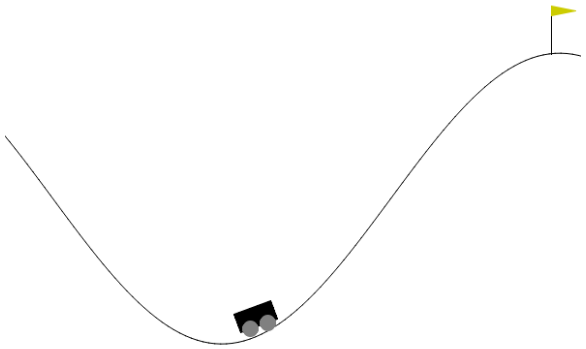

Figure 9: Screenshot of the Mountain Car Environment

This environment was first described in Andrew Moore's PhD Thesis [23]. The environment consists of a car placed at the bottom of a sinusoidal valley with the goal of reaching the top right peak by accelerating left and right. The observation space of the environment is a two-dimensional vector consisting of the car's position along the x-axis and the velocity of the car.

By default, the episode truncates after 200 time steps, but for this experiment, the environment was modified to truncate after 10,000 time steps. Since the transition rules are given for this environment, this environment serves as an experiment in seeing how the agent performs when it can perfectly predict future time steps.

4.7.1 Observation Preprocessing

Because the transition rules are given, no observation preprocessing is conducted.

4.7.2 Reward and Return Structure

The reward at each time step is -1. If the episode is truncated, an additional reward term is added:

$$r_{T+1} = \frac{-1}{1-\gamma}$$

From this (not normalized) reward, the return is then calculated and normalized in accordance with Section 3.2.3. A value of $\gamma = 0.99$ was used.

4.7.3 Agent Specifications

Both Q learning and policy agents were trained, each at depths 1 (baseline), 5, and 10. The state-action tree is generated via the transition rules provided:

$$v_{t+1} = \text{clamp}\left(v_t + 0.001(a_t - 1) - 0.0025\cos(3 * x_t), -0.07, 0.07\right).$$

$$x_{t+1} = x_t + v_{t+1}.$$

One exceptional case is when $x_t < -1.2$,

$$x_t = 1.2 \text{ and } v_t = 0.$$

The hidden layer sizes of both the Q-learning Network and Policy Network are 32, 64, and 32 (in order). An exploration factor of $\epsilon = 0$ and a buffer size of 20,000 was used.

CHAPTER 5

Results and Discussion

5.1 Results

5.1.1 Acrobot Environment

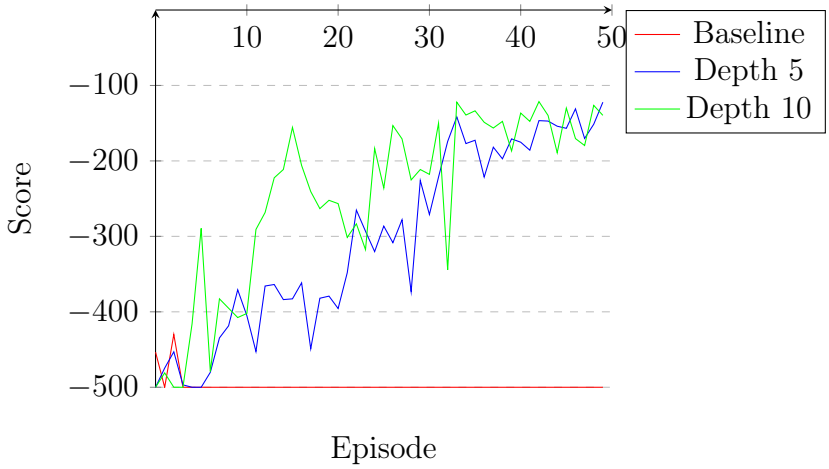

Figure 10: Q-Learning Average Performances (Duration vs Episode)

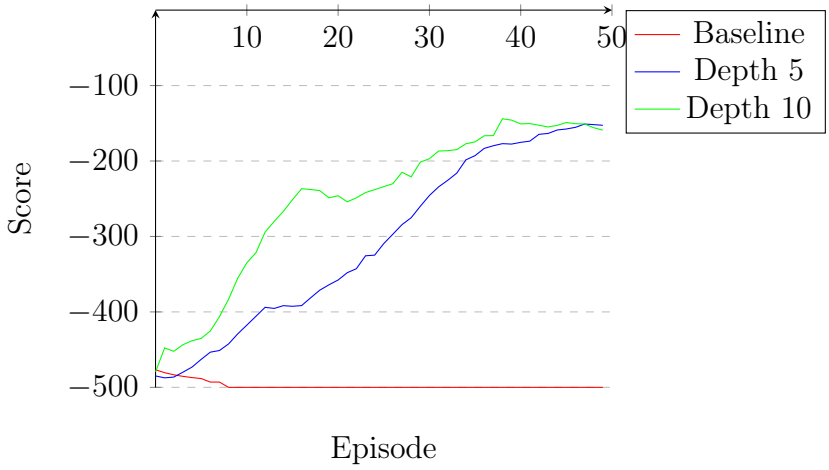

Figure 11: Q-Learning Moving Average Performances (Duration vs Episode)

In the Acrobot environment, only a Q-learning agent was run. The higher depth agents were able to learn the environment and even begin to optimize it while the baseline agent was not. This is likely indicative that environments with sparse rewards

which are only attainable after a longer sequence of actions are easier to learn at higher depths. Between the two higher depth agents, the Depth 10 agent begins to learn faster even if it hits the same end performance as the Depth 5 agent.

5.1.2 Cartpole Environment

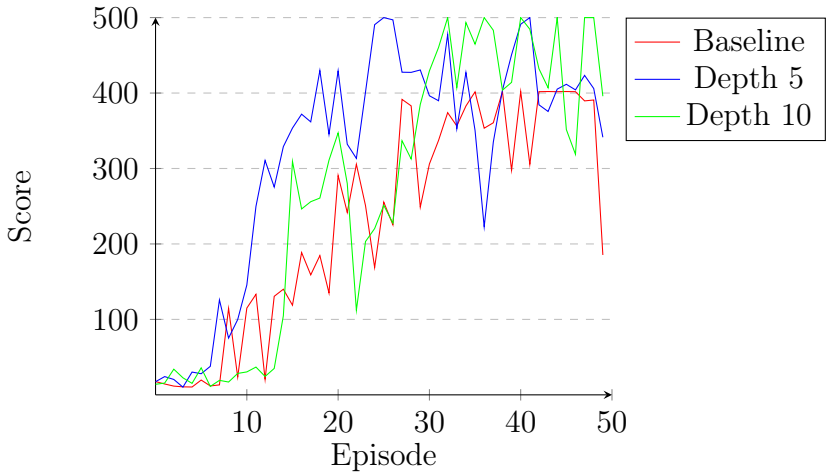

Figure 12: Q-Learning Average Performance (Duration vs Episode)

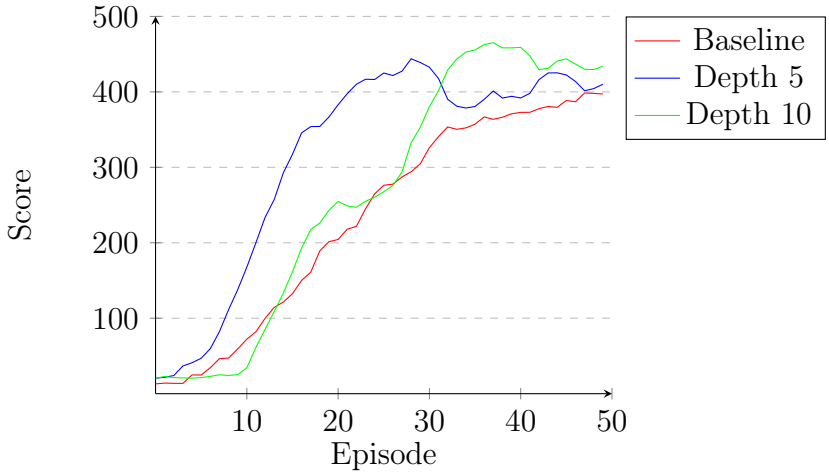

Figure 13: Q-Learning Moving Average Performance (Duration vs Episode)

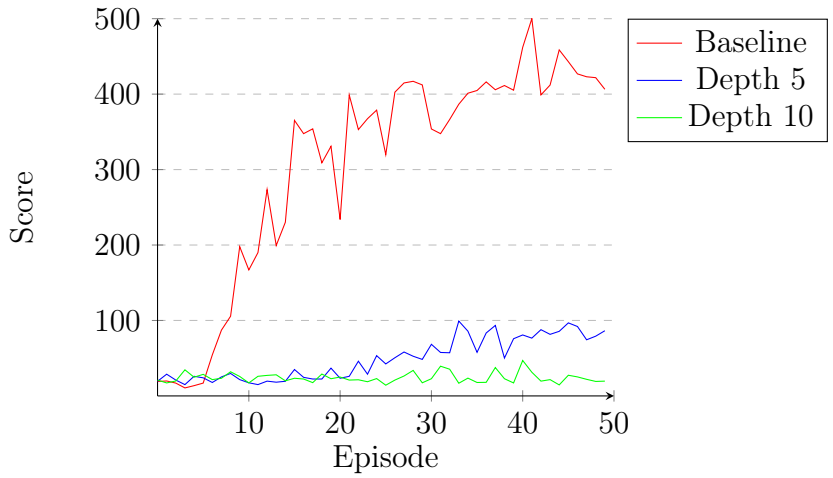

Figure 14: Policy Average Performance (Duration vs Episode)

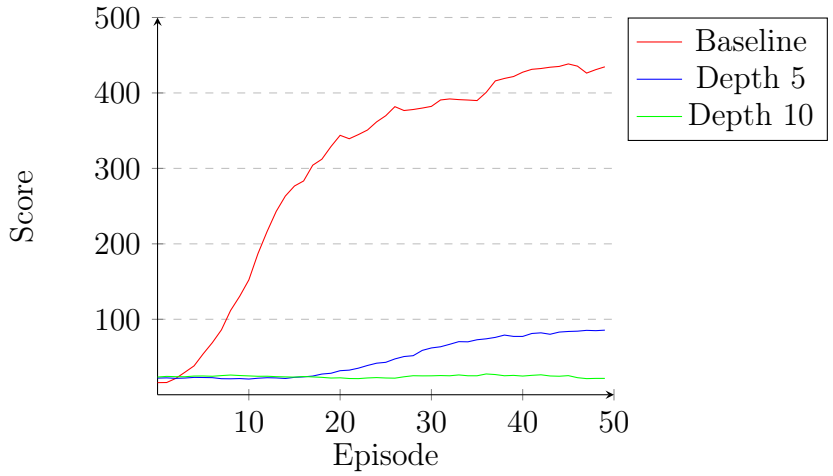

Figure 15: Policy Moving Average Performance (Duration vs Episode)

For the cartpole environment, both Q-learning and policy agents were run. The higher depth Q-learning agents were able to avoid the terminal state for longer than the baseline agent, even though all agents were able to learn the environment to some extent. The Depth 10 agent seemed to be the most stable at their final performance. This may indicate that the stacking of predictions at higher depths may introduce some instability.

The policy agents were completely ineffective except for the baseline agent, which learned moderately well. This could be indicative that high depth policies are not suitable for tasks that involve avoiding terminal states. For example, in this environment, the goal is to keep the pole upright on the cart but the termination condition for the episode is that the pole's angle falls outside of a desired range, so the agent's task is to avoid those terminal states where the pole's angle is out of range.

5.1.3 Pendulum Environment

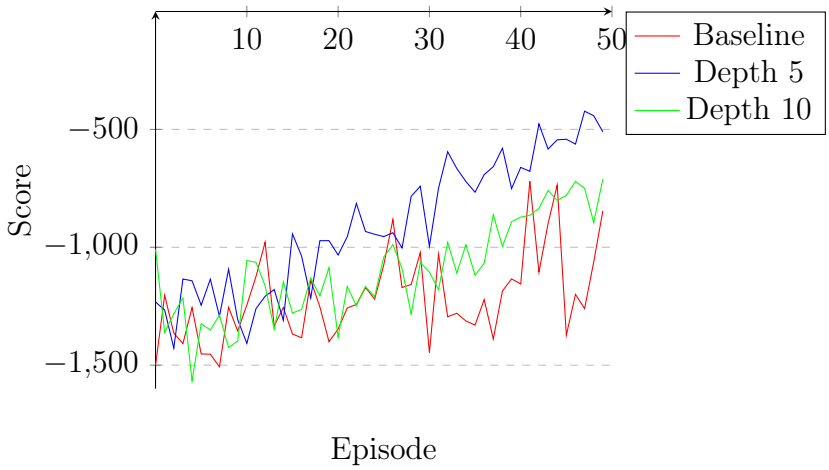

Figure 16: Q-Learning Average Performance (Score vs Episode)

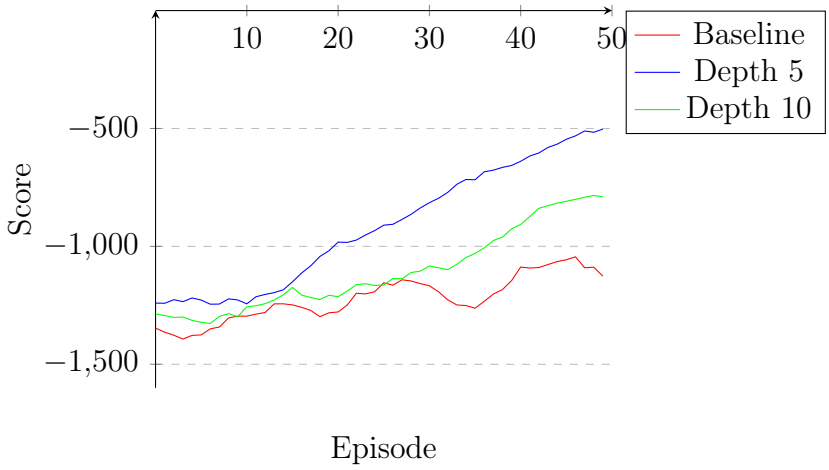

Figure 17: Q-Learning Moving Average Performance (Score vs Episode)

In the Pendulum environment, only the Q-learning agents were run. The baseline agent and the Depth 10 agent performed similarly, but the Depth 10 agent was more stable and seemed to actually learn towards the end, indicating its potential for long-term performance. Of note though, is that the Depth 5 agent significantly outperformed the other two agents. This supports the previous results that the excessive use of higher Depth may require longer training to perform as well as lesser Depth agents which can easily learn. This environment shows that the Tree Generative agents can outperform baseline agents in dense reward environments as well.

5.1.4 Mountain Car Environment

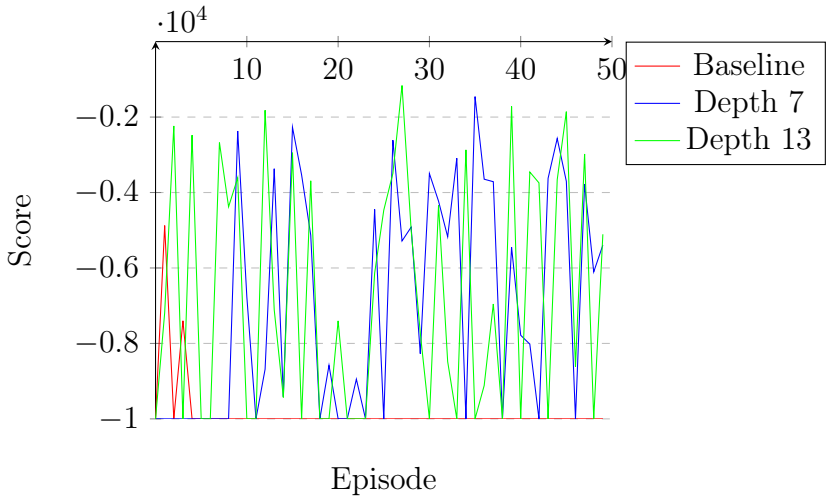

Figure 18: Policy Performance (Score vs Episode)

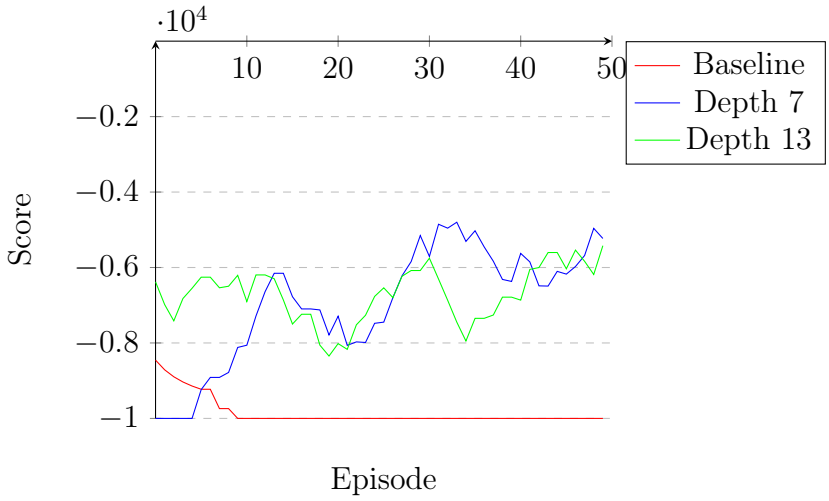

Figure 19: Policy Moving Average Performance (Score vs Episode)

In the Mountain Car environment, only the policy agent was run. The baseline agent was not able to learn the task at all, while the higher depth agents were able to sporadically succeed in it. By looking at Fig 19 indicating the moving averages, the Depth 7 agent was learning to perform better while the Depth 13 agent performed better from the start. Both agents performed roughly the same towards the end. This shows that, without the instability of compounded state predictions and with knowledge of terminal states, higher depth agents find it easier to learn the environment. This is likely because the agents need only to guide themselves towards a subset of states that are *within reach* of the target states. As depth increases, this subset grows exponentially by branching out, making it easier to find terminal states.

5.2 Discussion

Tree Generation agents were able to demonstrate superior capability of learning tasks that involved long sequences of actions. This indices that the search ability of the Tree Generation agent is capable of finding desirable states and reaching them. This also demonstrates that the state predictions are good enough to be used as a search.

The Tree Generation agents were also able to demonstrate the capability of learning sparse reward environments. This indicates that the agent can differentiate different action sequences from each other and identify the ones that lead to rewards. As an example, with a breadth factor of β, a depth 1 (baseline) agent can differentiate β paths, while a depth D agent can differentiate β^D paths, allowing it to more effectively identify which paths will lead to rewards.

In the Mountain Car environment, which had transition rules available to the agent, the Tree Generation agents were able to outperform the baseline agent, showing the agents are able to utilize the perfect information to learn the environment. This shows that the Tree Generation agent can also be used in environments where the transition function is known to improve performance.

One weakness of Tree Generation policy agent is that it was unable to learn to avoid terminal states, but it was able to reach terminal states well. This is likely because the agent probabilistically samples action sequences, which means that it's more difficult to avoid a large sample of terminal states because the probability of all of them must be low, but to reach a desirable terminal state, only a few actions sequence probability must be high. However, the Q-learning Tree Generation agent was able to perform well on the same environments, show that there is still promise for the Tree Generation agent even in these kinds of environments.

CHAPTER 6

Conclusion

Reinforcement learning is an important field of machine learning that deals with agents learning to complete tasks in given environments. Reinforcement learning agents often struggle with learning long or complex tasks in potentially sparse reward environments, especially when the transition function of the environment is unknown or unfeasible to compute. This paper proposes a novel Tree Generation agent which approximates the transition function of an MDP to generate a state-action tree which is then evaluated and searched to find ideal action sequences to take, thereby allowing it to explore and learn sparse reward environments more efficiently as well as learn complex tasks more efficiently. By testing the agent on four environments, we are able to demonstrate its capabilities of learning both sparse reward environments and environments that require long action sequences to complete successfully. It also demonstrated that its performance was not deteriorated compared to baseline agents in dense reward environments. Additionally, even when the state transition function was known, the Tree Generation agent outperformed the baseline agent, showing that the tree search is helpful whether an approximation or absolute.

For future work, the Tree Generation agent should be tested on environments with more complex state spaces, such as those with visual state spaces. This would require a more complex tree generation network than the one proposed in this paper. Additionally, utilizing tree generation on top of existing state-of-the-art reinforcement learning agents is a topic of interest for future study. Since the Tree Generation agent was able to improve upon naive reinforcement learning baselines, it would be interesting to see if these performance upgrades would apply to more advanced agents.

LIST OF REFERENCES

[1] 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.

[2] O. Vinyals, I. Babuschkin, W. Czarnecki, *et al.*, ''Grandmaster level in starcraft ii using multi-agent reinforcement learning,'' *Nature*, vol. 575, pp. 350--354, 2019.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, ''A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,'' *Science*, vol. 362, no. 6419, pp. 1140--1144, 2018.

[4] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, 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.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, ''Playing atari with deep reinforcement learning,'' *arXiv preprint arXiv:1312.5602*, 2013.

[6] A. Trott, S. Zheng, C. Xiong, and R. Socher, ''Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards,'' *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[7] F. Memarian, W. Goo, R. Lioutikov, S. Niekum, and U. Topcu, ''Self-supervised online reward shaping in sparse-reward environments,'' in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 2369--2375.

[8] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, ''Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards,'' *arXiv preprint arXiv:1707.08817*, 2017.

[9] V. G. Goecks, G. M. Gremillion, V. J. Lawhern, J. Valasek, and N. R. Waytowich, ''Integrating behavior cloning and reinforcement learning for improved performance in dense and sparse reward environments,'' *arXiv preprint arXiv:1910.04281*, 2019.

[10] C. J. Watkins and P. Dayan, ''Q-learning,'' *Machine learning*, vol. 8, no. 3, pp. 279--292, 1992.

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

[12] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, ''Deterministic policy gradient algorithms,'' in *International conference on machine learning*. PMLR, 2014, pp. 387--395.

[13] 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.

[14] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, ''A survey of monte carlo tree search methods,'' *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1--43, 2012.

[15] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, ''The graph neural network model,'' *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61--80, 2008.

[16] T. N. Kipf and M. Welling, ''Semi-supervised classification with graph convolutional networks,'' *arXiv preprint arXiv:1609.02907*, 2016.

[17] K. He, X. Zhang, S. Ren, and J. Sun, ''Deep residual learning for image recognition,'' in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770--778.

[18] Y. Tsuruoka, D. Yokoyama, and T. Chikayama, ''Game-tree search algorithm based on realization probability,'' *Icga Journal*, vol. 25, no. 3, pp. 145--152, 2002.

[19] D. P. Kingma and J. Ba, ''Adam: A method for stochastic optimization,'' *arXiv preprint arXiv:1412.6980*, 2014.

[20] R. S. Sutton, ''Generalization in reinforcement learning: Successful examples using sparse coarse coding,'' *Advances in neural information processing systems*, vol. 8, 1995.

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

[22] A. G. Barto, R. S. Sutton, and C. W. Anderson, ''Neuronlike adaptive elements that can solve difficult learning control problems,'' *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834--846, 1983.

[23] A. W. Moore, ''Efficient memory-based learning for robot control,'' University of Cambridge, Tech. Rep., 1990.