Differentiable Dec-Options: Scalable Neural Network Decomposition

Anonymous Author(s)
Affiliation
Address
email

Abstract

Temporally extended actions, or options, provide a powerful abstraction for solving reinforcement learning (RL) problems, especially in transfer learning settings where knowledge reuse is needed. Recent work presented Dec-Options, an approach that uses simple insights from computer programming: program decomposition and reuse. This method treats neural networks as programs that can be decomposed into functions, similarly to how software engineers refactor reusable code from larger programs. Dec-Options evaluates all functions in a network so it can select reusable options. The issue is that the number of functions that a feedforward neural network encodes grows exponentially with the size of the network, so the approach is restricted to small networks. In this paper, we show how the search for subfunctions of a neural network can be done with gradient descent, which allows us to scale Dec-Options to larger networks. Moreover, we show how the extracted functions can be parameterized with learned default parameters that help with generalization across tasks. When using piecewise-linear activations (e.g., ReLU), the extracted options are compressed into networks smaller than the original network. This is similar to how functions extracted from a program have fewer lines of code than the original program. Empirical results on challenging gridworld problems demonstrate the effectiveness of our approach—namely, our neural-programmatic options improve sample efficiency in downstream tasks.

1 Introduction

In transfer learning settings, an agent benefits from reusing skills learned in previous tasks. Knowledge reuse with neural networks can be complex due to factors such as catastrophic forgetting [French, 1999] and loss of plasticity [Dohare *et al.*, 2024]. Some of these difficulties do not arise when using other types of representations, such as programming languages. For example, DreamCoder learns a library of lambda calculus programs by compressing solutions to tasks it has solved, so that the programs in the library can reduce the complexity of solving downstream tasks [Ellis *et al.*, 2023].

Alikhasi and Lelis [2024] take a programmatic view of neural networks to learn reusable skills through a method called Dec-Options. Dec-Options decomposes feedforward neural networks encoding a solution to a sequential decision-making problem into an equivalent structure called a neural tree. Then, it evaluates each subtree of the neural tree for reusable subfunctions that could be stored in a library of programs to help reduce the complexity of solving downstream problems. The extracted programs can be seen as temporally extended actions, or options [Sutton *et al.*, 1999].

The main drawback of Dec-Options is that it evaluates all possible subtrees of a neural tree, whose size grows exponentially with the number of neurons in the original neural network. As a result, Dec-Options can only be used with small neural networks, which prevents it from being used on more challenging problems (Dec-Options were evaluated with networks with only six neurons). In this paper, we overcome this problem by searching in the space of possible subtrees of the neural tree with gradient descent. Our method, Differentiable Dec-Options (DIDEC, which we pronounce "Dye-Deck"), leverages that each subtree of the neural tree can be recovered by using a mask *over the neurons* of the underlying network. That way, learning masks over neurons is equivalent to searching for subfunctions of the network. This allows DIDEC scale to larger networks—we use networks with more than 100 neurons in our experiments.

We also show that subfunctions needed to help the agent solve downstream tasks might only be recovered if we also learn "default input parameters". For example, consider the case where the agent learns a behavior that could be reused in different locations of the environment. However, the network encoding the behavior depends on features present only in the location where the agent learned the behavior. In this case, a subfunction will encode the behavior that generalizes to any location only if we learn default location parameters, which allow the agent to "pretend" to be where it was when it originally learned the behavior. DIDEC also learns default parameters with gradient descent by masking the input observation—it searches for a subfunction and its default parameters simultaneously.

We evaluate DIDEC in a transfer learning setting where we have neural models encoding solutions to previous tasks, and the agent can use these models to improve its sample efficiency while learning how to solve the current task. DIDEC decomposes the networks encoding the solution to previous problems to equip the agent with a library of options. We hypothesize that DIDEC can extract reusable options even from larger networks. We evaluate this hypothesis by checking whether DIDEC's options can generalize better to downstream tasks than baselines that use the solution to previous tasks without decomposing them. Empirical results in challenging gridworld problems support our hypotheses.

2 Problem Formulation

We are interested in solving partially observable Markov decision processes (POMDP), which are defined as (S, A, O, p, q, r, S_0). Here, S, A, O are the sets of states, actions, and observations, respectively. The function $p : S \times A \to S$ defines the transition dynamics of the environment by returning the next state s_{j+1} given the current state s_j and an action the agent takes at s_j; function $q : S \to O$ defines what the agent observes in the current state. Function $r : S \times A$ defines the reward value the agent observes after performing an action in a given state. Finally, S_0 defines the distribution of initial states. A policy $\pi : O \times A \to [0, 1]$ receives an observation o and an action a and returns the probability that a is taken in o. We consider the transfer learning setting where we learn to solve a sequence of POMDPs $P_1, P_2, \cdots, P_{i-1}$ and is evaluated in the i-th POMDP, P_i. This means that for each P_j with $j < i$, we have a policy π_j that approximates π_j^*, where

$$\pi_j^* = \arg\max_{\pi \in \Pi} \mathbb{E}_{s_0 \sim S_0} [R(s_0, \pi)] \,.$$

In an episodic setting with T time steps, $R(s_0, \pi) = \sum_{t=0}^{T} r(s_t, a_t)$, and Π is the class of possible policies. Given the collection of neural policies $\{\pi_j\}_{j=1}^{i-1}$, we want to approximate π_i^* while optimizing for sample efficiency. We approach this problem by decomposing the neural policies $\{\pi_j\}_{j=1}^{i-1}$ (Section 3) into options (Section 4) that can improve the agent's sample efficiency while solving P_i.

3 Decomposing Feedforward Neural Networks

We assume that the policies $\{\pi_j\}_{j=1}^{i-1}$ are encoded in fully connected feedforward neural networks, such as the one shown in the lower left corner of Figure 1. Each layer k has n_k neurons $(1, \cdots, n_k)$, with $n_1 = |X|$, where X is the observation vector passed as input to the network. The network's trainable parameters are the values between any subsequent layers k and $k + 1$. We denote such weights as $W^k \in \mathbb{R}^{n_{k+1} \times n_k}$ and $B^k \in \mathbb{R}^{n_{k+1} \times 1}$. The z-th row vectors of W^k and B^k, denoted W_z^k and B_z^k, represent the weights and the bias term of the z-th neuron of the $(k + 1)$-th layer.

We denote the vector with the values produced in the k-th layer of the network as $A^k \in \mathbb{R}^{n_k \times 1}$. Here, $A^1 = X$ and A^m is the model output of a network with m layers. We compute $A^i = g(Z^i)$, where $g(\cdot)$ is an activation function, and $Z^i = W^{i-1} \cdot A^{i-1} + B^{i-1}$. Initially, we consider piecewise-linear activation functions, such as ReLU, where $g(z) = \max(0, z)$, later we consider other functions.

Alikhasi and Lelis [2024] showed how a mapping between a neural network that uses piecewise-linear activation functions into an equivalent neural tree allows us to extract subfunctions from the network. We explain this mapping with the example in Figure 1. Each neuron is mapped to a level in the tree. In Figure 1, neuron A^2_1 is represented by the root level of the tree, A^2_2 by the second level of the tree, and A^3_1 by the tree's leaves. Each node considers the two possible outcomes of a ReLU neuron: if $Z \leq 0$, the output is 0.0 (left branch); if $Z > 0$, the output is Z (right branch). In our example, the output of the leaf nodes in the left subtree of the root is calculated for $A^2_1 = 0$, while the output of the leaf nodes in the right subtree of the root is calculated for $A^2_1 = Z^2_1$. For example, when following the left branch twice from the root, we have $A^2_1 = A^2_2 = 0$, so $A^3_1 = \sigma(5)$, which is the Sigmoid function of the bias term of the output neuron. If we follow the right and then the left branches from the root, then $A^2_1 = -x_1 + 4x_3$ and $A^2_2 = 0$, so the leaf output is $\sigma(-1(-x_1 + 4x_3) + 5) = \sigma(x_1 - 4x_3 + 5)$.

Each subtree of the neural tree provides a subfunction of the policy. Since the number of subtrees grows exponentially with the number of neurons in the hidden layer, Alikhasi and Lelis [2024] considered networks with only six neurons so that all subtrees could be evaluated in terms of their utility as options for downstream problems. Next, we explain Alikhasi and Lelis's Dec-Options.

4 Dec-Options

Dec-Options extracts subfunctions of the policies in $\{\pi_j\}_{j=1}^{i-1}$ so that they can be used as options [Sutton *et al.*, 1999]. An option ω is a tuple $(I_\omega, \pi_\omega, T_\omega)$, where I_ω is the set of observations in which the option can be invoked, π_ω is the policy that the agent follows once the option starts, T_ω is a function that receives an observation o_t and returns the probability that the option terminates in o_t. We consider the call-and-return execution of options, where the agent follows π_ω until ω terminates.

Dec-Options operates in two steps, given the policies $\{\pi_j\}_{j=1}^{i-1}$, which we will refer to as Π_{train}. First, it decomposes each policy π_j in Π_{train} into all possible subtrees of the neural tree of π_j (Section 4.1). Then, it selects a subset of these subtrees to be used as options for solving P_i (Section 4.2).

4.1 Subtrees to Options

Options are temporal abstractions because they encode policies that execute over multiple steps before terminating. Feedforward neural networks do not represent programs with loops, which can be run many iterations; instead, they represent chains of if-then-else structures. Dec-Options incorporates the temporal aspect of these abstractions by wrapping each subtree, extracted from a policy, in loops.

Let $\mathcal{T}_{\text{train}} = \{\mathcal{T}_j\}_{j=1}^{i-1}$ be the set of trajectories obtained by rolling each π_j in Π_{train} out from an initial state of each POMDP in $\{M_j\}_{j=1}^{i-1}$. Each trajectory $\mathcal{T}_j$ is a sequence of observation-action pairs of the form $\{(o_0, a_0), (o_1, a_1), \cdots, (o_{T_j}, a_{T_j})\}$. Let $T_{\max} = \max_j T_j$ be the length of the longest trajectory in Π_{train}, and U_j be all subtrees that can be extracted from π_j. Dec-Options wraps each u in U_j in programs `repeat(t):u`, where subtree u is invoked t times before termination. Dec-Options considers one such program for each t in $\{2, \cdots, T_{\max}\}$. The programs obtained from the subtrees in U_j form the set of options Ω_j extracted from π_j. For Dec-Options, $I_\omega = O$, that is, the options can be invoked from any observation, and T_ω is deterministic as options are executed for t steps.

4.2 Dec-Options Subset Selection

We denote the set of options extracted from Π_{train} as $\Omega = \cup_{j=1}^{i-1}\Omega_j$. Given the size of Ω, attempting to use all options to solve downstream problems would slow down learning, as the agent would have to learn to use a very large number of options. That is why Dec-Options selects a subset Ω' of Ω based on the Levin loss [Orseau *et al.*, 2018] of Ω'. The Levin loss approximates the usefulness of Ω' in solving downstream tasks. Intuitively, the Levin loss evaluates whether the likelihood of an agent solving a problem P_j would increase if we augmented the action space of the agent with Ω'.

$$\mathcal{L}(\mathcal{T}_j, \pi) = \frac{|\mathcal{T}_j|}{\prod_{(s,o) \in \mathcal{T}_j} \pi(s, o)} .$$

The Levin loss $\mathcal{L}(\mathcal{T}_j, \pi)$ computes the expected number of samples an agent following π requires to recover the trajectory $\mathcal{T}_j$. The denominator of $\mathcal{L}$ gives the expected number of rollouts the agent following π needs to perform to observe the trajectory $\mathcal{T}_j$; the numerator is the required number of agent interactions with the environment in each rollout. Dec-Options selects a subset Ω' of Ω such that the resulting policy π minimizes the Levin loss on trajectories $\mathcal{T}_{\text{train}}$.

To simulate the scenario in which the agent is starting to learn how to solve a problem, the policy π used in the computation of the Levin loss is the uniform policy—a randomly initialized neural network can produce a probability distribution over actions that is close to uniform. The uniform policy on the agent's action space augmented with options Ω' is denoted $\pi_u^{\Omega'}$. In this way, the larger the set Ω', the smaller $\pi(s, o)$, which increases the Levin loss. Conversely, if the options in Ω' cover sub-trajectories of the trajectories in $\mathcal{T}_{\text{train}}$, then the number of decisions the agent must make to reproduce the trajectories is reduced, which decreases the Levin loss. The loss balances the negative (increase $\pi(s, o)$) and positive (decrease the number of decisions) effects of adding options to Ω'.

Dec-Options approximates a solution to the problem of selecting a subset Ω' of Ω that minimizes the sum of the Levin loss of the trajectories in $\mathcal{T}_{\text{train}}$. Importantly, when computing the value of $\mathcal{L}(\mathcal{T}_j, \pi_u^{\Omega'})$, Dec-Options does not consider the options ω in Ω' that were extracted from π_j. This is to prevent the selection of trivial policies that do not generalize to downstream problems. For example, the policy that reduces the Levin loss the most for the trajectory $\mathcal{T}_j$ is π_j, which is unlikely to generalize because it is too specialized in π_j. Alikhasi and Lelis [2024] used a greedy algorithm to approximate a solution to the subset selection problem. They also presented a dynamic programming procedure to compute the Levin loss for a given subset Ω'. Please refer to Appendix A for details.

5 Differentiable Dec-Options (DIDEC)

Dec-Options has two important shortcomings. First, since the number of subtrees grows exponentially with the number of neurons in the network, Dec-Options can only be used with small networks (Alikhasi and Lelis [2024] used networks with only six neurons), limiting the approach's applicability. Second, as we show in Section 5.2.1, the functions Dec-Options extracts from neural networks might not generalize to downstream problems because the extracted functions do not come with "default parameters". Differentiable Dec-Options (DIDEC) overcomes these two shortcomings.

5.1 Subtree Extraction as Masking Neurons

Alikhasi and Lelis [2024] showed that the number of possible subtrees a neural network with d hidden neurons is $\sum_{i=0}^{d} \binom{d}{i} \cdot 2^i$. For the neural tree in Figure 1, we have $1 + 4 + 4 = 9$ subtrees. The 1 represents the entire neural tree, the first 4 represents the subtrees of the root: there are 2 subtrees when A_1^2 is the root, and another 2 when A_2^2 is the root. Finally, the last 4 is the number of leaf nodes.

We note the binomial identity $(1+x)^d = \sum_{i=0}^{d} \binom{d}{i} \cdot x^i$, so if $x = 2$, then $\sum_{i=0}^{d} \binom{d}{i} \cdot 2^i = (1+2)^d = 3^d$. This identity is useful because it suggests a gradient-based solution to the problem of finding helpful neural subfunctions. In a subtree of the neural tree, each ReLU neuron can be **active**, when it returns z, **inactive**, when it returns 0, or **part of the program**, when the neuron's function is accounted for in the subtree. To illustrate, consider the left subtree of the tree in Figure 1, neuron A_1^2 is inactive (we follow its left child), while A_2^2 is part of the program because its node is in the subtree. This means that extracting a subtree from the neural tree is equivalent to setting one of the following states for each neuron: active, inactive, or part of the extracted program, for a total of 3^d possibilities.

DIDEC learns masks for neurons to extract subtrees of the underlying tree. Masks are given by a matrix $\Theta^k \in \mathbb{R}^{n_k \times 3}$ with trainable weights $\big((\theta_{1,1}^k, \theta_{1,2}^k, \theta_{1,3}^k), (\theta_{2,1}^k, \theta_{2,2}^k, \theta_{2,3}^k), \ldots, (\theta_{n_k,1}^k, \theta_{n_k,2}^k, \theta_{n_k,3}^k)\big)$ for the n_k neurons in the k-th layer of the network. The parameters $(\theta_{j,1}^k, \theta_{j,2}^k, \theta_{j,3}^k)$ are used in a Softmax operation to determine the state of the j-th neuron in the k-th layer, as shown in Algorithm 1.

In line 1 of Algorithm 1, we compute the logits Z^k. In line 2 we compute the mask of the neurons. This is done by computing the Softmax function for each row vector of Θ^k and then discretizing the values by generating one-hot vectors as rows of the mask matrix M^k. The j-th element of the i-th row will be 1 and the other columns 0 if the j-th element is the largest. Finally, in line 3 we compute the masked output A^k of the n_k neurons. The rows whose first element is 1 contribute with 0 in the sum (inactive neurons); the rows whose second element is 1 contribute with Z^k (active neurons); the rows whose third element is 1 contribute with $\text{ReLU}(Z^k)$ (neurons that are part of the program).

Algorithm 1 MASKED FORWARD PASS OF THE k-TH LAYER

1: $Z^k = W^{k-1} \cdot A^{k-1} + B^{k-1}$

2: $M_{ij}^k = D\left(\frac{\exp(\theta_{ij}^k)}{\sum_{j'=1}^{3}\exp(\theta_{ij'}^k)}\right)$ where

$$D(x) = \begin{cases} 1 & \text{if } x = \max\left(\frac{\exp(\theta_{i1}^k)}{\sum_{j'=1}^{3}\exp(\theta_{ij'}^k)}, \frac{\exp(\theta_{i2}^k)}{\sum_{j'=1}^{3}\exp(\theta_{ij'}^k)}, \frac{\exp(\theta_{i3}^k)}{\sum_{j'=1}^{3}\exp(\theta_{ij'}^k)}\right) \\ 0 & \text{otherwise} \end{cases}$$

3: $A^k = M_{:,1}^k \times 0 + M_{:,2}^k \times Z^k + M_{:,3}^k \times \text{ReLU}(Z^k)$

The function $D(x)$ in line 2 is non-differentiable due to the $\max$ operation. When updating Θ^k with gradient descent, we use the straight-through estimator [Bengio *et al.*, 2013], which passes the gradient computed up to the discretization step in the backward pass directly to the Softmax layer. We also considered the modified $\tanh$ function of Pitis [2017] and Koul *et al.* [2019] to discretize over 3 values, but preliminary experiments favored the use of the Softmax approach of Algorithm 1.

5.1.1 Learning Masks as Neural Subtree Selection

Ideally, we would learn the parameters Θ^k such that we minimize the Levin loss of a subset of options Ω'. However, the Levin loss is computed with a dynamic programming process, as shown in Appendix A, which is not differentiable. Similarly to Dec-Options, DIDEC uses the same dynamic programming procedure to calculate the Levin loss, and it also treats the subset selection problem as a discrete optimization problem. In contrast to Dec-Options that considers all 3^d subtrees of a neural tree, DIDEC considers a potentially much smaller number by learning masks with gradient descent.

For each $\mathcal{T}_j$ of $\mathcal{T}_{\text{train}}$, we consider all subsequences τ of observation-action pairs of $\mathcal{T}_j$ with length z in $\{2, 3, \cdots, z_{\max}\}$. Here, $z_{\max} \leq T_j$ is a hyperparameter. Each subsequence is used to train masks for neurons in the π_j network. We use the cross-entropy loss to learn Θ such that the masked network predicts the actions in the observation-action pairs of τ. Similarly to Dec-Options, to allow for generalization, DIDEC learns parameters Θ for policies π_i and a subsequence τ of the trajectory $\mathcal{T}_j$ only if $i \neq j$. This forces DIDEC to extract subfunctions of π_i that help solve a problem P_j for which π_i was not trained to solve. We hypothesize DIDEC's gradient-based process for selecting subtrees of the policy allows for the discovery of options that generalize to downstream problems.

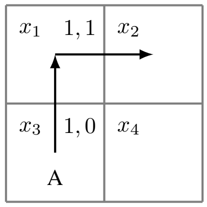
For a subsequence τ with length b of $\mathcal{T}_j$, we train the parameters Θ to generate an option of the form `repeat(b)`:π_j^Θ, where π_j^Θ is the policy π_j masked with Θ, as shown in Algorithm 1. Larger $z_{\max}$ values will generate more options. Also, options trained with larger $z_{\max}$-values tend to be specific to the behavior $\mathcal{T}_j$, thus less likely to generalize to downstream problems. Choosing smaller values of $z_{\max}$ reduces the system's overall computational complexity while automatically eliminating options less likely to generalize. DIDEC's set Ω is formed by one option for each subsequence τ. We describe the process in which DIDEC selects a subset Ω' of Ω in Appendix B.

Dec-Options considers all 3^d subtrees of a neural tree as options. Each of these subtrees is wrapped around programs with repeat-loops, yielding a total of $3^d \times \sum_{i=2}^{T_{\max}} (T_{\max} - i + 1)$ which is $O(3^d \cdot T_{\max}^2)$, where $T_{\max}$ is the length of the longest sequence in $\mathcal{T}_{\text{train}}$. This contrasts with DIDEC, which considers only $\sum_{i=2}^{z_{\max}} (T_{\max} - i + 1)$ options, with complexity $O(T_{\max} \cdot z_{\max})$. DIDEC's complexity does not include the term 3^d because it uses gradient descent to find the subtrees more likely to yield helpful options, rather than evaluating all 3^d possibilities as Dec-Options does. This means we can use DIDEC with larger networks, since the number of options evaluated no longer depends on d.

5.2 Learning Default Parameters

While subtrees of a neural tree might encode helpful subfunctions that can be reused in downstream tasks, in this section we argue that neural decomposition can be more effective if we learn "default parameters" to the extracted functions. Consider the motivating example in the next section.

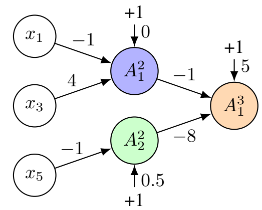
(a) ComboGrid



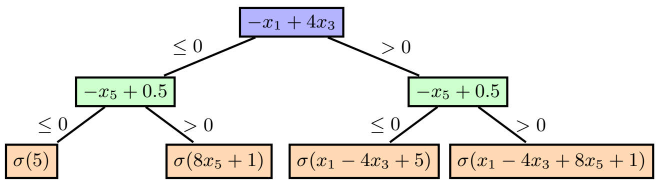
(b) Observation-Action Pairs

	x_1	x_2	x_3	x_4	x_5	a
o_1	0	0	1	0	1	1
o_2	0	0	1	0	0	0
o_3	1	0	0	0	1	1
o_4	1	0	0	0	0	1

(c) Neural Policy



(d) Neural Tree



(e) Compressed Policy

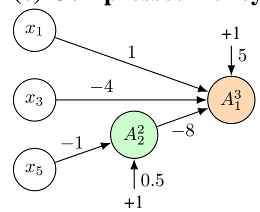


Figure 1: **(a)** Instance of a ComboGrid environment with combos of length two. The sequence of actions $1, 0$ has the effect of moving up, while the sequence $1, 1$ has the effect of moving right. The agent 'A' starts at x_3 and finishes at x_2 after applying the sequence of actions $1, 0, 1, 1$. **(b)** Table with observation-action pairs for the trajectory depicted in (a). In addition to the agent location (x_1, x_2, x_3, and x_4), the observation includes the bit x_5, which is set to 1 if no action in a sequence was taken, and 0 otherwise. **(c)** Neural network that fits the data from the observation-action pairs in (b). The hidden neurons use ReLU activation functions, while the output neuron uses a sigmoid function. The connections with zero weight between the input and the hidden neurons are omitted for clarity. **(d)** Neural tree equivalent to the network from (c). The left subtree of the neural tree represents the subfunction "right", because it returns the combo $1, 1$ independently of the agent's location. The right subtree with the default parameter $x_3 = 1$ represents the subfunction "up". **(e)** The compressed version of the original model when neuron A_1^2 is set to active, and is removed from the network.

5.2.1 Motivating Example

Consider the ComboGrid problem shown in the upper left corner of Figure 1. The cell in the 2×2 grid are denoted as x_1, x_2, x_3, and x_4; the agent 'A' starts in x_3. In ComboGrid, the agent needs to perform a sequence of actions until the effect of moving to a different cell is observed. The agent can perform two actions in a given time step: 0 or 1. After performing action 1 and then 0, the agent moves to x_1; the "combo" $1, 0$ produces the effect of moving up. Similarly, the sequence $1, 1$ produces the effect of moving right. The sequence $1, 0, 1, 1$ moves the agent from x_3 to x_2.

Observations are given by a one-hot encoding of the position of the agent on the grid and an extra bit, x_5, which indicates whether the number of actions the agent has taken so far is even ($x_5 = 1$) or odd ($x_5 = 0$). This way, if $x_5 = 1$, then the agent has not started a combo sequence. The observation of the agent shown in the grid is given by $o1$ in the table of observation-action pairs. Observations $o2$ and o_3 are obtained by applying actions $1, 0$ from o_1. Finally, once the agent performs action 1 in o_4, it moves to cell x_2, whose observation is not shown in the table of observation-action pairs.

The neural network shown in the lower left corner of Figure 1 produces the sequence of actions given in the table of observation-action pairs. This network uses ReLU actions in the hidden layer and a sigmoid function in the output layer. The neural tree shown in the lower right corner of the figure is equivalent to the neural network. The left subtree of the neural tree encodes the "right combo". Its root checks whether $-x_5 + 0.5 \leq 0$. Since before starting the sequence, $x_5 = 1$, we follow the left child, leading to $\sigma(5) = 0.99$ (action 1). After action 1 is performed, $x_5 = 0$, so we follow the right child to the function $\sigma(8x_5 + 1) = \sigma(1) = 0.73$ (action 1). Since this subtree depends only on x_5, it represents a function that performs the "up combo" independently of the location of the agent.

Consider now the right subtree of the neural tree. Before the agent starts performing a sequence, $x_5 = 1$, so we follow the left child, leading to $\sigma(x_1 - 4x_3 + 5)$. Note that $x_1 - 4x_3 + 5 \geq 1$ for any combination of x_1 and x_3, so $\sigma(x_1 - 4x_3 + 5) \geq 0.73$, thus always producing action

1 when $x_5 = 1$. After performing action 1, $x_5 = 0$ so we follow the right child, leading to $\sigma(x_1 - 4x_3 + 8x_5 + 1) = \sigma(x_1 - 4x_3 + 1)$. In this case, the output value the model produces depends on both x_1 and x_3 and that if $x_3 = 1$, then $x_1 - 4x_3 + 1 \leq -2$ and $\sigma(x_1 - 4x_3 + 1) \leq 0.11$. Thus, this subtree always produces action 0 when $x_5 = 0$ and $x_3 = 1$. By extracting the function represented by the right subtree of the neural tree and setting the default value of x_3 to 1, we have a function that performs the "up combo" independently of the agent's position.

5.2.2 Learning Input Masks

In addition to learning masks for neurons, DIDEC also learns masks for input values, to allow default parameters, as discussed in the example in Section 5.2.1. We consider learning masks for input values for problem domains with a discrete and finite number of input values. For binary inputs, we learn masks with the same Softmax approach described in Section 5.1. The input masks defines one of the three possibilities for a given input feature: **always 1**, **always 0**, or **read value from environment**. For the example from Section 5.2.1, a generalizable "up combo" can be obtained by setting the mask of x_3 to "always 1", while all other input values could be "read value from the environment".

Note that masking input values might be sufficient to learn subfunctions that generalize. For example, learning that x_3 is always 1 in the problem of Figure 1 is equivalent to extracting the right subtree with the default parameter $x_3 = 1$, thus making the masking of neurons unnecessary. Our experiments evaluate DIDEC with three masking schemas: input-only, neurons-only, and input-and-neurons.

Independent of the benefit of masking neurons in discovering options, masking neurons can be a valuable compression scheme. In the example from Figure 1 (e), once we mask A_1^2 to be active, we can reduce the size of the network by making A_1^3 a function of x and A_2^2: $A_1^3 = \sigma(x_1 - 4x_3 - 8A_2^2 + 5)$ and removing A_1^2 from the model. In general, every neuron at layer k that is masked to be active or inactive can be removed by rewriting the function of the neurons at layers $k + 1$ accordingly. This compression is similar to how a software engineer extracts functions from a codebase: both the extracted functions and the masked models have fewer lines than the original implementation.

The approach of masking only the inputs has the advantage of being applicable to neural networks that use activation functions other than piecewise-linear functions and to other neural architectures. We evaluate DIDEC with input-only masking to extract options from $\tanh$ recurrent networks.

6 Related Work

Options Early work on options relied on manual design [Sutton *et al.*, 1999], but many methods now learn options automatically—though they often require human choices such as the number of options [Bacon *et al.*, 2017; Igl *et al.*, 2020] or their duration [Frans *et al.*, 2017; Tessler *et al.*, 2017]. Dec-Options avoids such supervision but depends on data from previously solved tasks. Our setting matches that of Dec-Options, though we introduce a limit $s_{\max}$ on the number of options. This constraint reduces DIDEC's computational cost and regularizes the learned options, preventing overspecialization to $\mathcal{T}_{\text{train}}$. While options have also been explored for improving exploration [Jinnai *et al.*, 2020; Machado *et al.*, 2023], both Dec-Options and DIDEC focus on compressing reusable behaviors to facilitate downstream transfer [Konidaris and Barto, 2007].

Transfer Learning Knowledge transfer across tasks has been studied via regularization [Kirkpatrick *et al.*, 2017], architectural priors [Rusu *et al.*, 2016; Yoon *et al.*, 2017; Schwarz *et al.*, 2018], and experience replay [Rolnick *et al.*, 2019]. A common strategy is to reuse parts of pretrained models [Clegg *et al.*, 2017; Shao *et al.*, 2018]. Unlike these approaches, DIDEC transfers knowledge by extracting reusable programs—options—through gradient-based network decomposition.

Library Learning DIDEC belongs to a family of library-learning methods [Cao *et al.*, 2023; Bowers *et al.*, 2023; Rahman *et al.*, 2024; Palmarini *et al.*, 2024]. For instance, DreamCoder [Ellis *et al.*, 2023] builds a library of reusable lambda calculus functions to solve program synthesis problems more efficiently. Similarly, DIDEC constructs a library of reusable neural programs from solved tasks. However, prior work typically assumes symbolic representations and supervised settings, whereas DIDEC operates in reinforcement learning using neural function approximators. It contributes toward bridging symbolic and neural paradigms in library learning.

Masking Networks Masking has been used in transfer learning, e.g., SupSup [Wortsman *et al.*, 2020] and Modulating Masks [Ben-Iwhiwhu *et al.*, 2022], but these approaches mask weights, not activations. In contrast, DIDEC masks neurons with piecewise-linear activations, enabling subfunction extraction. Input masking has also been explored—for mitigating visual distractions [Bertoin *et al.*, 2022; Grooten *et al.*, 2024] or learning auxiliary tasks [Yu *et al.*, 2022]—but DIDEC uses input masking to define default parameters that allow subfunctions to generalize across tasks.

7 Experimental Results

In our experiments, we evaluate our hypothesis that DIDEC's masking scheme can extract options that generalize to downstream tasks, even when using networks with substantially larger than those considered by Dec-Options (six neurons). We begin by describing our experimental setup, then introduce the benchmark domains, and finally present results on option extraction and comparison to several baselines.

7.1 Experimental Setup

We focus on feed-forward policies with a single hidden layer of 64 ReLU units. Such networks induce an immense option space—on the order of 3.43×10^{30} possible subtrees—far exceeding the capacity of prior Dec-Options [Alikhasi and Lelis, 2024]. For each policy we learn three masking variants: *Input-Only*, *Neuron-Only*, and *Input-and-Neuron*.

To demonstrate that our approach is agnostic to the choice of learning algorithm, we used two different policy-gradient methods: Advantage Actor-Critic (A2C)[**?**] and Proximal Policy Optimization (PPO)[Schulman *et al.*, 2017]. All architectural and implementation details are provided in Appendix D.

7.2 Benchmark Domains

We evaluated on the same two benchmark suites originally used to assess Dec-Options: ComboGrid [Alikhasi and Lelis, 2024] and MiniGrid [Chevalier-Boisvert *et al.*, 2023]. In each suite, we designate a set of training environments $P_j{}_{j=1}^{i-1}$ from which options are extracted, and one or more held-out test environments P_i on which we measure the usefullness of the learned options.

MiniGrid. For training, we used three variants of the `SimpleCrossing` task on a 9×9 grid, where an agent must navigate around a central barrier to reach a fixed goal location. For testing, we adopted three configurations of the `FourRoom` environment on a 19×19 grid, each differing in agent and goal placement to impose increasing difficulty. In Level 1, the agent and goal share the same room. In Levels 2 and 3, the agent must cross one and two doorways, respectively, to reach the goal. The agent's observation consist of an egocentric view of size 5×5 or 9×9 around the agent in addition to the agent's orientation.

ComboGrid. Training consists of four simple 5×5 grids, in which the agent and goal occupy opposite corners. The test task is also a 5×5 grid, with multiple goals at the midpoints of the outer walls. The agent's observation includes its position, the goal's position, and the two most recent actions. We also replicate this setup on 6×6 grids to evaluate generalization to a larger environment.

7.3 Option Extraction and Baseline Comparison

For each masking variant (input-only, neuron-only, input-and-neuron) we learn masks over sub-trajectories of length 2 to 24 steps, using PPO in MiniGrid and A2C in ComboGrid. We compare the performance of DIDEC's extracted options against four baselines:

1. **Vanilla**: No options, only primitive actions.
2. **Transfer**: Directly applying the optimal policy from the training environments as the options for the test environment.
3. **DecWhole:** Apply the hill-climbing subset-selection procedure (Appendix B) to all sub-trajectory fragments (lengths 2–24) of the optimal policies, choosing the subset that minimizes Levin Loss.

4. **FineTune**: Fine-tune each training policy (instead of masking), then perform hill-climbing subset selection as in DecWhole.

Figure 2 compares the performance of DIDECagainst all baselines on the MiniGrid domain. Across every difficulty level and for both egocentric view sizes, DIDEC and the FineTune baseline consistently rank as the top two methods.

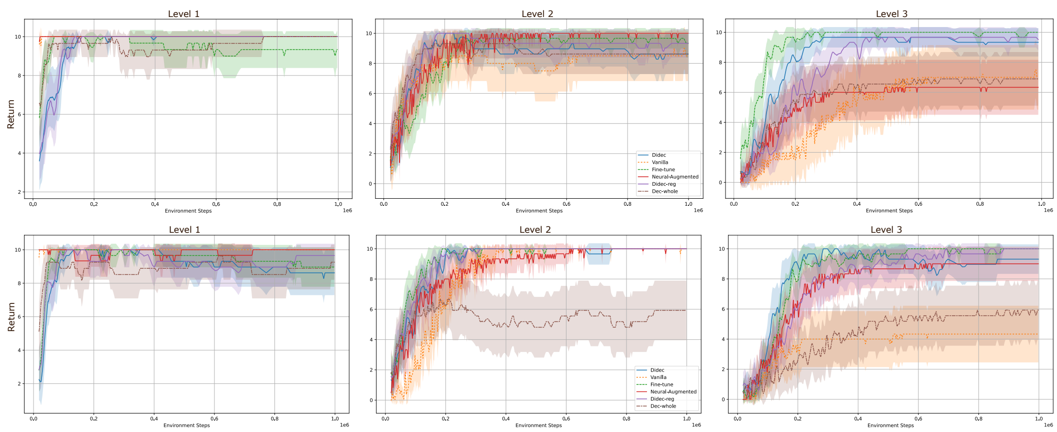


Figure 2: Average return on MiniGrid with egocentric view sizes of 5 (top) and 9 (bottom) across three difficulty levels (30 independent runs).

Figure 3 shows results on ComboGrid, where DIDEC and the DecWhole baseline achieve the highest average returns.

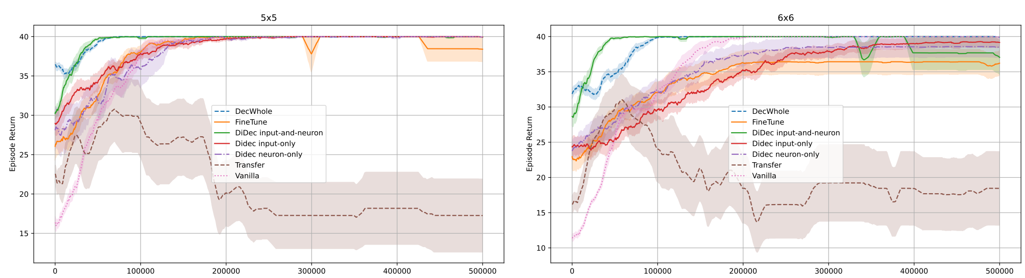


Figure 3: Average return on ComboGrid (15 independent runs).

Overall, DIDEC exhibited more stability than the other baselines and remains among the top performers across both domains.

8 Conclusion

In this work, we introduced Differentiable Dec-Options (DIDEC), a novel, gradient-based framework for extracting temporally extended actions (options) from neural policies at scale. By casting subtree selection as a differentiable masking problem over neurons and inputs, DIDEC overcomes the exponential blow-up of prior work [Alikhasi and Lelis, 2024], enabling option discovery in networks with tens or even hundreds of neurons and thereby highlighting its scalability.

Our empirical evaluation on challenging gridworld benchmarks—ComboGrid and MiniGrid—demonstrates that DIDEC's extracted options consistently match or outperform the strongest baselines across both domains, underlining its stability and robustness.

References

Mahdi Alikhasi and Levi Lelis. Unveiling options with neural network decomposition. In *The Twelfth International Conference on Learning Representations*, 2024.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Eseoghene Ben-Iwhiwhu, Saptarshi Nath, Praveen K Pilly, Soheil Kolouri, and Andrea Soltoggio. Lifelong reinforcement learning with modulating masks. *arXiv preprint arXiv:2212.11110*, 2022.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

David Bertoin, Adil Zouitine, Mehdi Zouitine, and Emmanuel Rachelson. Look where you look! saliency-guided q-networks for generalization in visual reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 2023.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 2023.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Alexander Clegg, Wenhao Yu, Zackory Erickson, Jie Tan, C Karen Liu, and Greg Turk. Learning to navigate cloth using haptics. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2799–2805. IEEE, 2017.

Shibhansh Dohare, J. Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A. Rupam Mahmood, and Richard S. Sutton. Loss of plasticity in deep continual learning. *Nature*, 632(8026):768–774, 2024.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sabl-Meyer, Luc Cary, Lore Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua Tenenbaum. Dreamcoder: growing generalizable, interpretable knowledge with wake?sleep bayesian program learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 381, 06 2023.

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

Robert M. French. Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135, 1999.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

Bram Grooten, Tristan Tomilin, Gautham Vasan, Matthew E. Taylor, A. Rupam Mahmood, Meng Fang, Mykola Pechenizkiy, and Decebal Constantin Mocanu. Madi: Learning to mask distractions for generalization in visual deep reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, page 733?742. International Foundation for Autonomous Agents and Multiagent Systems, 2024.

Maximilian Igl, Andrew Gambardella, Jinke He, Nantas Nardelli, N Siddharth, Wendelin Böhmer, and Shimon Whiteson. Multitask soft option learning. In *Conference on Uncertainty in Artificial Intelligence*, pages 969–978. PMLR, 2020.

Yuu Jinnai, Jee W. Park, Marlos C. Machado, and George Konidaris. Exploration in reinforcement learning with deep covering options. In *International Conference on Learning Representations*, 2020.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

George Konidaris and Andrew Barto. Building portable options: Skill transfer in reinforcement learning. In *International Joint Conference on Artificial Intelligence*, pages 895–900, 2007.

Anurag Koul, Alan Fern, and Sam Greydanus. Learning finite state representations of recurrent policy networks. In *International Conference on Learning Representations*, 2019.

Marlos C. Machado, André Barreto, Doina Precup, and Michael Bowling. Temporal abstraction in reinforcement learning with the successor representation. *Journal of Machine Learning Research*, 24:1–69, 2023.

Laurent Orseau, Levi H. S. Lelis, Tor Lattimore, and Théophane Weber. Single-agent policy tree search with guarantees. In *Proceedings of the International Conference on Neural Information Processing Systems*, page 3205–3215, 2018.

Alessandro B. Palmarini, Christopher G. Lucas, and N. Siddharth. Bayesian program learning by decompiling amortized knowledge. In *Proceedings of the International Conference on Machine Learning*, 2024.

Silviu Pitis. Beyond binary: Ternary and one-hot neurons. `https://r2rt.com/beyond-binary-ternary-and-one-hot-neurons.html`, 2017. Accessed: 2025-05-12.

Habibur Rahman, Thirupathi Reddy Emireddy, Kenneth Tjhia, Elham Parhizkar, and Levi Lelis. Synthesizing libraries of programs with auxiliary functions. *Transactions on Machine Learning Research*, 2024.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. In *International conference on machine learning*, pages 4528–4537. PMLR, 2018.

Kun Shao, Yuanheng Zhu, and Dongbin Zhao. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 3(1):73–84, 2018.

Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Mitchell Wortsman, Vivek Ramanujan, Rosanne Liu, Aniruddha Kembhavi, Mohammad Rastegari, Jason Yosinski, and Ali Farhadi. Supermasks in superposition. *Advances in Neural Information Processing Systems*, 33:15173–15184, 2020.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

Tao Yu, Zhizheng Zhang, Cuiling Lan, Yan Lu, and Zhibo Chen. Mask-based latent reconstruction for reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

A Greedy Algorithm and Levin Loss Computation

In this section, we explain the greedy algorithm Alikhasi and Lelis [2024] used to select subsets. In Dec-Options' greedy algorithm, we start with an empty Ω' and select the option ω from Ω that minimizes the sum of the Levin loss for the trajectories in Π_{train} the most. Then, it makes $\Omega' = \{\omega\}$. In the next iteration, the procedure chooses another ω from Ω such that $\Omega' \cup \{\omega\}$ minimizes the sum of the Levin losses the most. This process continues until adding another option to Ω' would increase the Levin loss. The algorithm then stops and returns Ω' as its selected subset.

A key step in the greedy algorithm for subset selection is the computation of the Levin loss for a given subset Ω' and a trajectory $\mathcal{T}$. The algorithm 2 shows a dynamic programming approach to compute the loss. Such a procedure is necessary because the computation of the Levin loss depends on which options are used in each trajectory step. For example, if $\Omega' = \{\omega_1, \omega_2\}$ and both options can be applied in time step 1 of the trajectory, then for which should the loss be computed? Algorithm 2 shows an efficient procedure for considering all possibilities of use of options.

Algorithm 2 COMPUTE-LOSS

Require: Sequence $\mathcal{S} = \{o_0, o_1, \cdots, o_{T+1}\}$ of states of a trajectory $\mathcal{T}$, probability $p_{u,\Omega}$, options Ω
Ensure: $\mathcal{L}(\mathcal{T}, \pi_u^{\Omega})$

```
▷ Initialize table P as if no options were available: to reach the j-th state we need j actions
M[j] ← j for j = 0, 1, ⋯ , T + 1
for j = 0 to T + 1 do
    if j > 0 then
        M[j] ← min(M[j − 1] + 1, M[j])
    for ω in Ω do
        if ω is applicable in o_j then
            ▷ Option ω is used in o_j for ω_z steps
            M[j + ω_z] ← min(M[j + ω_z], M[j] + 1)
▷ M[T + 1] stores the smallest number of actions to reach the end of the sequence. The value of
  p_{u,Ω} is the probability of taking an action in the option-augmented action space according to the
  uniform policy. The function returns the minimum Levin loss for 𝒯 and Ω.
return |𝒯| · (p_{u,Ω})^{−M[T+1]}
```

B Stochastic Hill Climbing for Subset Selection

Alikhasi and Lelis [2024] used a common approximation to the NP-hard subset selection problem [Garey and Johnson, 1979], which greedily and iteratively selects the option that minimizes the Levin loss the most (see Appendix A). Preliminary experiments favored a stochastic hill climbing (SHC) algorithm over the greedy approach for selecting a subset of options. We use SHC with both DIDEC and with baselines that require approximating a solution to the subset selection problem.

SHC is a hill-climbing approach with a stochastic neighborhood function. SHC starts with a randomly selected candidate solution c and greedily selects the best neighbor c' of c. If c' has a better Levin loss value than c, the search continues with c' as the new c. Otherwise, the search terminates and returns c. We use SHC with random restarts. Once a candidate is returned, we repeat it from another initial candidate. The SHC result is the candidate with the smallest loss across all restarts.

In DIDEC, a candidate solution is a subset Ω' of Ω. To reduce the search space, all candidates considered in the search satisfy $|\Omega'| \leq s_{\max}$, where $\leq s_{\max}$ is a hyperparameter that limits the maximum number of options that can be selected. The neighborhood function $\mathcal{N}$ is defined as follows. Given a candidate Ω', we sample v options Ω'' from $\Omega - \Omega'$. If $|\Omega'| < s_{\max}$, we generate v neighbors $\Omega' \cup \{\omega\}$, one for each ω in Ω''. We also generate v^2 neighbors, where each ω'' in Ω'' is used to replace an ω' in Ω'. Finally, we generate other $|\Omega'|$ neighbors where we remove each ω' from Ω', thus generating neighbors of size $|\Omega'| - 1$. The function $\mathcal{N}$ returns the union of all these neighbors.

We sample the v options from Ω'' according to a distribution that favors options complementary to the current candidate subset Ω'. Let $\mathcal{T}_{\text{train}}^{\Omega'}$ be the set of observation-action pairs from Π_{train} that are "not covered" by an option in Ω'. Formally, the j-th observation-action pair of a trajectory is not covered by an option in Ω' if, while computing the Levin loss of Ω' in Algorithm 2, line 5, the $\min$ operator returns $M[j-1]+1$. Given the set of observation-actions pairs $\mathcal{T}_{\text{train}}^{\Omega'}$, we define the value of each ω in Ω'' as the number of times ω can be initiated at a pair in $\mathcal{T}_{\text{train}}^{\Omega'}$. An option ω of the form `repeat(b)`:π_ω can be initiated at a pair (o_j, a_j) if $\pi_\omega(o_{j+i})$ returns a_{j+i} for i in $\{0, 1, \cdots, b-1\}$. We sample options from Ω'' proportionally to their value. This neighborhood function favors options that can be used more often in pairs that are not yet covered by the options in the current candidate.

C DIDEC - Overall Approach

Algorithm 3 Differentiable Dec-Options (DIDEC)

Require: Observation-action trajectories $\mathcal{T}_{\text{train}} = \{\mathcal{T}_j\}_{j=1}^{i-1}$, neural policies $\Pi_{\text{train}} = \{\pi_j\}_{j=1}^{i-1}$ that generated the trajectories in $\mathcal{T}_{\text{train}}$, maximum subset size $s_{\max}$, maximum length of an option $z_{\max}$, neighborhood function $\mathcal{N}_v$, number of epochs E, learning rate α, a number of restarts r.
Ensure: A set of at most $s_{\max}$ options of the form `repeat(b)`:ω_π.

▷ *Generating training sequences by sliding a window of size $z = 2, \cdots, z_{\max}$ over the observation-action trajectories the policies in Π_{train} generate.*
$\mathcal{D} \leftarrow \emptyset, \Omega \leftarrow \emptyset$
for $z = 2, 3, \ldots, z_{\max}$ **do**
  for each trajectory $\mathcal{T}_j$ in $\mathcal{T}_{\text{train}}$ **do**
    for $t = 0, 1, \ldots, T_j - z$ **do**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t), (o_{t+1}, a_{t+1}), \ldots, (o_{t+z-1}, a_{t+z-1})\}$
▷ *Training masks for selecting subtrees of the neural trees of policies in Π_{train}. The masks can be input-only, neurons-only, or input-and-neurons. Each masked policy π^Θ results in an option.*
for each subsequence τ in $\mathcal{D}$ **do**
  Initialize Θ randomly
  for epoch $= 1, 2, \ldots, E$ **do**
    $\hat{a} = \pi^\Theta(\tau)$
    $\mathcal{L}(\Theta) = -\tau_a^\top \log(\hat{a})$
    $\nabla\Theta \leftarrow \frac{\partial \mathcal{L}(\Theta)}{\partial \Theta}$
    $\Theta \leftarrow \Theta - \alpha \cdot \nabla\Theta$
  $\Omega \leftarrow \Omega \cup \{$`repeat`$(|\tau|)\pi^\Theta\}$
▷ *Perform hill climbing to select a subset of Ω. The neighborhood function $\mathcal{N}_v(c, s_{\max})$ returns a set of neighbors after sampling v options from the options not in c; each neighbor has at most $s_{\max}$ options, as described in Appendix B. `Compute-Loss` is as described in Algorithm 2.*
$c^* \leftarrow \emptyset, l^* \leftarrow \infty$
for restart $= 1, 2, \ldots, r$ **do**
  $c \leftarrow$ Random subset of Ω with size at most $s_{\max}$
  $l \leftarrow$ `Compute-Loss`(c)
  $c_{\text{best}} \leftarrow c$, $l_{\text{best}} \leftarrow l$
  while True **do**
    $i \leftarrow$ False
    for each neighbor $c' \in \mathcal{N}_v(c, s_{\max})$ **do**
      $l' \leftarrow$ `Compute-Loss`(c')
      if $l' < l_{\text{best}}$ **then**
        $c_{\text{best}} \leftarrow c'$, $l_{\text{best}} \leftarrow l'$, $i \leftarrow$ True
    if not i **then**
      break
  if $l_{\text{best}} < l^*$ **then**
    $c^* \leftarrow c_{\text{best}}$, $l^* \leftarrow l_{\text{best}}$
return The set of options c^* represents

D Neural Architectures

We adopt a neural Actor-Critic framework based on the Advantage Actor-Critic (A2C) algorithm. The model is composed of two networks with shared input: an actor that parameterizes a stochastic policy, and a critic that estimates the value function.

Let d_{obs} denote the dimensionality of the observation vector (after flattening), and let $|\mathcal{A}|$ be the number of discrete actions.

Actor Network. The actor network maps an input observation to a distribution over actions via the following architecture:

- A linear layer with input size d_{obs} and output size 64,
- ReLU activation,
- A final linear layer with output size $|\mathcal{A}|$ producing unnormalized action logits.

The final layer is initialized with a reduced standard deviation ($\text{std} = 0.01$) to promote stability during early exploration.

Critic Network. The critic network has a deeper architecture to estimate the state value:

- A linear layer from d_{obs} to 64 units,
- ReLU activation,
- Another linear layer with 64 hidden units,
- ReLU activation,
- A final linear layer mapping to a scalar value.

The final layer of the critic is initialized with $\text{std} = 1.0$, which improves learning stability by producing meaningful value estimates early in training.

Weight Initialization. Weights are initialized orthogonally with a gain factor (default $\sqrt{2}$) to preserve activation variance, and biases are set to a constant (default 0.0), following common RL practice for stable and efficient training.