A DIFFERENTIABLE TREE SEARCH ALGORITHM

A.1 DIFFERENTIABLE TREE SEARCH PSEUDO-CODE

Algorithm 1: Differentiable Tree Search (DTS)

```
Input: Input state, $s_{root}$
Result: Q-values, $Q(s_{root}, a)$
$h_{root} \leftarrow \mathcal{E}_\theta(s_{root})$ ;                //Encode $s_0$ to its latent state $h_0$
$node_{root} \leftarrow initialise(h_{root})$ ;                   //Initialise root node
$Tree \leftarrow \{node_{root}\}$ ;                                  //Initialise Tree
$Open \leftarrow \{node_{root}\}$ ;                    //Initialise candidate set $Open$
// Expansion phase
$trial \leftarrow 0$;
repeat
    foreach $node \in Open$ do
        $h_{node} \leftarrow getLatent(node)$ ;                //Get latent state of $node$
        $\bar{V}(node) \leftarrow sumOfRewards(node) + \mathcal{V}_\theta(h_{node})$
    // Compute the tree expansion policy $\pi_{tree}$
    $\pi_{tree} \leftarrow \text{softmax}_n\Big(\bar{V}(n)\Big)$;
    // Sample the node to expand from $\pi_{tree}$
    $node^* \leftarrow \text{sample}(\pi_{tree})$;
    // Expand $node^*$ using Transition module $\mathcal{T}_\theta$ and calculate
       rewards using $\mathcal{R}_\theta$
    foreach $a \in Actions;$ do
        $h_{child} \leftarrow T_\theta(h_{node^*}, a)$;
        $r_{child} \leftarrow \mathcal{R}_\theta(h_{node^*}, a)$;
        $createNode(h_{child}, r_{child})$;
    // Update $Open$ with children of $node^*$ and remove $node^*$:
    $Open \leftarrow Open \cup \{child_a | child_a = getChild(node^*, a); \forall a \in A\} - node^*$;
    $trial \leftarrow trial + 1$;
until $trial$ < $MAX\_TRIALS$;
// Backup phase
foreach $node \in Tree$, iterating from leaf nodes to the root node; do
    if node is a leaf; then
        $h_{node} \leftarrow getLatent(node)$ ;                //Get latent state of $node$
        $V(node) = \mathcal{V}_\theta(h_{node})$ ;          //Compute value using Value module
    else
        // Backup Q-value estimates
        foreach $a \in Actions;$ do
            $node_{child[a]} \leftarrow getChild(node, a)$ ;         //Get child of $node$ that
             corresponds to action $a$
            $h_{node} \leftarrow getLatent(node)$ ;            //Get latent state of $node$
            $r_{node} \leftarrow \mathcal{R}_\theta(h_{node}, a)$ ;        //Get reward using Reward module
            $Q(node,\ a) \leftarrow r_{node} + V\left(node_{child[a]}\right)$
        $V(node) \leftarrow \max_a Q(node, a)$
// Return Q-value of the root node
return $Q(node_{root})$
```

A.2 DIFFERENTIABLE TREE SEARCH PSEUDO-CODE EXPLANATION

The Differentiable Tree Search (DTS) algorithm operates in two main phases: Expansion and Backup.

1. **Initialisation**: Given an input state s_{root}, the algorithm begins by encoding this state into its latent representation h_{root} using an encoder $\mathcal{E}_\theta$. This latent representation serves as the root node of the search tree.
2. **Expansion Phase**:
    - The algorithm initiates a set of candidate nodes, termed 'Open', starting with the root node.
    - In each trial, the algorithm considers every node in the 'Open' set, retrieves its latent state, and computes an interim value $\bar{V}(\text{node})$, which combines the cumulative rewards of the node's path with a value estimation from the value module $\mathcal{V}_\theta$.
    - Using the path values of nodes in 'Open', the tree expansion policy, π_{tree}, is computed. From this policy, a node, $node^*$, is sampled for expansion.
    - Each possible action from $node^*$ results in the creation of a child node. This is achieved by leveraging the transition module $\mathcal{T}_\theta$ to predict the latent state of the child and the reward module $\mathcal{R}_\theta$ to determine the associated reward. After expansion, $node^*$ is removed from 'Open' and its children are added.
    - This expansion process continues until a pre-defined number of trials, `MAX_TRIALS`, is reached.
3. **Backup Phase**:
    - Starting from the leaf nodes, the algorithm backpropagates value estimates to the root.
    - For each leaf node, its value is directly computed from the value module $\mathcal{V}_\theta$. For non-leaf nodes, the Q-value for each action is estimated by combining the reward for that action with the value of the corresponding child node.
    - The value of a non-leaf node is set to the maximum Q-value among its actions.
4. **Result**: The algorithm finally returns the Q-values associated with the root node, $Q(node_{\text{root}})$, providing an estimation of the value of taking each action from the initial state.

This explanation provides a high-level view of the DTS algorithm's operation. By breaking down the search process into expansion and backup phases, the pseudo-code highlights how DTS incrementally builds the search tree and then consolidates value estimates back to the root.

B CONTINUITY OF Q-FUNCTION

The efficient optimisation of DTS parameters via gradient descent necessitates that the loss function is continuous with respect to the network's parameters. We begin by showing that the Q-value at the root node of a search tree, computed by expanding a tree fully to a fixed depth d and backpropagating the value using Bellman equation, is continuous.

Suppose we have two functions $f(x)$ and $g(x)$ which are continuous at any point c in their domains.

Lemma B.1. ***Continuity of Composition***: *The composition of two continuous functions, denoted as $f(g(x))$, retains continuity. (Theorem 4.7 in Rudin (1976))*

Lemma B.2. ***Continuity of Sum operation***: *The result of adding two continuous functions, expressed as $f(x) + g(x)$, is a continuous function. (Theorem 4.9 in Rudin (1976))*

Lemma B.3. ***Continuity of Max operation***: *Applying Max over two continuous functions, expressed as $\max(f(x), g(x))$, results in a function that is continuous.*

Proof. Consider a function $h(x) = \max(f(x), g(x))$, and we aim to demonstrate that $h(x)$ is continuous. Now, $h(x)$ can be expressed as a combination of continuous functions:

$$h(x) = \frac{f(x) + g(x) + |f(x) - g(x)|}{2}$$

Since sums and absolute values of continuous functions are continuous (Rudin, 1976), $h(x)$ is continuous. Hence, the maximum of two continuous functions is also a continuous function. ☐

Theorem B.1. *Given a set of parameterised submodules that are continuous within the parameter space, expanding a tree fully to a fixed depth 'd' by composing these modules and computing the Q-values by backpropagating the children values using addition and max operations, is continuous.*

Proof. Let us represent the set of parameters, encoder module, transition module, reward module, and value module respectively as θ, $\mathcal{E}_\theta$, $\mathcal{T}_\theta$, $\mathcal{R}_\theta$, and $\mathcal{V}_\theta$. These submodules are assumed to be composed of simple neural network architectures, comprising linear and convolutional layers, with the Rectified Linear Unit (ReLU) serving as the activation function. These submodules, therefore, are continuous within the parameter space.

We can subsequently rewrite the Q-value as the output of a full tree expansion as follows:

$$\begin{aligned} Q(s_0, a_0) &= Q(h_0, a_0) \\ &= r_0 + V(h_1) \end{aligned}$$

where,

$$h_0 = \mathcal{E}_\theta(s_0) \tag{11}$$

$$r_t = \mathcal{R}_\theta(h_t, a_t) \tag{12}$$

$$h_{t+1} = \mathcal{T}_\theta(h_t, a_t) \tag{13}$$

$$Q(h_t, a_t) = \mathcal{R}_\theta(h_t, a_t) + V(h_{t+1}) \tag{14}$$

$$V(h_t) = \begin{cases} \mathcal{V}_\theta(h_t) & \text{if } h_t \text{ is a leaf} \\ \max_a \left(Q(h_t, a)\right) & \text{otherwise} \end{cases} \tag{15}$$

Given that $\mathcal{E}_\theta$, $\mathcal{R}_\theta$, and $\mathcal{T}_\theta$ are continuous, h_0, r_t, and h_{t+1} in Equations 11 and 13 are similarly continuous (derived from Lemma B.1).

Further, $V(h_t)$ in Equation 15 can either be $\mathcal{V}_\theta(h_t)$, if h_t is a leaf node, or $\max_a(Q(h_t, a))$ otherwise. In the first scenario, $V(h_t)$ retains continuity by assumption. In the second scenario, if $Q(h_t, a_t)$ is continuous, then $V(h_t)$ remains continuous as per Lemma B.3.

Now, we show that $Q(h_t, a_t)$ is continuous using a recursive argument that for any node in the search tree, if the Q-values of all its child nodes are continuous, then its Q-value is also continuous. Q-value of an internal tree node h_t can be written as $Q(h_t, a_t) = \mathcal{R}_\theta(h_t, a_t) + V(h_{t+1})$, where $h_{t+1} = \mathcal{T}_\theta(h_t, a_t)$ is the child node of h_t. Considering the base case, when the h_{t+1} is a leaf node, then $Q(h_t, a_t) = \mathcal{R}_\theta(h_t, a_t) + \mathcal{V}_\theta(h_{t+1})$, which is continuous as per Lemma B.2. Consequently, $V(h_t)$ maintains continuity. Applying this logic recursively, the Q-value $Q(h_t, a_t)$ of all the tree nodes maintains continuity.

Thus, we can decompose the Q-value at the root node, $Q(s_0, a_0)$, as a composition of continuous functions, ensuring that the output Q-value, $Q(s_0, a_0)$, is continuous. ☐

In the preceding theorem, we demonstrated the continuity of the Q-value, which is the output of a full tree expansion. When, this Q-value is used as the input to a continuous loss function, the resulting loss is continuous in the network's parameter space.

C DERIVATION OF THE GRADIENT OF THE EXPECTED LOSS FUNCTION

Consider the Q-values predicted by DTS as $Q_\theta(s, a|\tau)$. This output is contingent upon the final tree, τ, sampled after T trials during the online search. Let's denote the corresponding loss function on this output Q-value as $\mathcal{L}\Big(Q_\theta(s, a|\tau)\Big)$. Our objective is to compute the gradient of the expected loss value, averaging over trees sampled.

Let us represent the Q-values predicted by DTS as $Q_\theta(s, a|\tau)$, which depends on the final tree τ sampled after T trials of the online search. Let us denote the corresponding loss function on this output Q-value as $\mathcal{L}\Big(Q_\theta(s, a|\tau)\Big)$. Our objective is to compute the gradient of the expected loss value, averaging over trees sampled.

The gradient of expected loss, considering the expectation over the sampled trees, is derived as:

$$\begin{aligned}
\mathcal{L} &= \mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
\nabla_\theta\mathcal{L} &= \nabla_\theta\mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
&= \nabla_\theta\sum_\tau \pi_\theta(\tau)\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big) \\
&= \sum_\tau \nabla_\theta\Big[\pi_\theta(\tau)\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
&= \sum_\tau \mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\nabla_\theta\pi_\theta(\tau) + \sum_\tau \pi_\theta(\tau)\nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big) \\
&= \sum_\tau \pi_\theta(\tau)\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\nabla_\theta\log\pi_\theta(\tau) + \sum_\tau \pi_\theta(\tau)\nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big) \\
&= \mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\nabla_\theta\log\pi_\theta(\tau) + \nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
&= \mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\nabla_\theta\log\prod_{t=1}^{T}\pi_\theta(n_t|\tau_t) + \nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
&= \mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(n_t|\tau_t) + \nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big]
\end{aligned}$$

Leveraging the telescoping sum trick, as elaborated in Section 3.5, the gradient of the expected loss can be rewritten as a lower-variance estimate:

$$\begin{aligned}
\nabla_\theta\mathcal{L} &= \mathbb{E}_\tau\Big[\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(n_t|\tau_t) + \nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big] \\
&= \mathbb{E}_\tau\Big[\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(n_t|\tau_t)R_t + \nabla_\theta\mathcal{L}\Big(Q_\theta(s,a|\tau)\Big)\Big]
\end{aligned}$$

where

$$\begin{aligned}
R_t &= \sum_{i=t}^{T} r_i = \mathcal{L}_T - \mathcal{L}_{t-1} \\
\mathcal{L}_t &= \mathcal{L}\Big(Q_\theta(s,a|\tau_t)\Big), \text{ representing the loss value after the } t^{th} \text{ search trial.}
\end{aligned}$$

In practice, we utilise the single-sample estimate for the expected gradient, as elaborated in Schulman et al. (2015a)

D LOSS FUNCTIONS

Differentiable Tree Search (DTS) can be used as a drop-in replacement for commonly used Convolutional Neural Network (CNN) based architectures. In this paper, we train DTS using offline reinforcement learning to test its sample complexity and generalisation capabilities. The training dataset, $\mathcal{D}$, is represented by a series of tuples, each comprising the state observed, action taken, reward observed, and the target Q-value of the observed state, denoted as $(s_{t,i}, a_{t,i}, r_{t,i}, Q_{t,i})$. These tuples are generated during the trajectory i simulated by the optimal policy π^*. As the agent is following the optimal policy, we can compute the target Q-value for state $s_{t,i}$ by adding the rewards obtained in the i^{th} trajectory from timestep t onwards.

$$\begin{aligned} Q_{t,i} &= Q^{\pi^*}(s_{t,i}) \\ &= r_{t,i} + r_{t+1,i} + r_{t+2,i} + ... r_{T,i} \\ &= \sum_{j=t}^{T} r_{j,i} \end{aligned}$$

Our approach involves training the Q-values, as computed by DTS, to approximate the target Q-values for the corresponding observed states and actions. We aim to minimise the difference between the predicted Q-values and the target Q-value by employing the mean squared error as the loss function. The loss is formally defined as:

$$\mathcal{L}_Q = \mathbb{E}_{(s_{t,i},\ a_{t,i},\ Q_{t,i}) \sim \mathcal{D}} \left[\Big(Q_\theta(s_{t,i},\ a_{t,i}) - Q_{t,i} \Big)^2 \right] \tag{16}$$

To further support the learning process, we also incorporate auxiliary loss functions. Given that we utilise a limited training data, the expert is capable of covering only a limited part of the state-action distribution. Consequently, the Q-values of out-of-distribution actions could potentially be overestimated, as observed in (Kumar et al., 2020). To rectify this issue, we introduce an additional loss function in accordance with Conservative Q-learning (CQL) (Kumar et al., 2020), thereby encouraging the agent to follow the actions observed in the training data distribution.

$$\mathcal{L}_{\mathcal{D}} = \mathbb{E}_{(s_{t,i},\ a_{t,i}) \sim \mathcal{D}} \left[\log \sum_a \exp \Big(Q_\theta(s_{t,i},\ a) \Big) - Q_\theta(s_{t,i},\ a_{t,i}) \right] \tag{17}$$

During the online search, the transition, reward, and value networks operate on the latent states. Consequently, it is imperative to ensure that the input to these networks is of a consistent scale, as suggested in (Schwarzer et al., 2021; Ye et al., 2021). To achieve this, we apply Tanh normalisation on the latent states, thereby adjusting their scale to fall within the range $(-1, 1)$.

$$\begin{aligned} h &= tanh(x) \\ &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \in (-1, 1) \end{aligned}$$

Further, in order to avoid overburdening the latent states with extraneous information required to reconstruct the original input states like in Model-based RL methods, we utilise self-supervised consistency loss functions as described in (Schwarzer et al., 2021; Ye et al., 2021). These functions aid in maintaining consistency within the transition and reward networks. For example, let us assume a state $s_{t,i}$ and the subsequent state $s_{t+1,i}$ resulting from action $a_{t,i}$. The latent state representations for the environment states $s_{t,i}$ and $s_{t+1,i}$ can be computed as $h_{t,i}$ and $h_{t+1,i}$ respectively. The latent state encoding $\hat{h}_{t+1,i}$ can be predicted using the transition module, $\hat{h}_{t+1,i} = \mathcal{T}_\theta(h_{t,i},\ a_{t,i})$. We minimise the squared error between the latent representation $h_{t+1,i}$ and $\hat{h}_{t+1,i}$, to ensure that the transition function $\mathcal{T}_\theta$ provides consistent predictions for the transitions in the latent space. In accordance with the approach detailed in (Ye et al., 2021), we use a separate encoding network, referred to as the target encoder, to compute target representations.

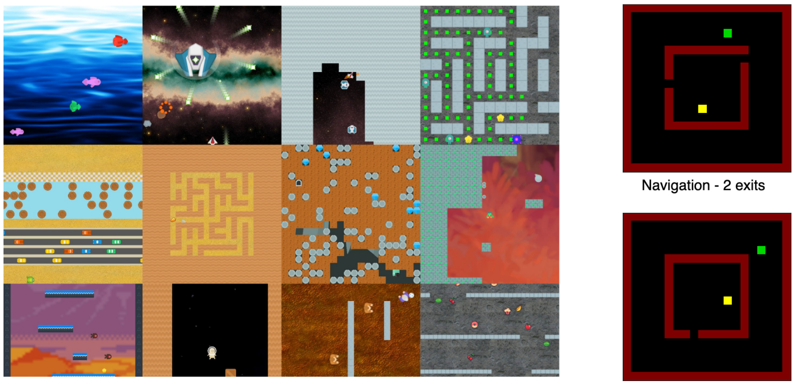
$$\mathcal{L}_{\mathcal{T}_\theta} = \mathbb{E}_{(s_t,\ a_t,\ s_{t+1}) \sim \mathcal{D}} \left[\Big(\hat{h}_{t+1,i} - h_{t+1,i} \Big)^2 \right] \tag{18}$$

where

$$\begin{aligned} &\hat{h}_{t+1,i} = \mathcal{T}_\theta(h_{t,i},\ a_{t,i}); \quad h_{t,i} = \mathcal{E}_\theta(s_{t,i}) \\ &h_{t+1,i} = \mathcal{E}_{\theta'}(s_{t+1,i}) \end{aligned}$$

The parameters of the target encoder, θ', are updated using an exponential moving average of the parameters of the base encoder, θ.

$$\theta'_{i+1} \leftarrow \alpha\ \theta'_i + (1 - \alpha)\ \theta_{i+1}$$



Procgen Games

Navigation - 2 exits

Navigation - 1 exit

Figure 1: An illustrative example showcasing the Procgen games (on the left) juxtaposed with the Grid Navigation task (on the right).

Notably, we refrain from adding projection or prediction networks, as done in Schwarzer et al. (2021); Ye et al. (2021), prior to calculating the squared difference.

Lastly, we also seek to minimise the mean squared error between the predicted reward $\mathcal{R}_\theta(h_{t,i}, a_{t,i})$ and the actual reward observed $r_{t,i}$ in the training dataset $\mathcal{D}$.

$$\mathcal{L}_{\mathcal{R}_\theta} = \mathbb{E}_{(s_{t,i},\ a_{t,i},\ r_{t,i})\sim\mathcal{D}}\left[\left(\mathcal{R}_\theta(h_{t,i},\ a_{t,i}) - r_{t,i}\right)^2\right] \tag{19}$$

The cumulative loss for imitation learning is hence given by:

$$\mathcal{L} = \lambda_1\mathcal{L}_Q + \lambda_2\mathcal{L}_\mathcal{D} + \lambda_3\mathcal{L}_{\mathcal{T}_\theta} + \lambda_4\mathcal{L}_{\mathcal{R}_\theta} \tag{20}$$

where $\lambda_1, \lambda_2, \lambda_3$ and λ_4 serve as weighting hyperparameters.

E EXPERIMENT DETAILS

E.1 TEST DOMAINS

In our evaluations, we choose two distinct domains to assess and compare the sample complexity and generalisation capabilities of various methods: the Procgen games and a grid navigation task. These domains are selected due to their diverse challenges, offering a comprehensive evaluation spectrum. A visual representation of these domains can be viewed in Figure 1.

E.1.1 GRID NAVIGATION

The grid navigation task serves as a foundational test that mimics the challenges a robot might face when navigating in a 2D grid environment. This environment provides both a quantitative metric and a qualitative visualisation to understand an agent's capacity to generalise its policy. Specifically, this task involves a 20×20 grid with a central hall. At the beginning of each episode, the robot is positioned at a random point within this central hall. Simultaneously, a goal position is sampled randomly at a location outside the hall, challenging the robot to find its way out and reach this target. There are two variations of this task. The first provides the robot with a single exit from the central hall, while the second offers two exits. The single-exit hall scenario is similar to the two-exit scenario but requires a longer-horizon planning to successfully evade the walls to exit the hall and reach the goal.

E.1.2 PROCGEN

Procgen is a unique suite consisting of 16 game-like environments, each of which is procedurally generated. This means that they are designed to present slightly altered challenges every time they are played. Such design intricacy makes Procgen an ideal choice to test an agent's generalisation capabilities. It stands in contrast to other commonly used testing suites, like the renowned Atari 2600 games (Mnih et al., 2013; 2016; Badia et al., 2020). The diverse array of environments within Procgen emphasises the pivotal role of robust policy learning. The environmental diversity in Procgen underlines the importance of robust policy learning for successful generalisation. The open-source code for the environments is publicly accessible at `https://github.com/openai/procgen`.

E.2 TRAINING SETUP

Differentiable Tree Search (DTS) is a drop-in replacement for conventional Convolutional Neural Network (CNN) architectures and can be trained using both online and offline reinforcement learning algorithms. In this paper, we employ the offline reinforcement learning (Offline-RL) framework to focus on the sample complexity and generalisation capabilities of DTS when compared with the baselines. Offline-RL, often referred to as batch-RL, is the scenario wherein an agent learns its policy solely from a fixed dataset of experiences, without further interactions with the environment.

However, the Offline-RL framework does present significant challenges, especially when it comes to the generalisation capabilities of methods, even for seemingly straightforward tasks like navigation. Consider, for instance, that the optimal policy is employed to collect experiences from an environment. It would predominantly select the optimal actions at every state. This means that only a fraction of the vast state-action space would be covered in the dataset. As a result, the model might not learn about many interactions, such as what happens when it collides with a wall, due to the limited training data on such environment interactions. When this world model is then applied in an online search scenario, the search process might unknowingly venture into out-of-distribution state space. These explorations, stemming from the model's limited generalisation capabilities, can lead to overly optimistic value predictions, subsequently affecting the Q-values computation at the root node. In practical terms, the online search might mistakenly believe it can travel through a wall to reach its goal more quickly and would then run into it.

However, the DTS design offers a solution to this problem by jointly optimising both the world model and the online search. During the training phase, when the online search strays into the out-of-distribution states, it might overestimate the value of these states. This overestimation would then influence the final output after the search. When such a mismatch between the post-search output and the expert action is detected, the gradient descent algorithm adjusts these overestimated values, effectively lowering them to align the final Q-values more closely with the expert action. As a direct consequence of this, by the end of the training phase, the search process will have effectively learned to ignore these the out-of-distribution states.

E.3 TRAINING DATA COLLECTION

E.3.1 NAVIGATION

For our navigation task, which is relatively small in size, we are able to compute the optimal policy for any given state and configuration. We employ the value iteration algorithm for this purpose, as detailed by Sutton & Barto (2018). At the beginning of each episode, we formulate a random passage through the central hall. Subsequently, we also randomly determine the starting position of the robot within the hall and its goal position outside of it. Taking a cue from our approach with Procgen, we collect a total of 1000 expert trajectories for training. Each of these trajectories incorporates a sequence comprising states, actions, rewards, and Q-values observed throughout the episode.

E.3.2 PROCGEN

When it comes to Procgen, even though there isn't a public repository of pre-trained models, there exists an open-source code base for Phasic Policy Gradient (PPG) (Cobbe et al., 2021). With this in hand, we could effectively train a robust policy for every individual Procgen game starting from the ground up. We rely on the default set of hyperparameters for training a specific policy for each game. Once equipped with these trained policies, we treat them as expert agents. Using these expert policies, we then gather expert trajectories for 1000 successful episodes. Just like in the case of Navigation, each of these trajectories represents a sequence comprising states, actions, rewards, and Q-values observed throughout the episode.

E.4 IMPLEMENTATION DETAILS

In an effort to assess the distinctive elements of each agent's design, we ensure uniformity in the number of parameters across all agents. This is achieved by integrating the submodules from DTS into the network architecture of every agent. However, while the number of parameters are consistent, the way in which these submodules are utilised to construct the computation graph varies among agents.

E.4.1 DTS

DTS utilises the submodules in alignment with the algorithm presented in Section A. For our empirical evaluations, we set the maximum limit for search trials at 10. Throughout the training process, the computation graph, formulated via online search, is optimised to accurately predict the Q-values. This optimisation serves a dual purpose: it not only refines the Q-value predictions but also facilitates robust learning for the submodules when they are employed in context of online search. The loss function designated for DTS training is:

$$\mathcal{L}_{DTS} = \lambda_1 \mathcal{L}_Q + \lambda_2 \mathcal{L}_{\mathcal{D}} + \lambda_3 \mathcal{L}_{\mathcal{T}_\theta} + \lambda_4 \mathcal{L}_{\mathcal{R}_\theta}$$

E.4.2 MODEL-FREE Q-NETWORK

In this baseline, the submodules are utilise to perform a one-step look-ahead search. The input state undergoes an expansion using the world model, and Q-values are computed using the Bellman equation represented as $Q(s, a) = Rew(h, a) + Val(h')$, where $h = Enc(s)$ and $h' = Tr(h, a)$. Intriguingly, this structure does encapsulate a basic inductive bias via the 1-step look-ahead search. However, in keeping with its model-free characteristic, auxiliary losses aren't employed for training the transition and reward model. The loss function for this model is:

$$\mathcal{L}_{Qnet} = \lambda_1 \mathcal{L}_Q + \lambda_2 \mathcal{L}_{\mathcal{D}}$$

E.4.3 TREEQN

In this baseline, the starting step is encoding the input state to its latent counterpart with the Encoder module. Following this, a full-tree expansion, based on a predefined depth d, is performed using both Transition and Reward modules. The values at the leaf nodes are then backpropagated to the root node via the Bellman equation, as discussed in the Backup phase in Section 3.2. The root node Q-values serve as the final output, that is utilised for training. Given the exponential growth of TreeQN's computation graph with an increase in depth d, we choose a depth of 2 for our analysis, as recommended by TreeQN's original code base (Farquhar et al., 2018). The associated loss function is:

$$\mathcal{L}_{TreeQN} = \lambda_1 \mathcal{L}_Q + \lambda_2 \mathcal{L}_{\mathcal{D}} + \lambda_3 \mathcal{L}_{\mathcal{R}_\theta}$$

E.4.4 ONLINE SEARCH WITH SEPARATELY LEARNT WORLD MODEL

For this approach, we employ the best-first algorithm showcased in DTS. However, there's a divergence: the world model and the value module are trained independently, each focusing on their specific objectives. As outlined in (Ye et al., 2021), we incorporate self-supervised consistency losses defined in Equation 7 and 8 as they improve the online search, even in cases where the world model is not jointly trained with the online search. The Q-values used for training are computed directly using the value module without performing online search during training. The loss function tailored for this approach is:

$$\mathcal{L}_{Search} = \lambda_1 \mathcal{L}_Q + \lambda_2 \mathcal{L}_{\mathcal{D}} + \lambda_3 \mathcal{L}_{\mathcal{T}_\theta} + \lambda_4 \mathcal{L}_{\mathcal{R}_\theta}$$

E.5 ISSUE WITH BASELINE NORMALISED SCORE

Within the context of Atari games, the Baseline Normalised Score (BNS) is frequently utilised to evaluate the performance of agents. When human players are used as the baseline, it is often termed Human Normalised Score. The primary allure of BNS lies in its capacity to offer a relative assessment of an agent's performance, comparing it against a standard benchmark—this could be human players or even another agent.

One of the primary benefits of the BNS is its ability to provide a consistent metric across different games, addressing the difference in scale inherent in raw scores. By enabling the calculation of the average BNS across multiple games, we gain insight into the overall efficacy of an agent. This not only facilitates direct performance comparisons between diverse agents and methodologies but also paints a picture of how the agent's abilities stack up against human standards.

To derive the BNS, we start by logging the agent's raw score in an Atari game. This raw score is then normalised against a baseline score, derived from baseline agent's performance on the same game. By dividing the agent's score by the baseline's score (and sometimes subtracting the score of a random agent), we get a relative metric. Mathematically, this can be represented as:

$$BNS(\pi) = \frac{S_\pi - S_R}{S_B - S_R} \tag{21}$$

Here, S_π, S_B and S_R denote the raw scores of the agent, the baseline policy, and a random policy, respectively. Interpretation-wise, a BNS of 1 indicates parity with the baseline. Values exceeding 1 signify outperformance, while those below 1 indicate underperformance relative to the baseline.

Nevertheless, the BNS has its frailties. It inherently presumes the baseline policy will always surpass the performance of the random policy. But there can be instances, contingent on the environment or the specific baseline policy, where this isn't the case. In scenarios where the baseline policy underperforms the random policy, the BNS results in a negative denominator. This poses a predicament: even if our agent's policy performs better than the random policy, the BNS unfairly penalises it. In our experiments with Procgen, we observed that for 2 out of the 16 games, namely heist and maze, the baseline policy underperformed compared to the random policy. Given these pitfalls, our evaluations pivot towards a more robust metric: the Z-score. The Z-score, often termed as the "standard score," provides a statistical measurement that describes a value's relationship to the mean of a group of values. It is measured in terms of standard deviations from the mean. If a Z-score is 0, it indicates that the data point's score is identical to the mean score. Z-scores may be positive or negative, with a positive value indicating the score is above the mean and a negative score indicating it is below the mean.