Learning Operators with NEAT for Boolean Composition in Reinforcement Learning

---

Amir Esterhuysen

Supervisors:
Prof. Benjamin Rosman, Dr. Steven James, Dr. Geraud Nangue Tasse

A research report submitted in partial fulfilment of the requirements for the degree of MSc in the field of Artificial Intelligence

in the

School of Computer Science and Applied Mathematics
University of the Witwatersrand, Johannesburg

4 June 2025

Declaration

I, Amir Esterhuysen, declare that this report is my own, unaided work. It is being submitted for the degree of MSc in the field of Artificial Intelligence at the University of the Witwatersrand, Johannesburg. It has not been submitted for any degree or examination at any other university.

Amir Esterhuysen

4 June 2025

Abstract

The idea of skill composition has been gaining traction within reinforcement learning research. This compositional approach promotes efficient use of knowledge and represents a realistic, human-like style of learning. Existing work has demonstrated how simple skills can be composed using Boolean operators to solve new, unseen tasks without the need for further learning. However, this approach assumes that the learned value functions for each atomic skill are optimal—an assumption that is violated in most practical cases. We thus propose a method that instead *learns* operators for composition using evolutionary strategies. Our approach is empirically verified first in a tabular setting and then in a high-dimensional function approximation environment. Results demonstrate outperformance of existing composition methods when faced with learned, suboptimal behaviours, while also promoting the development of robust agents and allowing for fluid transfer between domains.

Contents

Chapter 1

Introduction

The development of versatile agents capable of adapting to new contexts and challenges is a critical goal across multiple fields of artificial intelligence. Agents that are able to leverage their existing abilities into new forms are more robust and better equipped to deal with novel, complex situations. If designing a robot for long-term household assistance, it is not feasible to train the robot in every task it might face during a period of months or years within a constantly evolving backdrop. Instead, it could benefit from having a set of broadly useful skills and the ability to adapt or combine these skills to deal with the specific situations it encounters. Notably, this approach mimics the way in which humans actually learn—by gradually expanding their knowledge base of elementary skills and combining them in endless ways, instead of independently learning new abilities from scratch each time a novel problem arises.

These principles are especially relevant in the context of *reinforcement learning* (RL)—a field of artificial intelligence in which agents are guided towards desired behaviour by receiving feedback in response to their chosen actions. Here, we consider the idea of *composition*—the existing behaviours of an RL agent are combined to produce novel ones, without additional learning [1]. One particular approach is to combine skills according to the fundamental logical operations—disjunction (OR), conjunction (AND), negation (NOT)—using a fixed set of composition operators that are proven to produce optimal behaviour for tasks composed according to each of these operations. [2].

Although this approach is both interpretable [3] and results in the combinatorial

explosion of an agent's abilities [4], it suffers from a fatal issue: successful composition is only guaranteed when an agent has optimal knowledge of the behaviours being composed. This may be achievable in small synthetic environments, but if we are ever to apply this framework to real-world problems, we must also be capable of composing non-optimal behaviours. We identify three scenarios (further discussed in Section 3.1) in which this method of composition falters:

- RL algorithms often encode behaviours as value functions, but it is typically infeasible to train these value functions to convergence, especially in the case of large practical environments that may contain billions of states. An agent might have learned a set of good quality value functions capable of solving tasks in an environment, but which have not sufficiently converged to perform composition. It would be wasteful to discard these value functions, and training further might exceed a pre-defined computational budget.
- The specification of tasks may be such that the composition of said tasks is not semantically meaningful. For instance, we might have a reward function that allows an agent to learn tasks in the environment, but when attempting their composition incentivises the agent to travel in a loop and avoid the composed goal state.
- Stochastic dynamics in an environment can result in suboptimal composition—and catastrophic failures. Even when optimal value functions are used, this randomness can result in an agent taking invalid actions and preventing it from displaying logical behaviour.

Consider the case where a robot is trained in some environment to exhibit a set of skills, before being sent away for external use. The new environment would likely differ from the training environment to some extent, meaning that the original skills may not represent optimal behaviour in the new task setting, and thus preventing the successful use of Boolean composition. Furthermore, with the robot being divorced from its original training setup, it is impractical for users to forego composition altogether and learn the desired skills from scratch.

To this end, we propose that the agent learns how to compose behaviours, even

when these behaviours are suboptimal. We adapt the NEAT algorithm (Neuroevolution of Augmenting Topologies) [5] to learn an alternative set of robust composition operators that outperform the fixed Boolean algebra approach [2] when faced with the above-mentioned challenges that often manifest in practical RL environments. Essentially, this is done by initialising a population of neural networks and mimicking principles of natural evolution to incentivise the generation of networks that achieve the best performance when acting as operators for achieving the desired composed tasks. Our approach is shown to be viable first in a tabular setting, and then in an environment where function approximation is required. Most notably, we are able to learn operators in the tabular setting that can then be applied with positive results in the function approximation setting—despite these environments differing in scale and state space. This is particularly encouraging for the goal of producing robust RL agents.

1.1 Report Overview

In this introductory chapter we have contextualised the research problem, before providing a broad outline of our problem-solving approach.

Chapter 2 contains a more thorough grounding of the content underpinning the study—including a high-level background of RL principles, an examination of the current Boolean composition framework, and a discussion of the NEAT algorithm. We also present a review of the task composition literature directly informing this work, while touching on related forays into compositional RL.

Chapter 3 expands on this foundation and discusses the core areas in which the existing approach to Boolean composition fails. We detail the methods, metrics, and settings used to solve this problem. We also present pseudo-code illustrating how the NEAT algorithm is adapted to produce flexible composition operators.

Next, we move on to a discussion of experiments and results. Chapter 4 considers experiments conducted in the tabular setting. These experiments underline the applicability of learned operators in dealing with a number of practical RL scenarios. Chapter 5 examines the viability of our method in a high-dimensional setting where

function approximation techniques are necessary. A central discussion in this chapter focuses on the ability to learn operators within a tabular environment, and then apply them to composition tasks in a larger function approximation environment—either without modification or with minimal fine-tuning.

We conclude in chapter 6 by summarising our findings in the context of the original problem motivating this report. This also includes a discussion of limitations present in the work and how our approach could be refined in potential future expansions.

Chapter 2

Background and Related Work

2.1 Reinforcement Learning

Reinforcement learning (RL) is concerned with optimising an agent's behaviour through feedback from its environment. An agent observes its environment and on this basis makes decisions, which are then rewarded or penalised based on how effectively they drive the agent towards its goals.

A *task* in an RL environment is modelled as a *Markov decision process* (MDP) [6], given by a tuple of the form $(\mathcal{S}, \mathcal{A}, \rho, r, \gamma)$, where

- $\mathcal{S}$ is the *state space* containing all states in the environment.
- $\mathcal{A}$ is the *action space* containing all actions available to the agent.
- $\rho : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is the *transition function* giving the probability of an agent moving from state s to state s' via action a. We can express this function as $\rho(s,a,s') = Pr(S_{t+1} = s' \mid S_t = s \text{ and } A_t = a)$, with t indexing time steps in the MDP.
- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function* giving the reward received for an agent being in state s, performing action a, and through this entering state s'. $r(s,a,s') = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s']$, where R_t is the reward received at time step t.
- $\gamma \in [0,1]$ is the *discount factor* governing the degree to which the agent prioritises future vs. immediate rewards.

This framework is visualised in Figure 2.1.

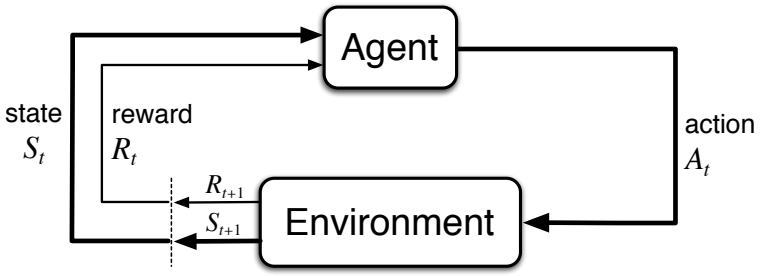


FIGURE 2.1: MDP Framework [6]. $t \in \mathbb{N}_0$ is the time index of the MDP.

In this setup, the agent transitions through states making up the environment. The agent has a set of available actions it can take, and at each time-step t it will be present at some state $S_t = s \in \mathcal{S}$, from which it executes an action $A_t = a \in \mathcal{A}$. The agent selects actions by following its *policy*, $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$. π gives the probability of taking action a from state s, and expressed more intuitively represents a strategy followed by the agent to solve the MDP. In response to the agent's action, the environment sends the agent to a successor state $S_{t+1} = s' \in \mathcal{S}$ according to the transition function ρ. The environment also emits a reward $R_t = r(s, a, s')$ according to the reward function, which the agent uses to guide its future behaviour. Fundamental to this process is the agent's *value function*, $V^\pi : \mathcal{S} \to \mathbb{R}$, where $V^\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \mid S_t = s\right]$. This gives us the total expected reward for an agent being in state $S_t = s$ and thereafter following its policy π. This is a measure of the 'goodness' or utility of each state when using π to solve the MDP. Value functions and policies are closely related, and given a value function an agent is able infer an associated policy.

We also make use of the term *episode*. An episode consists of the agent following a policy in the environment until some termination criteria is triggered—either having exhausted a pre-defined step limit or having reached a goal. In this study, episodes are finite but of varying length. An agent's performance is often measured by tracking the cumulative reward over a length of time, referred to as its *return*. Given an episode of length $n \in \mathbb{N}$ we can define the *episodic return* as $\sum_{t=0}^{n} R_t$,

and if given $N \in \mathbb{N}$ episodes where each episode i is of length n_i, we can define the *total return* as $\sum_{i=0}^{N} \sum_{t=0}^{n_i} R_t$

2.2 Q-Learning

Q-Learning [7] is a widely used RL algorithm, most applicable to tabular environments where each state can be represented explicitly. The algorithm learns an *action-value function*

$$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R},$$

defined as

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.1}$$

The expression defined in 2.1 gives the total expected reward for being in state s, taking action a, and thereafter following policy π. The term action-value function is used interchangeably with the term *Q-value function*, and we refer to $Q^\pi(s,a)$ as the *Q-value* for state s and action a.

The aim of learning is to discover the optimal action-value function Q^* such that $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, from which we derive the optimal policy π^* by using Q^* to act greedily at every state. Q-learning is built around a Bellman equation, which iteratively updates the Q-value for each state-action pair as the agent gains more information about its environment. The update equation takes the form $Q^\pi(s,a) = (1-\alpha) \cdot Q^\pi(s,a) + \alpha \cdot (r + \gamma \cdot \max_a Q^\pi(s',a))$, where $\alpha \in (0,1)$ is the learning rate parameter which governs convergence of the algorithm.

2.3 Deep Q-Learning

In Q-Learning, the action-value function is represented as a table containing values for each state-action pair. This structure is referred to as a *Q-table*. As the state space $\mathcal{S}$ and the action space $\mathcal{A}$ become larger, the Q-table grows exponentially in

size. In higher-dimensional function approximation environments, storing and updating a Q-table uses a great deal of system memory. Additionally, it is extremely time consuming for an agent to visit each state repeatedly and build up the requisite knowledge for the Q-table.

Deep Q-Learning [8] is an extension to Q-Learning, more suitable for large or complex environments. The action-value function is now represented as a *Deep Q-Network* (DQN)—a neural network which takes states in the environment as input and outputs an action along with its associated value. A policy is determined on the basis of these action-value pairs for a given state. Algorithm 1 illustrates how the Deep Q-Learning process [8] is used to approximate the optimal action-value function Q^* with the DQN Q, where $Q(s, a; \theta) \approx Q^*(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Note that Q is parameterised by the set of weights θ.

Training is performed using the loss function $L_i(\theta_i) = (y_i - Q(s, a; \theta_i))^2$, defined for each iteration i of the algorithm. The idea is to push our current estimate $Q(s, a; \theta_i)$ closer to the target value $y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$. Differentiating this loss function with respect to θ produces the gradient

$$\nabla_{\theta_i} L_i(\theta_i) = \big(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\big) \nabla_{\theta_i} Q(s, a; \theta_i) \tag{2.2}$$

A key concept in Deep Q-Learning is *experience replay*, the idea that an agent's choice of action during training is informed by a random selection of previous action-reward experiences. Another important feature is the use of a *target neural network* in addition to the main Q-network being learned. The target network is initialised with the same architecture and weights as the principal Q-network, but these weights are updated at a delayed rate compared to the Q-network. Without the target network, the agent's predicted Q-values would be dependent on the same set of parameters that are being optimised during the algorithm's run—essentially leading to the agent chasing a constantly moving target and with the potential for drastic oscillations during training. The target network remedies this issue by keeping the parameters for target prediction temporarily static, and allowing for a smoother convergence without exceedingly volatile updates.

Algorithm 1 Deep Q-learning with Experience Replay [8]

Initialize:
replay memory $\mathcal{D}$ to capacity N
action-value function Q with a random set of weights θ
target action-value function $\hat{Q}$ with weights $\theta' = \theta$
for episode = 1, M **do**
Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
for t = 1, T **do**
With probability ϵ select a random action a_t,
otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

$$\text{Set } y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta') & \text{for non-terminal } \phi_{j+1} \end{cases}$$

Perform gradient descent step on loss $L_j = (y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 2.2
end for
end for

2.4 World Value Functions

Previously, we have introduced the role of state space $\mathcal{S}$ in the RL framework. We now define the *goal space* $\mathcal{G} \subseteq \mathcal{S}$. This is the set of all goal states in an environment. If reaching a certain goal causes the episode to end, this state is referred to as *terminal*. In some cases, there is a direct correspondence between the notions of goals and tasks. Consider, for instance, the environment depicted in Figure 2.2, containing three tasks: `Red`, `Green`, and `Pink`. Each task corresponds to the agent having successfully reached the goal state of that colour—completing the task `Red` means reaching the red cell in the top left corner of the grid. If we have learned an

action-value function Q_{Red} for this task, acting greedily with respect to Q_{Red} produces the policy π_{Red}. Following π_{Red} allows the agent to reach the state `Red` and simultaneously solve the task.

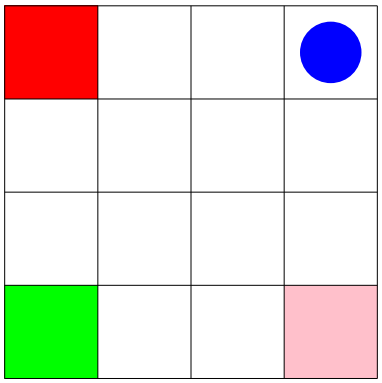


FIGURE 2.2: Gridworld with 3 goal states. The agent, illustrated by a blue circle, can move in any of the cardinal directions and must navigate to one of the coloured squares.

This process, however, is not as straightforward when we wish to perform composition over tasks made up of multiple sub-goals. At this point, it is important to distinguish between two classes of tasks: *base tasks*, which are independently learned through standard methods (e.g., Q-Learning), and *composed tasks*, which are derived by combining or manipulating the base tasks.

Remaining in the same environment, say we now have two base tasks expressed as `(Red OR Green)` and `(Green OR Pink)`. Clearly, the intersection of these tasks is `Green`. The action-value function for each task only provides knowledge of how to reach the currently most desirable goal (based on the reward achieved by navigating there) from each state $s \in \mathcal{S}$. Hence, there may be states at which neither value function provides knowledge of `Green`.

To overcome this, Nangue Tasse, Rosman, and James [9] introduce *world value functions*, a family of goal-oriented value functions. Given state space $\mathcal{S}$, goal space $\mathcal{G}$, and action space $\mathcal{A}$, we can define the world value function as

$$\bar{Q}^{\bar{\pi}}(s,g,a,s') = \bar{r}(s,g,a,s') + \gamma \sum_{s' \in \mathcal{S}} \rho(s,a,s') \bar{V}^{\bar{\pi}}(s',g) \tag{2.3}$$

where

- $\bar{r}$ is the *extended reward function* given by

$$\bar{r}(s,g,a,s') = \begin{cases} \bar{r}_{\text{min}} \in \mathbb{R} & g \neq s \text{ and } s \in \mathcal{G} \\ r(s,a,s') & \text{otherwise.} \end{cases}$$

  $\bar{r}$ penalises the agent for achieving an undesired goal, thereby learning the currently desired goal while also encoding knowledge of all other goals in the environment. $\bar{r}_{min} \in \mathbb{R}$ is chosen to bound $\bar{r}$ from below. Formally, $\bar{r}_{min} \leq \min\{r_{min}, (r_{min} - r_{max})D\}$, where D is the MDP diameter [10]. In practice, we can set $\bar{r}_{min}$ to be any large negative number.

- $\bar{V}^{\bar{\pi}}$ is an *extended value function* given by $\bar{V}(s,g) = \mathbb{E}^{\bar{\pi}}\big[\sum_{t=0}^{\infty} \bar{r}(s_t,g,a_t)\big]$. $\bar{V}^{\bar{\pi}}$ assigns values to state-goal pairs. Knowledge of all goals is stored and the agent explicitly incorporates the current goal when choosing actions.

- $\bar{\pi} : \mathcal{S} \times \mathcal{G} \times \mathcal{A} \to [0,1]$ is the associated *world policy*. This policy can be followed to reach any goal in the environment.

These world value functions allow us to represent knowledge of all goals in the environment, not just the currently most desirable goal. A Q-Learning approach built around Bellman's equation is still used for learning world value functions, but with added specification of the goal space.

Optimality is a topic of focus in RL. The optimal world value function is given by $\bar{Q}^*(s,g,a) = \max_{\bar{\pi}} \bar{Q}^{\bar{\pi}}(s,g,a)$, where $\bar{\pi}^*$ is the associated optimal world policy. $\bar{Q}^*$ satisfies the Bellman optimality equation [4] given by

$\bar{Q}^*(s,g,a) = \sum_{s' \in \mathcal{S}} \rho(s,a,s')\big(\bar{r}(s,g,a) + \bar{V}^*(s',g)\big)$, where $\bar{V}^*(s',g) = \mathbb{E}_{\bar{\pi}^*}\big[\sum_{t=0}^{\infty} \bar{r}(s_t,g,a_t)\big]$.

In practice we use algorithms, like Q-Learning, which converge towards optimal value functions.

2.5 Composition with Boolean Algebra over Tasks

One form of composition uses a *Boolean task algebra* framework defined over the domain of RL tasks [2], in which we derive a composed task value function by applying a composition operator to the value functions of the base tasks. This method develops its theoretical guarantees on the assumption that we have access to optimal world value functions, and violating this assumption can lead to unforeseen problems when composing tasks. The same authors further discuss the precise conditions for optimality and their consequences in subsequent work [11].

The Boolean algebra leads to provably optimal operators (again, given optimal world value functions) for performing *disjunction* (OR), *conjunction* (AND), and *negation* (NOT) over the domain of RL tasks. We also maintain the following assumptions:

- The tasks have the same state and action spaces.
- The tasks have the same transition dynamics, and these transitions are deterministic.
- The reward function differs between tasks only at the terminal goal states, and is the same at all non-terminal states.
- The reward function for each task has only two possible outcomes at terminal goal states, representing the reward for desired and undesired goals.

Now consider two tasks with optimal world value functions $\bar{Q}_1^*$ and $\bar{Q}_2^*$. Let $\bar{Q}_{min}^*$ be the optimal world value function in the case where all goals in the environment are marked as undesirable, and $\bar{Q}_{max}^*$ be the optimal world value function when all goals are desirable. For $(s, g, a) \in \mathcal{S} \times \mathcal{G} \times \mathcal{A}$, tasks are composed according to the three fundamental logical operations:

- *disjunction*: $(\bar{Q}_1^* \vee \bar{Q}_2^*)(s, g, a) = \max\{\bar{Q}_1^*(s, g, a), \bar{Q}_2^*(s, g, a)\}$
- *conjunction*: $(\bar{Q}_1^* \wedge \bar{Q}_2^*)(s, g, a) = \min\{\bar{Q}_1^*(s, g, a), \bar{Q}_2^*(s, g, a)\}$
- *negation*: $(\neg\bar{Q}_1^*)(s, g, a) = (\bar{Q}_{min}^*(s, g, a) + \bar{Q}_{max}^*(s, g, a)) - \bar{Q}_1^*(s, g, a)$

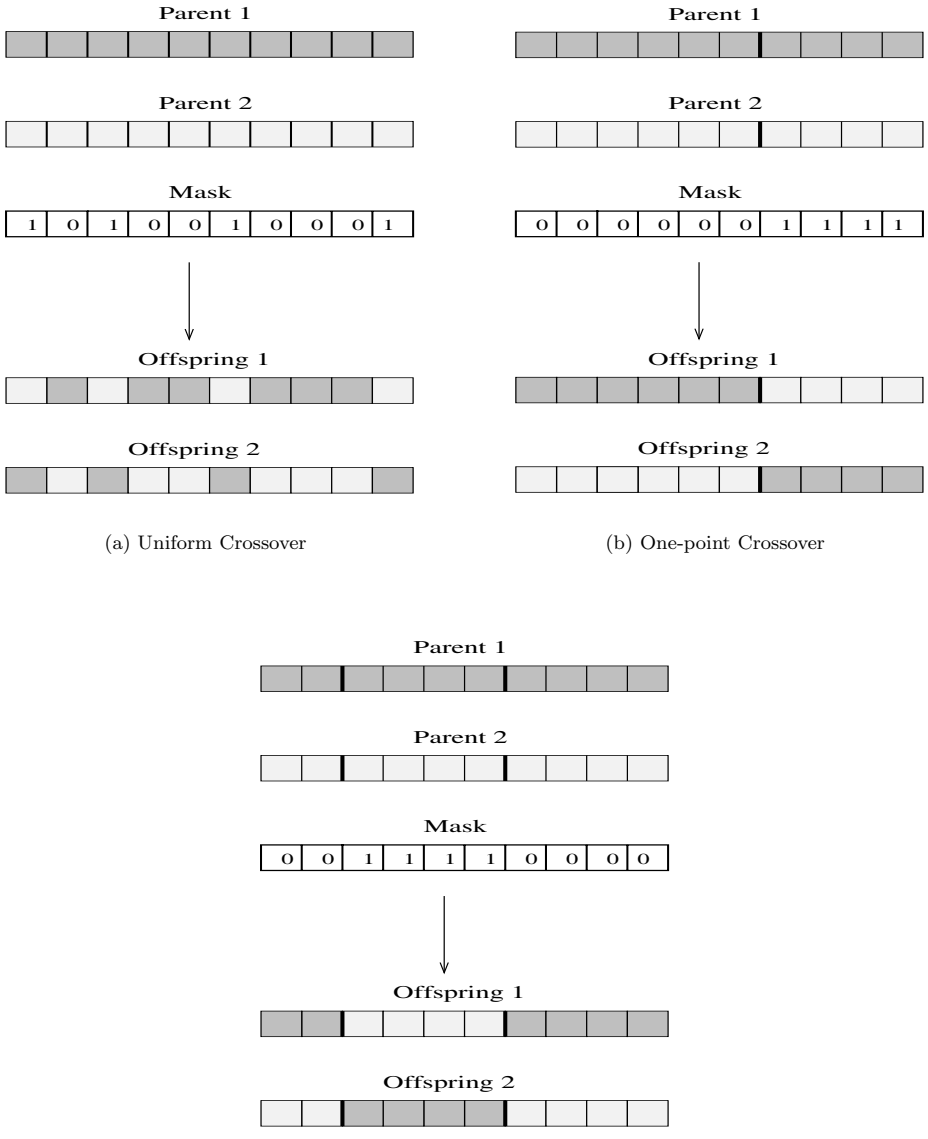
Given optimal value functions for the base tasks, these operations will produce optimal value functions for the composed tasks. Note that the *min* and *max* functions are used as in a standard Boolean algebra. If 1 represents True and 0 represents False, then the conjunction and disjunction of two Boolean variables x, y will be respectively $\min(x, y)$ and $\max(x, y)$. We can use a similar formulation in the case of negation: The term $\bar{Q}^*_{min}(s, g, a) + \bar{Q}^*_{max}(s, g, a)$ can be thought of as equivalent to 0 + 1. Then for Boolean variable x, we have $0 + 1 - x = 0$ for $x = 1$ and $0 + 1 - x = 1$ for $x = 0$.

2.6 Genetic Algorithms

Reinforcement learning optimises behaviour by incentivising approaches that produce desired results and penalising those producing undesired results. This is reminiscent of how humans actually learn. A similar optimisation process is present in the *genetic algorithm*—also inspired by natural systems. Genetic algorithms use principles of biological evolution to generate effective solutions to computational problems. The process of optimising solutions is driven by a combination of natural selection—where attributes of the most 'fit' members in a *population* are combined and passed down over time to subsequent *generations*, and *mutation*—where solutions undergo random alterations to inject genetic diversity into the population and avoid falling into local optima [13].

Given an optimisation problem where we seek a solution that achieves the best result for some *fitness* measure, we start with a randomly initialised population of candidate solutions. Solutions are represented as *genomes*, and are evaluated according to this fitness measure during each generation of the algorithm run. In this discussion we are focused on providing a high-level overview, and so highlight the traditional *binary encoding* genome representation—solutions are represented as strings of 1s and 0s where each bit corresponds to a particular solution attribute. Note, however, that alternative formulations like *value encoding* and *tree encoding* have become more popular within the modern GA landscape where representing solutions as binary is not always feasible.

A selection of the best solutions is taken from the population and combined to



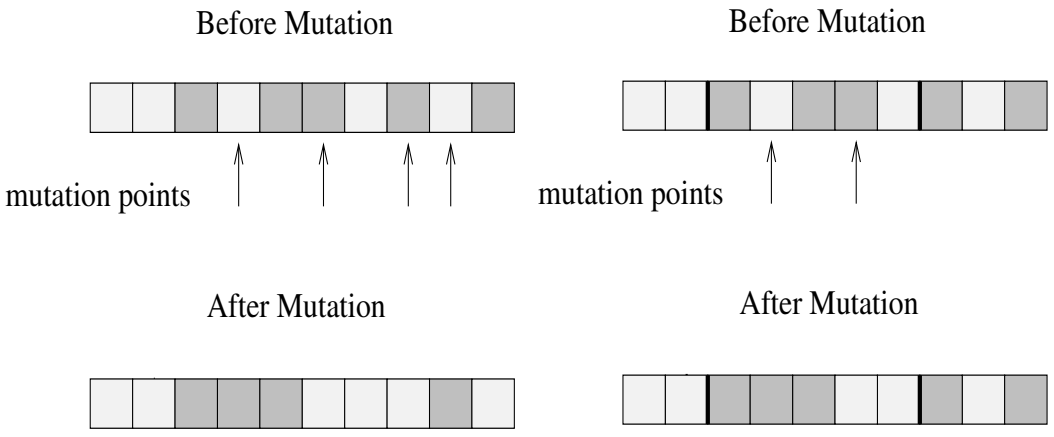
(a) Uniform Crossover

(b) One-point Crossover

(c) Two-point Crossover

FIGURE 2.3: Three crossover variants - uniform, one-point, and two-point [12].

produce *child* solutions which inherit important attributes from their *parents*. This *mating* operation is performed using *crossover* of genome representations. Three crossover variants are depicted in Figure 2.3. A mask is used to indicate which attributes are taken from each parent, and from this a child solution is produced. In *uniform crossover* this mask is randomly generated, while in *one-point crossover* and *two-point crossover* we divide the mask into sections for each parent. A second child is produced by inverting the first.



(A) Random mutation (B) Inorder mutation

FIGURE 2.4: Two mutation variants. a shows random mutation. All bits in the string are candidates for inversion, and the arrows point to those which have been selected. b shows inorder mutation. Bold lines are used to demarcate a segment of the string, and the bits within this segment are candidates for inversion. Arrows point to the selected bits. [12]

In reality, children are not simply the sum of their parents. A mutation operation is performed to diversify the number of potential solutions being explored and to prevent the population from stagnating. Figure 2.4 illustrates random and inorder mutation. In *random mutation*, a number of bits across the whole string are randomly selected and then inverted. With *inorder mutation*, two mutation points are randomly selected to create a segment of the original string. Then, a number of bits within this segment are randomly selected for inversion—that is, random mutation is applied not to the whole string but only a segment.

The process of evaluating fitness, mating, and mutating is repeated either for a chosen number of generations, until a satisfactory solution has been reached, or until no further progress is being made.

2.7 NEAT

The NEAT algorithm, standing for Neuroevolution of Augmenting Topologies, was introduced by Stanley and Mikkulainen [5]. NEAT uses genetic algorithm principles to evolve solutions which themselves are neural networks. Typically, neural network training algorithms (e.g. backpropagation) optimise the weight configurations for a pre-defined network architecture or *topology*. NEAT simultaneously optimises network weights and the overall network topology. This allows us to explore a much larger space of candidate networks for a given problem. Importantly, NEAT does not require gradient computations, and hence does not require that our fitness function be differentiable with respect to the network parameters. This is useful in RL applications—we might receive a discrete reward at each time step, before outputting a final scalar return as our fitness. However, there is no meaningful relationship between this return and the individual weights. NEAT is able to handle this situation, and we can proceed without modifying our desired non-differentiable fitness function.

A solution within a run of NEAT is represented as a genome containing two lists. The first list identifies nodes in the network, allowing new nodes to be added and existing nodes that are not inputs or outputs to be marked as inactive. The second list identifies connections between nodes in the network. A connection is represented by its input and output nodes, as well as its weight value. Connections can also be added or disabled in this representation. An individual characteristic of a genome—like a particular connection between two nodes—is referred to as a *gene*. Figure 2.5 illustrates this genetic representation.

NEAT employs two mutation variants to add diversity to a population of solutions. *Structural mutation* involves the addition of a node or connection to a candidate network, while *weight mutation* alters the weight values along existing connections. Mutation occurs randomly according to a chosen probability.

Each gene in a NEAT solution contains an *innovation number*. This is a unique marker used to keep track of new structural mutations entering the population. Whenever a structural mutation occurs, e.g. introducing a connection between two unconnected nodes, the algorithm checks whether this particular mutation had been previously observed at any point during the run. If yes, it is marked with the innovation number that was originally assigned upon the mutation's first appearance. Otherwise, it is assigned a new innovation number by incrementing the latest number used. Innovation numbers are preserved and passed down through generations as the population evolves, acting as a record of the solution characteristics that have been trialled on the way towards our optimisation goal.

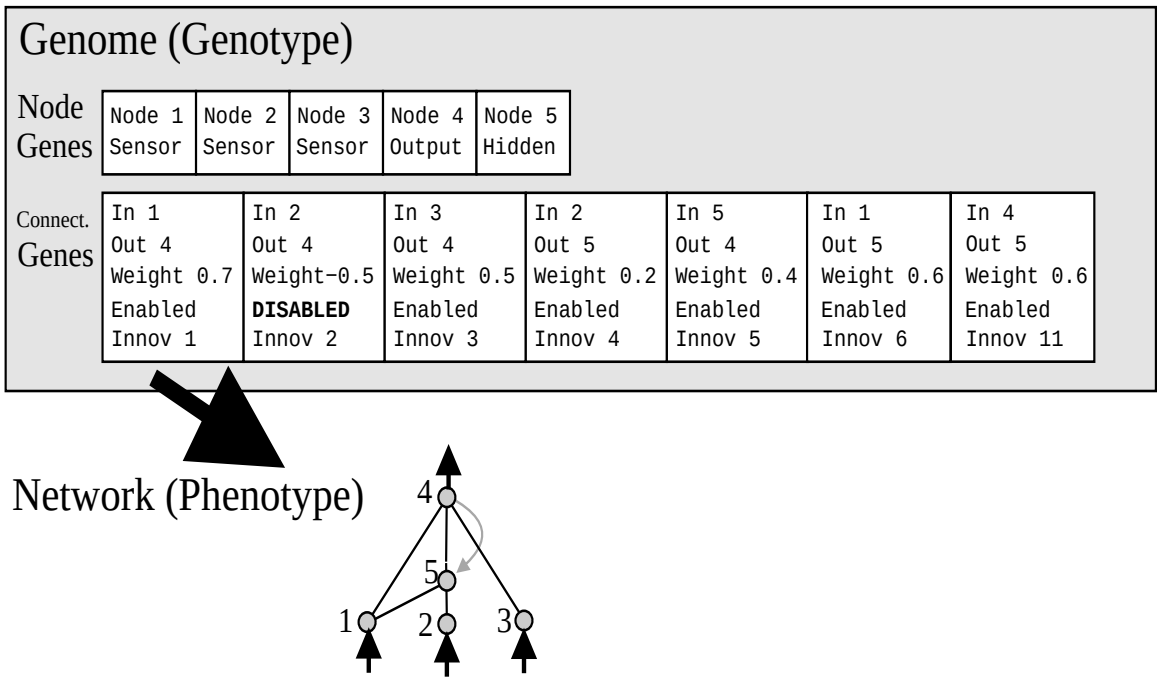


FIGURE 2.5: NEAT solution representation, containing a list of nodes for a given network and list of connection details between nodes [5].

Innovation numbers are also fundamental to NEAT's mating process. When two parent solutions are combined, their respective genes are aligned according to innovation number. If both parents share a gene (indicated by a common innovation number), this is passed on to the child. Weights are randomly inherited from one of the parents. Other genes, possessed by only one parent, are inherited along with

relevant weights from the more fit parent. Using innovation numbers allows for the smooth combination of structurally compatible networks. Mating in NEAT is depicted in Figure 2.6

A key feature of NEAT is *speciation*. The population of networks is divided into species based on similarity between them, and parent selection is done within a species rather than across the whole population. This allows for the development of evolutionary paths in which solutions might initially perform poorly, whereas otherwise initial poor fitness might lead to these evolutions being discarded without having the opportunity to produce fruitful results later on.

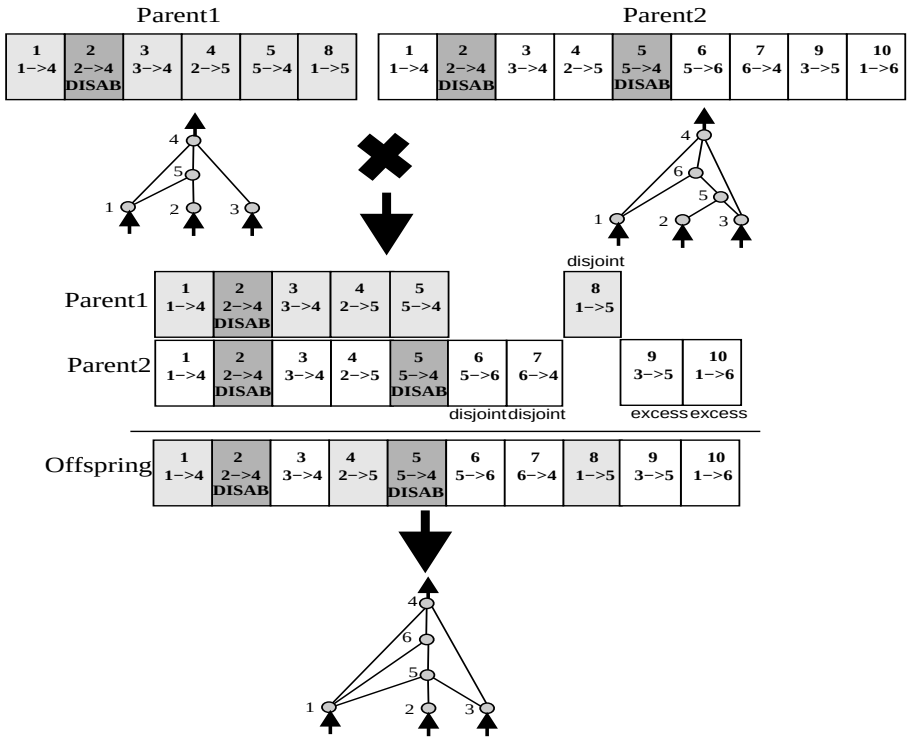


FIGURE 2.6: NEAT mating procedure [5].

Speciation is guided by *genetic distance*, a measure of similarity between networks in the population. Let G_1 and G_2 be genomes, respectively made up of N_1 and N_2 genes. Then we have $N := max\{N_1, N_2\}$. Further, we let D be the number of *disjoint* genes—exclusive to one genome but with a structural equivalent in the other genome (for example, a connection to the left of a base node rather than to its right). These differ from *excess* genes, which are exclusive to one genome but

do not have structural equivalent in the other (for example, the introduction of a new node which is directly connected to the output but not to any other part of the architecture). All other genes are said to be *matching*, and $\overline{W}$ is the mean weight difference between the matching genes. We use these quantities to define the genetic distance δ between G_1 and G_2, given by $\delta = c_1\frac{D}{N} + c_2\frac{E}{N} + c_3\overline{W}$, for coefficients $c_1, c_2, c_3 \geq 0$.

2.8 Related Work

We briefly examine some of the research into task composition directly underpinning this study, as well as forays into different areas of compositional RL.

Todorov introduced the *linearly-solvable MDP* to compose value functions [14]. However, there is no correspondence in Todorov's work between the ideas of logical composition in sets and composition in the domain of RL tasks. Nangue Tasse, James, and Rosman [2] bridge this gap with the *Booolean task algebra* over MDP-modelled tasks (Section 2.5). The same authors extend this work to discounted tasks, also proposing a method to determine whether a given task can be composed from existing knowledge [4].

Cheng et al. [15] consider the composition of motion tasks for robots operating in dynamic non-Euclidean spaces. These motion tasks are represented as *Riemannian motion policies* (RMPs)—a robotics framework which combines different motion objectives using Riemannian geometry to generate natural and efficient movements. The work introduces *RMPflow*, an algorithm for combining RMPs into a compositional policy across the task space. This is expanded upon by Li et al. [16], who propose a modified version of RMPflow called *RMP*^{2}. This modification leads to improved accessibility as well as gains in efficiency.

Qiu and Zhu [17] take a programmatic approach to composition. First, they propose an interpretable RL framework built around program-based policy architectures. They then show how these architectures can be integrated with a set of *primitive functions*, built to represent basic skills with applicability across several contexts. Composition is performed by leveraging these primitive functions according to an

agent's perception of its current environment.

Nam et al. [18] explore RL composition through the lens of *meta-learning*. Given a large dataset storing agent skills, they meta-train an overarching policy for performing skill composition. Here, emphasis is on the agent discovering transferable skills and being able to adapt the learned policy to unseen tasks.

Composition is also relevant in the space of *hierarchical RL*. Long-term problems are divided into a hierarchy of *sub-problems*, and an agent learns policies for these sub-problems which can then be reused [19]. A related idea is that of *options*, extended *macro-actions* made up of multiple *low-level actions*. Sutton et al. [20] define an option as a tuple $(\mathcal{I}, \pi, \beta)$. Here, $\mathcal{I} \subseteq \mathcal{S}$ is an *initiation set* containing the states that make up the option, β is the probability of termination, and π is a policy.

Stolle and Precup [21] illustrate how learning options can be used for more robust solving of goal-reaching tasks in practical environments. However, options and other hierarchical methods are best suited to single-goal tasks [4], not to the flexible composition of multi-goal tasks considered here.

Peng et al. [22] propose *multiplicative compositional policies* (MCP) as an extension to hierarchical RL. Behaviours are broken down into a set of *primitive policies* which can be reused and composed together. This allows an agent to generate a large number of novel behaviours by combining primitives. They demonstrate their results by simulating complex motion. Singh et al. [23] propose using generative modelling techniques to learn *behavioural prior distributions*. An agent uses these priors to streamline their exploration by attempting behaviours that have already been shown as useful.

We also consider the related study of *life-long learning*, where agents are left to interact with an environment that continues challenging them in new ways. Mendez and Eaton [24] break down the agent's goals under this framework: 1) successfully adapting to unseen tasks, 2) retaining the ability to perform previously learned tasks, and 3) not depending on the permanent storage of large models or datasets. Agents are pushed to harness *reusable knowledge* which can be manipulated into

new forms depending on the current task. The idea of reusable knowledge is employed by Lee et al. [25], who, given a set of pre-trained skills, propose a method for learning policies that transition between them.

2.9 Concluding Remarks

This study lies at the intersection of two areas—reinforcement learning and evolutionary computation. The background chapter provides necessary context to each of these areas. On the RL side, this comprises an overview of the field's core principles and the relevant algorithms deployed. We focus on the idea of composition and detail the Boolean task algebra framework. While the Boolean algebra motivates our study, we also discuss alternative approaches to compositional RL found in the literature. The evolutionary angle is introduced through a discussion of genetic algorithms, before segueing into NEAT and its method of evolving neural network solutions. Based on this foundation, the following chapter presents a more focused examination of our research problem and the techniques used to combat it.

Chapter 3

Research Methodology

3.1 Research Problem

While prior work has demonstrated how to compose value functions with a fixed set of Boolean operators, there are certain assumptions regarding the underlying tasks and set of value functions that must be satisfied. When these assumptions are violated, it is possible for such operations to result in undesirable behaviours. This might involve an agent getting stuck in a certain position or travelling in a loop. We identify three potential failure modes of this approach that are likely to occur in practice.

Standard suboptimality: The composition operators defined in Section 2.5 are guaranteed to hold only in the case of optimal value functions. This is problematic for a number of reasons. It is typically infeasible to train value functions until convergence. Instead, learning takes place until the behaviour demonstrated by the agent from a set of start states is desirable. As a result, these value functions will likely have incorrect estimates for states that are not along the optimal trajectory. This can lead to errant agent behaviour, for instance getting stuck in a loop between two states. Nevertheless, such value functions are useful as they might allow an agent to achieve a given task. Furthermore, value functions—even if suboptimal—are often time-consuming to obtain, and so it would be undesirable simply to discard them.

Stochastic environments: Challenges can occur as a result of the environment's dynamics, or due to noisy sensors or actuators of the agent. Although an agent may

act optimally under stochastic dynamics, unpredictability in the environment hinders optimal composition, and existing theory shows that optimal value function composition is only guaranteed to hold under deterministic transition dynamics [1].

Non-viable reward functions: It is possible for the reward function of individual tasks to be specified such that the individual tasks are correct and meaningful, but their composition is not. For example, the reward may be such that an agent is incentivised to travel in a loop so that it avoids both desired and undesired goals. Consider the environment in Figure 3.1. There are two tasks: `bottom` (navigating to the pink or green goals) and `left` (navigating to the red or green goals). The intersection of these tasks is reaching the green goal. There is no discount factor, and the reward function is such that desirable and undesirable goals carry a reward of $+1$ and -1, respectively, while a reward of -0.7 is given for each step the agent takes. If we denote the agent's current state in the top right corner as s, we have that:

$$\max_a \bar{Q}_{\texttt{bottom}}(s, \text{green}) = 5(-0.7) + 1 = -2.5$$
$$\max_a \bar{Q}_{\texttt{left}}(s, \text{green}) = 5(-0.7) + 1 = -2.5$$

From here, we get the composed task value:

$$\max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{green}) = \min\left\{\max_a \bar{Q}_{\texttt{bottom}}(s, \text{green}), \max_a \bar{Q}_{\texttt{left}}(s, \text{green})\right\} = -2.5$$

But we can also see that:

$$\max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{red}) = 2(-0.7) - 1 = -2.4$$
$$\max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{pink}) = 2(-0.7) - 1 = -2.4$$

So:

$$\max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{green}) < \max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{red}) = \max_a \bar{Q}_{\texttt{bottom}\wedge\texttt{left}}(s, \text{pink})$$

This means that the composed task does not maximise over possible return. As such, the Boolean algebra framework is unable to produce an appropriate value function that achieves the conjunction of our two base tasks. If we were to replace the *min* conjunction operator with something like $\min\{-2.5, -2.5\} + 1 = -1.5$ then this will achieve the desired conjunction. Adding $+1$ is an arbitrary choice for this example—the larger point is the presence of scenarios that cannot be solved by fixed Boolean operators yet which are amenable to some alternative operator.

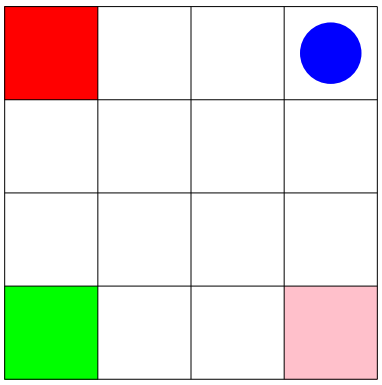


FIGURE 3.1: Gridworld environment with `bottom` (green or pink) and `left` (red or green) tasks. The agent, illustrated by a blue circle, can move in any of the cardinal directions and must navigate to one of the coloured squares.

Assuming that we work with a fixed set of value functions, the Boolean algebra does not cope with these issues, and the option of further training may not be available to us. Alternatively, even when additional training is possible, it may be expensive or impractical—especially as all possible compositions would have to be accounted for. Learning a new set of composition operators allows for flexible circumvention of challenges to optimality in the environment, without requiring us to modify the underlying set of value functions.

3.2 Using NEAT to Learn Robust Composition Operators

Recall the Boolean algebra defined in Section 2.5 for optimal world value functions $\bar{Q}_1^*$ and $\bar{Q}_2^*$. Now assume that instead we have suboptimal world value functions $\hat{Q}_1$ and $\hat{Q}_2$. We also let $\hat{Q}_{min}$ and $\hat{Q}_{max}$ be the minimum and maximum world value functions. These are trained over the same number of episodes as $\hat{Q}_1$ and $\hat{Q}_2$. Our goal is to produce alternative operators $f_\vee, f_\wedge, f_\neg$—each a neural network—which outperform their Boolean algebra equivalents when given suboptimal input:

- $(\hat{Q}_1 \vee \hat{Q}_2)(s,g,a) \approx f_\vee\big(\hat{Q}_1(s,g,a), \hat{Q}_2(s,g,a)\big)$
- $(\hat{Q}_1 \wedge \hat{Q}_2)(s,g,a) \approx f_\wedge\big(\hat{Q}_1(s,g,a), \hat{Q}_2(s,g,a)\big)$
- $(\neg\hat{Q}_1)(s,g,a) \approx f_\neg\big(\hat{Q}_1(s,g,a), \hat{Q}_{min}(s,g,a), \hat{Q}_{max}(s,g,a)\big)$

We can further define the optimisation problem. Consider a set $\hat{\mathbf{Q}}$ containing a world value function for each base task in the environment. Let f^θ be a neural network composition operator whose weights and architecture are collectively specified by the parameter θ. We can produce a new composed-task world value function $\hat{Q}^\theta = f^\theta(\hat{Q}_{base})$ where $\hat{Q}_{base} \subseteq \hat{\mathbf{Q}}$. From this, we act greedily at each state to derive the policy $\hat{\pi}^\theta$ which is followed to achieve the composed task.

Now, assume we are evaluating operator performance over $N \in \mathbb{N}$ episodes. We wish to discover the network specification θ that maximises reward over this set of episodes. That is, we want $\underset{\theta}{\text{argmax}}\, \mathbb{E}_{s_0 \sim U(\mathcal{S})}\left[\sum_{n=1}^{N} V^{\hat{\pi}^\theta}(s_0)\right]$, where the notation $s_0 \sim U(\mathcal{S})$ indicates that the agent's initial state s_0 at each episode is uniformly sampled from the state space $\mathcal{S}$.

Neural network training algorithms tend to optimise the weight configurations for a pre-defined network architecture. In this case, determining a suitable architecture is part of the optimisation problem. We thus turn to NEAT, which uses a genetic algorithm to optimise network weights and the overall network topology simultaneously—exploring a larger space of candidate networks and not requiring

gradient computation. More on the workings of NEAT can be found in Section 2.7. Of particular importance is the fitness function via which we compare solutions. As in our formulation of the optimisation problem, we use total reward over a chosen number of episodes as the default fitness measure.

Algorithm 2 shows how the underlying principles of NEAT are adapted to learn robust neural network composition operators. Input is a set $\hat{\mathbf{Q}}$ of base task world value functions **(1)**. Three hyperparameters are most influential in any run—the initial population size, the number of generations before termination, and the number of episodes over which we evaluate fitness. More discussion on the role of parameters in NEAT can be found in Section 3.3. Learning an operator for each composition type (OR, AND, NOT) requires its own run, so the desired type is also supplied. For a given composition type (e.g., AND), there are multiple skill compositions possible (e.g., task 1 AND task 2, task 2 AND task 3). At the start of an evaluation episode, one such skill composition is randomly assigned **(7)**. This produces operators capable of achieving all variations of a composition type.

During each generation, we evaluate the fitness of population members **(9)**. After the final generation, we recover the set $f_{best} = \{f^{\theta_1} \dots f^{\theta_M}\}$ of highest-fitness networks found during the run **(25)**. Since performance of all observed networks is recorded, $M \in \mathbb{N}$ is arbitrarily chosen. Each $f^{\theta_m} \in f_{best}$ acts as an operator for performing the desired composition type. Note that each run produces operators only for a single desired composition type. For example, if we are performing conjunction, then $f_{best} = f_{\wedge_{best}} = \{f_{\wedge}^{\theta_1} \dots f_{\wedge}^{\theta_M}\}$.

Algorithm 2 Learning Composition Operators

Inputs:
Desired Composition type (OR/AND/NOT), set of base task
world value functions $\hat{\mathbf{Q}} = \{\hat{Q}_1 \dots \hat{Q}_n\}$, where
$\hat{Q}_i : \mathcal{S} \times \mathcal{G} \times \mathcal{A} \to \mathbb{R}$ for all $i = 1 \dots n$.
Initial Population Size *Num_Pop*
Number of generations *Num_Gen*
Number of evaluation episodes *Num_Eval*
Initialize:
population of *Num_Pop* networks
for *generation* = 1 to *Num_Gen* **do**
  for $f \in population$ **do**
    $fitness \leftarrow 0$
    for *evaluation episode* = 1 to *Num_Eval* **do**
      Randomly generate specific desired composition within type (eg `RED OR BLUE`)
      Fetch relevant value functions $\hat{Q}_{base} \subset \hat{\mathbf{Q}}$
      while *episode not terminated* **do**
        $\hat{Q}_{composed} \leftarrow f(\hat{Q}_{base})$
        Randomly generate starting state s for agent
        Select action $a = \arg\max_{a \in \mathcal{A}} \hat{Q}_{composed}(s, g, a)$
        Execute action a to collect reward r and reach state s'
        $fitness \leftarrow fitness + r$
      end while
    end for
    Assign final $fitness$ to f
  end for
  Arrange all $f \in population$ by $fitness$
  Discard networks $f \in population$ with lowest $fitness$
  Choose networks $f \in population$ with highest $fitness$ to act as *parents*
  Mate *parents* together to produce *children* which are added to *population*
  Randomly mutate *children* to add variety to *population*
  Update *population* going into next *generation*
end for
return $f_{best} = \{f^{\theta_1} \dots f^{\theta_M}\}$, the M highest-fitness networks produced across entire run

3.3 NEAT Parameters

A run of NEAT is governed by a group of parameters. We highlight the most influential of these:

- The initial *population size*, *number of generations* before NEAT terminates, and *number of evaluation episodes* to determine fitness of a given network. All defined over $\mathbb{N}$.
- The potential *activation functions* considered during a run. Any number of customisable functions are possible, however we typically restrict our search to the following functions: *sigmoid*, *relu*, *tanh*, *sin*, *softplus*, as well as the identity and absolute value functions.
- The respective probabilities of adding or removing nodes and connections, and the *mutation rates* at which network weights, responses, and choice of activation function are modified. All defined over $[0, 1]$.
- *Elitism*: the number of highest-fitness networks in the population that are carried over to the next generation without mutation, defined over $\mathbb{N}$.
- *Compatibility threshold*: the genomic distance at which two networks are grouped in the same species, defined over $\mathbb{R}$.
- *Survival threshold*: the proportion of highest-fitness population members engaging in mating, defined over $[0, 1]$.
- The *number of stagnant generations* without improvement before a species is disregarded, defined over $\mathbb{N}$.

Code for the base NEAT implementation is found in the NEAT-Python library [26], and its documentation contains a comprehensive survey of the tunable parameters in a run of NEAT—within the section titled 'Configuration file description'. Because there is a large number of parameters which take different forms—some real-valued, some integer-valued, some qualitative—this task is not well suited to standard parameter-tuning. Instead, we test empirically to find a set of high-performing parameters. In general, running for more generations with a greater population size

should yield better results, but this must be weighed against increased time burden.

3.4 Network Structure

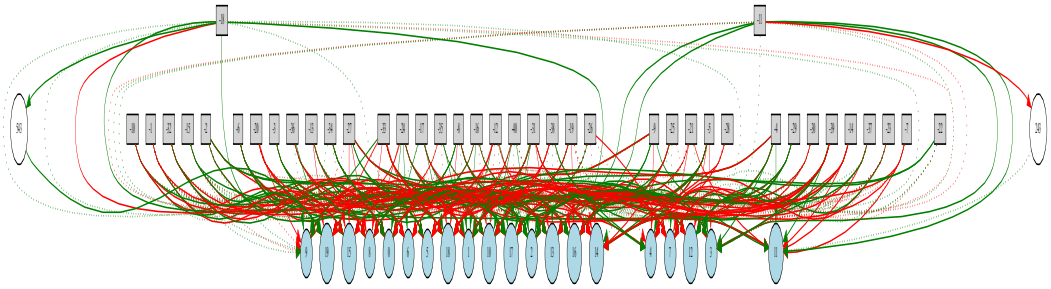


FIGURE 3.2: Complex Network with $\mathcal{G} \times \mathcal{A}$ input and output. This network acts as a conjunction operator.

Fundamentally, networks produced by NEAT take in a set of base-task value functions and provide a composed-task value function. However, there is some choice in how we represent these inputs and outputs. Consider the case, in a tabular environment, where we have state space $\mathcal{S}$, goal space $\mathcal{G}$, and action space $\mathcal{A}$. We want to achieve the intersection between two tasks via their world value functions $\hat{Q}_1^*$ and $\hat{Q}_2^*$. The value functions are stored as arrays of size $|\mathcal{S}||\mathcal{G}||\mathcal{A}|$. If we know that the agent is in state $s \in \mathcal{S}$, we can eliminate the state space dimension and represent the world value function at s as an array of size $|\mathcal{G}||\mathcal{A}|$. These two value functions are combined into a vector of length $2|\mathcal{G}||\mathcal{A}|$ that is fed to each network. The output of these networks is then a vector of length $|\mathcal{G}||\mathcal{A}|$, representing the single composed-task world value function $\hat{Q}_{1\wedge 2}^*$ at state s. If f is a conjunction operator, we have the mapping $f : \mathcal{G} \times \mathcal{A} \to \mathcal{G} \times \mathcal{A}$. We produce the complete composed-task world value function by iterating over all states.

This setup, however, can lead to highly convoluted networks, which grow even more complex as the size of the goal and action spaces increases. Figure 3.2 shows a network where optimal base-task value functions are used in a tabular environment containing 4 goals and 5 actions. Even in this relatively simple case, we produce strongly connected networks which have exploded in size and possess an infeasible number of connections.

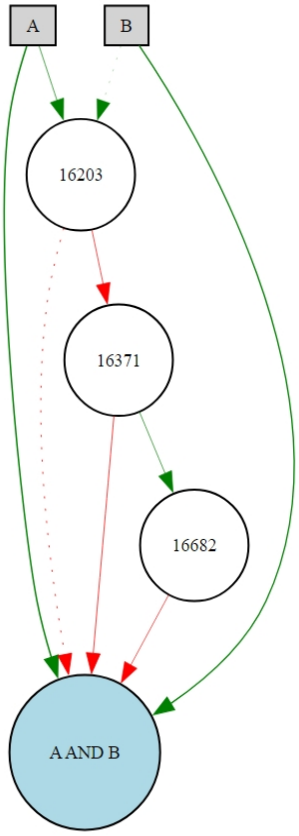


FIGURE 3.3: Simple neural network produced with NEAT. Acts as alternative to *min* operator when performing conjunction. Node numbers are an identifying marker used to keep track of innovations.

In the initial approach, networks take a high level view by processing a multi-dimensional slice of the input value functions. In contrast, fixed operators work on a pointwise basis—e.g., we would apply $\hat{Q}^*_{1\wedge 2}(s,g,a) = \min\{\hat{Q}^*_1(s,g,a), \hat{Q}^*_2(s,g,a)\}$ to every triplet $(s,g,a) \in \mathcal{S} \times \mathcal{G} \times \mathcal{A}$. To replicate this flow, we let the conjunction operator f be a mapping $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Forcing our networks to follow this structure results in much simpler operators, as shown in Figure 3.3, and generally superior performance. The algorithm uses the same information as previously, but divided into smaller pieces. We note, however, that this change means networks are given less context—which may not be suited to certain tasks. There are potential middle-grounds between these two approaches—for example, partitioning value functions into manageable chunks. Instead of feeding networks a vector of

size $2|\mathcal{G}||\mathcal{A}|$, we can define them to accept vectors of size $\frac{2|\mathcal{G}||\mathcal{A}|}{2}$ or $\frac{2|\mathcal{G}||\mathcal{A}|}{4}$ etc (depending on the exact specification of $\mathcal{G}$ and $\mathcal{A}$). In this work we use the per-value setup, as it most closely mimics the existing Boolean algebra structure from which we draw.

3.5 Fitness

3.5.1 Fitness Measures

NEAT is underpinned by a fitness function, through which we compare solutions. The aim is to generate solutions that achieve the best fitness. We want this fitness function to model the interaction between the agent and its environment. A reasonable approach is to use the total return achieved by the agent over a number of evaluation episodes.

Alongside total return, *success rate* is a secondary fitness measure—we tally the number of times the agent reaches a desired composed-task goal across all evaluation episodes. This approach is most effective when there is a disconnect between our reward function and the behaviour we desire from the agent—see Section 3.1. However, there is a clear limitation. Successful networks are incentivised, but 'failed' networks are crudely grouped together and their positive attributes discarded. We might have a network that causes the agent to act intelligently but simply gets stuck at one state, yet success rate does not sufficiently distinguish between this and a network which causes the agent to act randomly. As such, we default to return as our primary fitness measure.

3.5.2 Fitness Evaluation

We aim to learn operators that are effective over all possible instances of a desired composition. Consider, for example, the case of learning a conjunction operator in an environment containing four goals. If the goals are labelled G_i, there are twelve possible conjunctions between pairwise disjunctions: $(G_1$ `OR` $G_2)$ `AND` $(G_1$ `OR` $G_3)$, $(G_1$ `OR` $G_2)$ `AND` $(G_2$ `OR` $G_3)$, etc. For each evaluation episode the agent is randomly assigned a starting state and one of these possible compositions, giving the

resultant operators a broader understanding of the problem domain rather than being tuned to any one composed-task. This also adds a degree of flexibility, allowing for transition to unseen problem domains.

Given a desired composition, we fetch the relevant base-task value functions and feed a representation of these to networks in the population. Using this input, each network produces a representation of the composed-task value function. The agent uses this new value function to determine a policy, and we record the total return achieved by the agent when following this policy across all evaluation episodes. This process is repeated for each generation.

There is of course a trade-off between performance and efficiency when choosing the number of evaluation episodes. Without any limit on time or computing power, we could use an arbitrarily large number of episodes—mitigating the impact of randomness and boosting network performance. In practice, however, this becomes computationally strenuous and time-consuming. Owing to these practical concerns, but still wanting to provide our networks with sufficient information, we default to a number of evaluation episodes in the region between 10 and 50.

3.6 Differentiable Approximations

While emphasis is on NEAT, we also consider a middle ground between these learned functions and the Boolean set of fixed operators. The operators for optimal disjunction and conjunction are respectively the *max* and *min* functions. It is reasonable to hypothesise that, even when these operators are insufficient, a function of similar form might achieve superior results. To this end, we examine three differentiable functions that approximate the *min* and *max*.

Consider an environment with state space $\mathcal{S}$, goal space $\mathcal{G} \subseteq \mathcal{S}$, and action space $\mathcal{A}$. Let Q_1 and Q_2 be world value functions in this environment. For a given triplet $(s, g, a) \in \mathcal{S} \times \mathcal{G} \times \mathcal{A}$, let $q_1 = Q_1(s, g, a)$ and $q_2 = Q_2(s, g, a)$. Then,

- $Boltzmann(q_1, q_2) = \dfrac{q_1 \cdot e^{\alpha \cdot q_1} + q_2 \cdot e^{\alpha \cdot q_2}}{e^{\alpha \cdot q_1} + e^{\alpha \cdot q_2}}$

- $LogSumExp(q_1, q_2) = \frac{1}{\alpha} \log\left(e^{\alpha \cdot q_1} + e^{\alpha \cdot q_2}\right)$
- $MellowMax(q_1, q_2) = \frac{1}{\alpha} \log\left(\frac{e^{\alpha \cdot q_1}}{2} + \frac{e^{\alpha \cdot q_2}}{2}\right)$

If *approx* signifies any of these functions, then $min(q_1, q_2) < approx(q_1, q_2) < max(q_1, q_2)$, as well as $\lim_{\alpha \to -\infty} approx(q_1, q_2) = min(q_1, q_2)$ and $\lim_{\alpha \to \infty} approx(q_1, q_2) = max(q_1, q_2)$

A composed value function, $Q_{composed}$, is built up by applying the chosen function to every corresponding pair of values from Q_1 and Q_2:

For all $(s, g, a) \in \mathcal{S} \times \mathcal{G} \times \mathcal{A}$,

$Q_{composed}(s, g, a) = approx(Q_1(s, g, a), Q_2(s, g, a)) = approx(q_1, q_2)$

The single parameter α can be optimised with any chosen method—for example, iterating over possible α values in the range $[-20, 20]$ was observed empirically to provide satisfactory results. α values either side of these bounds produced output that was indistinguishable from using the true *min* and *max* operations. The intractable case where $\alpha = 0$ can be handled by simply incrementing over the modified set $[-20, 20] \setminus \{0\}$, or through an appropriate choice of step-size. Alternatively, α can be optimised using a standard genetic algorithm. As with NEAT, the chief fitness measure used is total return over a set of evaluation episodes. We aim to find the parameter α which performs best across all combinations $(s, g, a) \in \mathcal{S} \times \mathcal{G} \times \mathcal{A}$.

3.7 Concluding Remarks

Through the first part of this chapter, we sketch out limitations of the existing Boolean algebra framework. In particular, we identify three scenarios where the Boolean algebra is insufficient. These scenarios are maintained throughout the study as motivating cases. Given this more detailed treatment of our research problem, we construct an optimisation scenario and describe how NEAT is adapted to

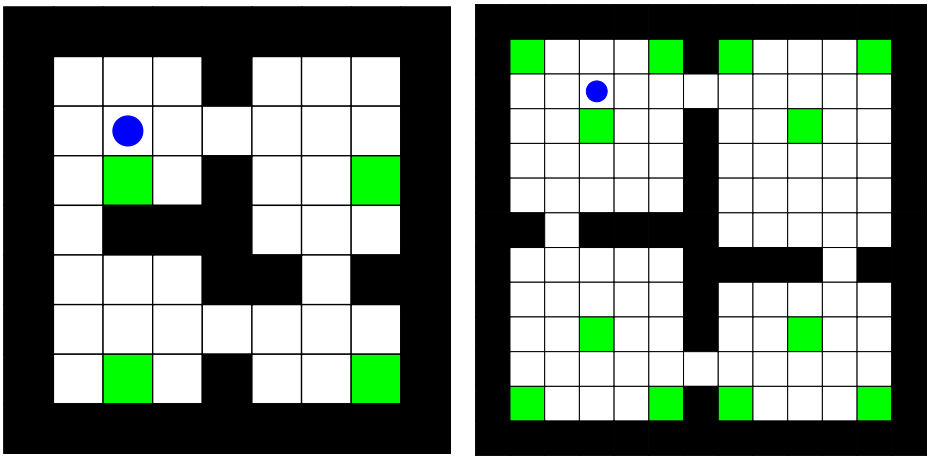
learn a new set of composition operators. We also delve into two important considerations associated with usage of NEAT—the structure taken by resultant networks, and the fitness method used to compare performance of these networks. The methodology concludes by presenting an alternative approach both to NEAT and the original Boolean algebra—using differentiable functions which approximate the *min* and *max* operators. In the following chapter, we evaluate these methods with a series of experiments conducted in a tabular reinforcement learning domain.

Chapter 4

Tabular Environments

In this chapter, we explore the performance of learned operators within the tabular Four Rooms environment. Initial tests are conducted with standard suboptimal value functions. From this base we expand to analyse operator performance in the face of a larger state space, when subject to stochastic agent transitions, and when dealing with a non-viable reward function.

4.1 Environment



(A) 9 × 9 Four Rooms with 4 goals (B) 13 × 13 Four Rooms with 12 goals.

FIGURE 4.1: Green squares indicate potential goals, while the agent is represented by the blue circle.

We conduct experiments in the Four Rooms domain [20], with Figure 4.1 depicting two variations. An agent can move in the four cardinal directions and has a fifth

'stay' action that allows it to remain in its current state. Attempting to enter a wall also causes the agent to remain stationary. Goal states are terminal should the agent choose to stay in one, and otherwise can be travelled through. Otherwise, termination of an episode is triggered by reaching a limit of 15 steps in the 9×9 case and 25 steps in the 13×13 case. In this setting, we run NEAT for 3000 generations, with 25 evaluation episodes and an initial population size of 300 members. These numbers are reached empirically, and were observed to represent a balance between network performance and the time taken for a NEAT run.

4.2 Learning Base Tasks

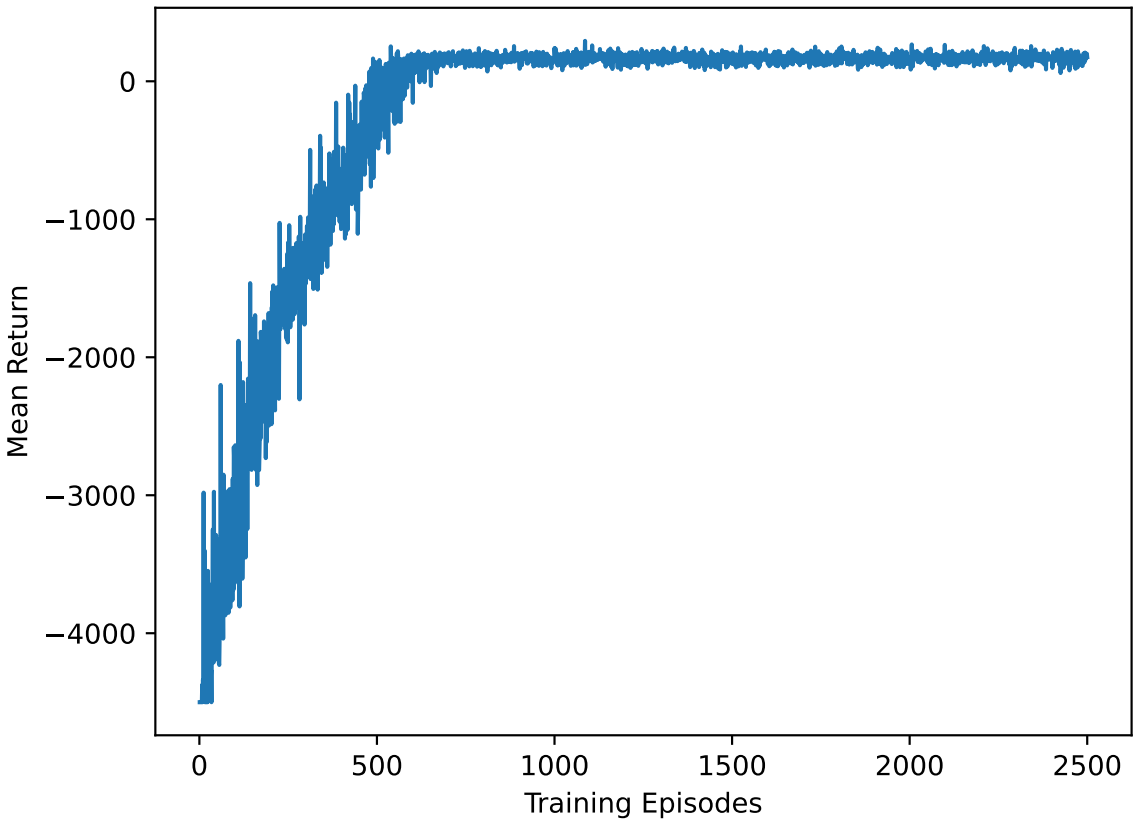


FIGURE 4.2: Number of Training Episodes vs. Mean Return for Q-Learning of a single base task. Environment considered is 9x9 4-Goal Four Rooms with deterministic transitions.

Base tasks in the Four Rooms setting are pairwise disjunctions between goals. For example, the base tasks $(G_1 \texttt{ OR } G_2)$ and $(G_2 \texttt{ OR } G_3)$ can be used to define the composed task $(G_1 \texttt{ OR } G_2) \texttt{ AND } (G_2 \texttt{ OR } G_3) = G_2$. Value functions for the base tasks are obtained with Q-Learning [7].

We monitor convergence during training, allowing us to learn over a varying number of episodes and to generate value functions of a desired optimality level. Figure 4.2 shows how the number of episodes used to learn a value function relates to the return generated by an agent using that value function. We see a steady ascent, before a prolonged plateau around the optimal return. Fully converged value functions are treated as optimal. If reaching convergence takes N training episodes, value functions trained over $n \leq N$ episodes are estimated to be $\frac{100*n}{N}\%$ optimal.

4.3 Fitness Evolution

Figure 4.3 shows the evolution of fitness during a NEAT run in the 9×9 Four Rooms setting with deterministic transitions. We initialise a population of 150 networks, and run NEAT over 1500 generations. Optimal value functions are used, as this plot acts as a general showcase of NEAT's evolutionary process. In order to produce a versatile conjunction operator, we evaluate over all possible AND tasks in the environment. The fitness metric employed is total return over a chosen number of evaluation episodes—in this case 15. More discussion on the role of fitness can be found in Section 3.5.

The blue and red plots, respectively, show the mean and maximum fitness during each generation. Initially, networks lack useful knowledge of the environment and perform poorly, before being honed through the evolutionary process. Eventually a plateau is reached where the best networks stop showing large improvement. From this point on, NEAT acts similarly to a standard search algorithm—combing through different combinations in order to find good solutions. We see that even after the best networks in each generation have stopped making jumps in performance, the average population fitness still shows gradual improvement. Within a run of NEAT, networks are evaluated over a small number of episodes and as such

the fitness of a single network is highly sensitive to randomness in initial conditions of these episodes (the starting state where an agent spawns and the desired composition it has been assigned). This means that fitness is not entirely representative of a network's 'true' performance when tested over a much larger set of test episodes (typically 10 000). A population of higher average fitness indicates a more stable evolutionary process and provides us with a larger set of suitable candidate networks for the testing phase. There is still the possibility of high-performing networks manifesting from poorly performing populations, and we use observed results to push NEAT hyperparameters towards a configuration that balances stability with exploration.

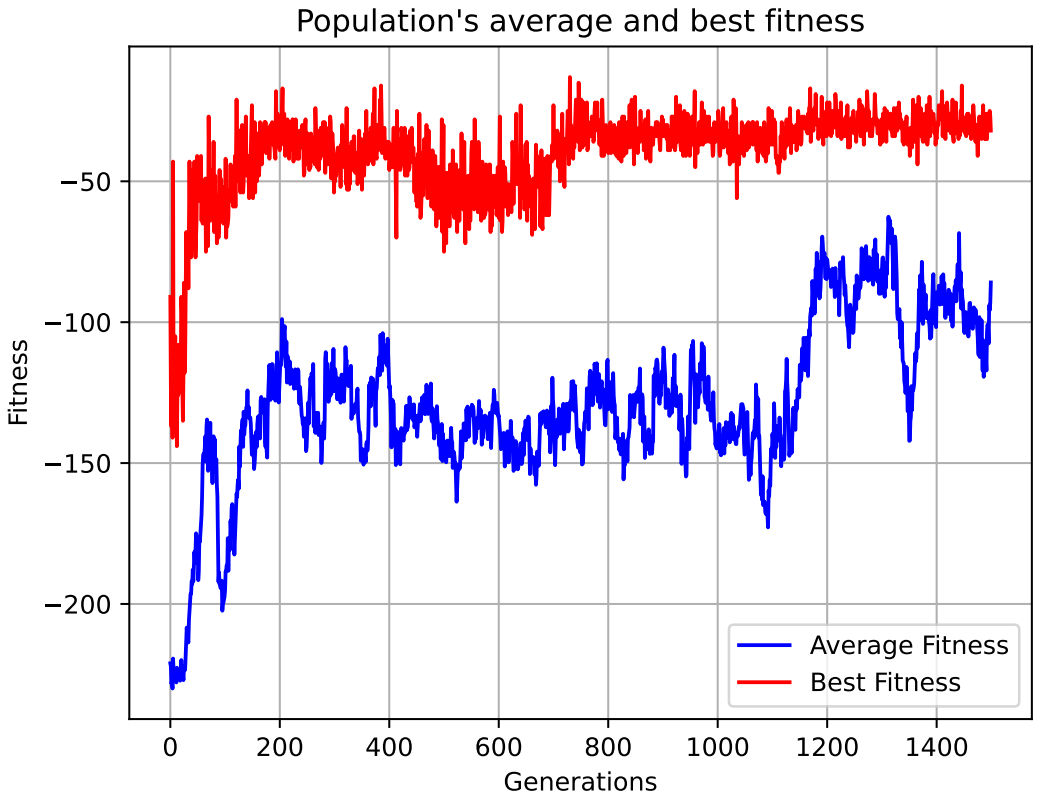


FIGURE 4.3: Learning with NEAT. Fitness measure is total return. Best performing networks after each generation are shown in red, while blue depicts the population average.

4.4 Evaluation of Results

Across the experiments presented in this report, we store the 10 highest-fitness operators observed at any point during a run of NEAT. Additionally, we store the highest-fitness population member at the end of the final generation. We then generate a list of random agent configurations (starting point and desired composition) over which we track performance when using each of the learned operators to compose value functions. This performance is compared to use of the relevant fixed operator. The performance metric corresponds to the fitness used to learn the operators—predominantly total return over a set of test episodes, alternatively success rate achieved over these test episodes.

95% confidence intervals are used to construct error bars. In the case of learned operators, as far as possible, data points for these intervals will be the best-performing network returned by a self-contained run of NEAT. This is our primary approach for generating data and each experiment is accompanied by multiple full runs of NEAT. We also supplement this data with additional networks returned by the same run, or with networks returned by the same run but tested with a re-learned set of value functions. The high degree of exploration in the learning process leads to variability among the resultant operators. In the case of fixed operators, data points are generated by new agent configurations and re-trained base-task value functions of equivalent optimality.

These data-points are used to perform significance tests. First, for each set of data-points, we run the Shapiro-Wilk test to evaluate its normality. A sample is considered sufficiently Gaussian if this test returns a p-value > 0.05. If both samples being compared abide by this condition, we proceed to a 2-sample t-test. For each learned operator and differentiable approximation, we set up the null hypothesis that performance is equal to that of the Boolean operator, and the alternative hypothesis that performance is superior. We use the convention that p-values ≤ 0.05 indicate varying degrees of significance, while p-values > 0.05 indicate that the result is not statistically significant. If at least one of the compared samples is not sufficiently normal according to the Shapiro-Wilk test, we use the Mann-Whitney U test as a non-parametric alternative to the t-test. Hypotheses and p-value interpretations

remain the same between both test methods.

4.5 Comparison to Known Functions

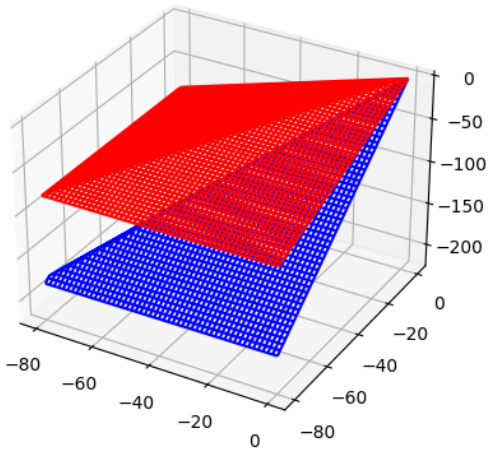


FIGURE 4.4: Plot of learned conjunction operator (blue) vs *min* (red) for optimal value functions.

It is natural to consider the form taken by these operators. Can they be described in terms of existing functions? Consider the case of optimal base-task value functions. Figure 4.4 shows the output of a learned conjunction operator against that of the *min*. There is overlap in parts of the space, but to gain more insight we use a genetic programming approach to perform symbolic regression on the learned operator. Code for the symbolic regression is implemented by the gplearn library [27]. With a high degree of accuracy, the operator is fit to the following function: $f(x, y) = \min(2.274y, 2.771x - 0.565y)$. Clearly, there is a strong relationship between the learned and fixed operators.

4.6 Composition of Suboptimal Value Functions

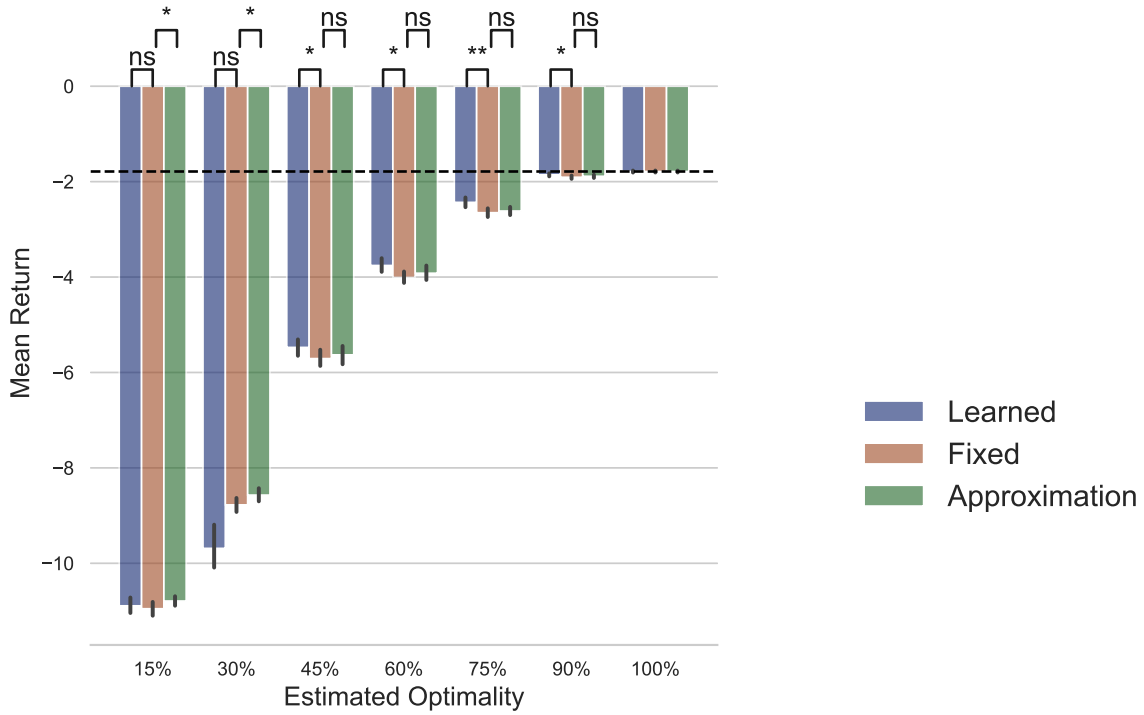


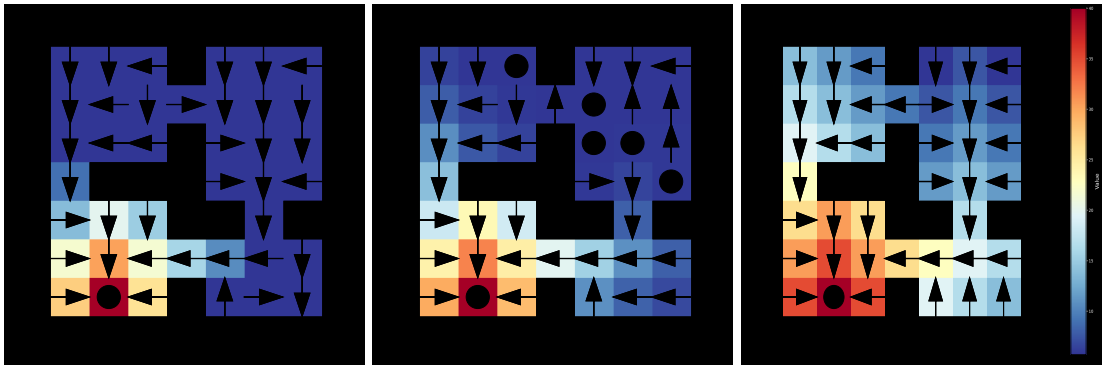
FIGURE 4.5: AND performance for 3 operator-types and value functions of increasing optimality. Results taken over 10 runs. Error bars represent a 95% confidence interval. The dotted horizontal line represents theoretically optimal performance. A significance marker of * indicates a p-value $\leq$ 0.05, while ** shows a p-value $\leq$ 0.01. ns indicates the alternative operator does not have a significant advantage over the fixed operator. Note that at 100% optimality, all operators achieve equal performance.

The simplest setting we consider is a 4-goal 9×9 Four Rooms with deterministic transitions. The reward function gives $+4$ for desired goals, -15 for undesired goals, and -1 for each step.

Figure 4.5 shows returns across value functions of increasing optimality when performing conjunction. We compare the fixed Boolean operator to the alternative operators learned with NEAT and to the differentiable approximations discussed in Section 3.6. Any of these approximations (Boltzmann, LogSumExp, MellowMax) would take the place of the *min* function (AND). When evaluating disjunction tasks

in the following experiments, we use the same approximations to replace the *max* function (OR). The choice of parameter α controls the output of these functions, and can be tuned to make them more suitable for approximating either the *min* or *max*. In the case of negation, we simply compare the fixed Boolean operators to those learned with NEAT—no equivalent to these approximations is used.

Learned operators have an advantage over their fixed equivalents when estimated optimality is in the range of 45% to 90%. Intuitively, this covers value functions with enough knowledge of the base tasks to produce intelligent behaviour, but which still lack the information required for acting optimally. We also see that our method matches fixed operator performance in the case of optimal value functions. Differentiable approximations achieve superior performance to both alternatives in the case of 15% and 30% optimality, while maintaining competitive results at subsequent levels.



(A) Learned Operator (B) Fixed Operator (C) Optimal

FIGURE 4.6: Policies for the conjunction task $(G_1$ `OR` $G_2)$ `AND` $(G_2$ `OR` $G_4) = G_2$. Arrows indicate direction of movement, while circles indicate the agent chooses to stay in its current state. These actions are derived by using the composed world value function to act greedily. Colours are assigned according to the value at each state. a and b each produce these policies for value functions which are approximately 60% optimal. c shows the policy produced when value functions are optimal

Figure 4.6 shows how a resultant policy may differ depending on the operator used. We consider the specific conjunction task $(G_1$ `OR` $G_2)$ `AND` $(G_2$ `OR` $G_4) = G_2$, in the bottom left corner of the grid. Value functions are estimated to have 60% optimality. The first plot is produced with a learned conjunction operator. The agent gets stuck

at 3 states in the bottom right. From all other states, it successfully reaches the goal. In the next plot, produced with the fixed Boolean *min* operator, the agent gets stuck at 12 different states. Evidently, we achieve tangible benefit when using a learned composition operator. Finally, we visualise the policy produced when value functions are optimal. The agent does not get stuck at any state and we see a smoother distribution of values.

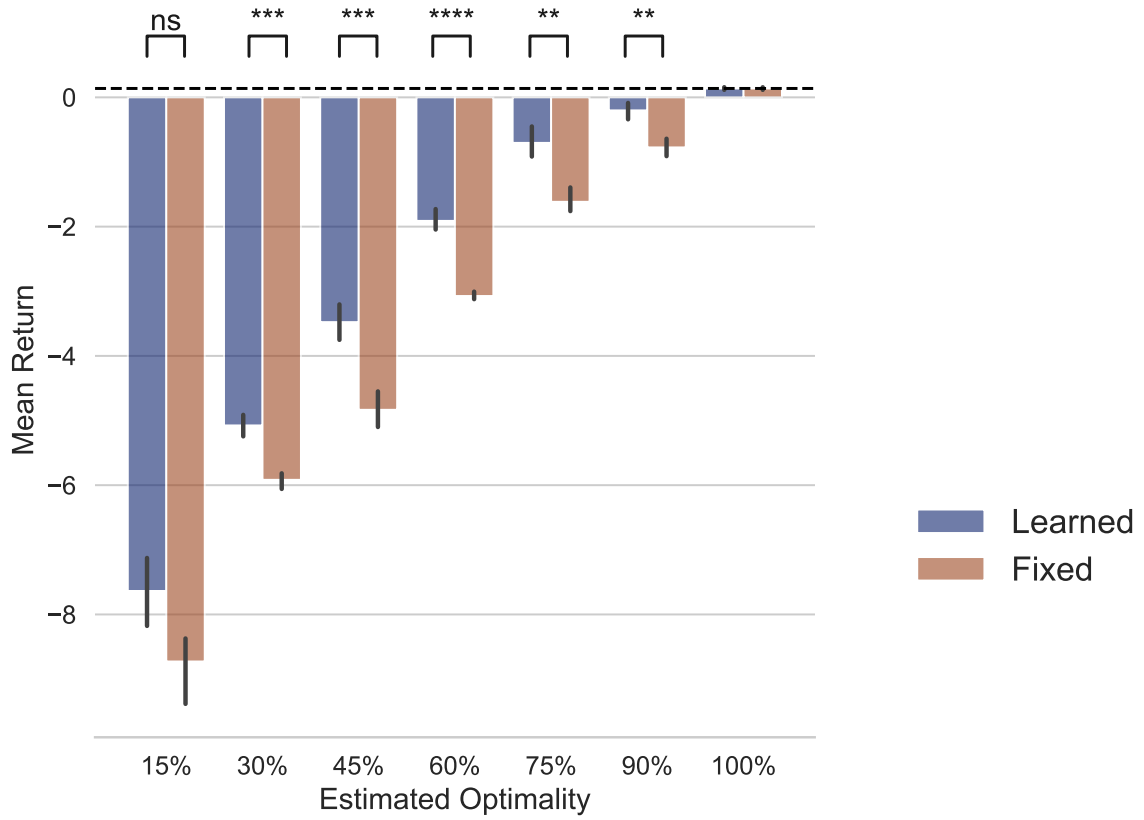


FIGURE 4.7: NOT performance for 2 operator-types and value functions of increasing optimality. Results taken over 5 runs. Error bars represent 95% confidence interval. The dotted horizontal line represents theoretically optimal performance. A significance marker of * indicates a p-value ≤ 0.05, ** is a p-value ≤ 0.01, *** is a p-value ≤ 0.001, and **** is a p-value ≤ 0.0001

Figure 4.7 shows mean returns in the case of negation. We see greater advantages across the spectrum of optimality. Both the Boolean negation operator and our networks take in three inputs: from the relevant value function and from the global min/max value functions—see section 3.2. Our method scales more effectively to

the extra complexity induced by the ternary negation operation as compared to the binary conjunction operation.

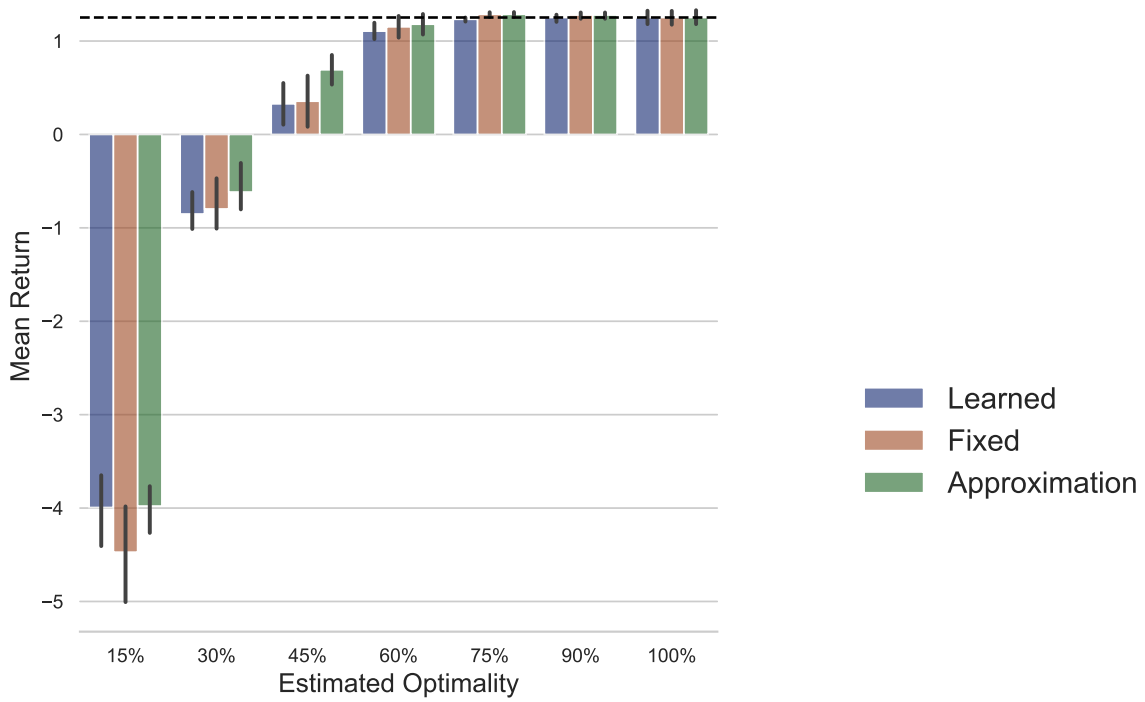


FIGURE 4.8: OR performance for 3 operator-types and value functions of increasing optimality. Results taken over 5 runs. Error bars represent 95% confidence interval. The dotted horizontal line represents theoretically optimal performance

In Figure 4.8, we see that disjunction results differ from the other operations. Learned operators only have an advantage over fixed operators for value functions of low optimality, with both being outdone by the approximations. Given more optimal value functions, performance is competitive with the *max* but does not exceed it. None of these performance gaps are significant. A potential reason for the differing results is disjunction having a high margin of error in small environments. For example, in this 4-goal setting, the composition $(G_1\ OR\ G_2)\ OR\ (G_3\ OR\ G_4)$ is achieved by merely staying in any goal state. Our learned neural network operators are geared to provide adaptable behaviour in cases where we lack sufficient information. The simple modifications made to the *max* operator by our differentiable approximations prove more appropriate than learning neural networks.

4.7 Composition in Larger Domains

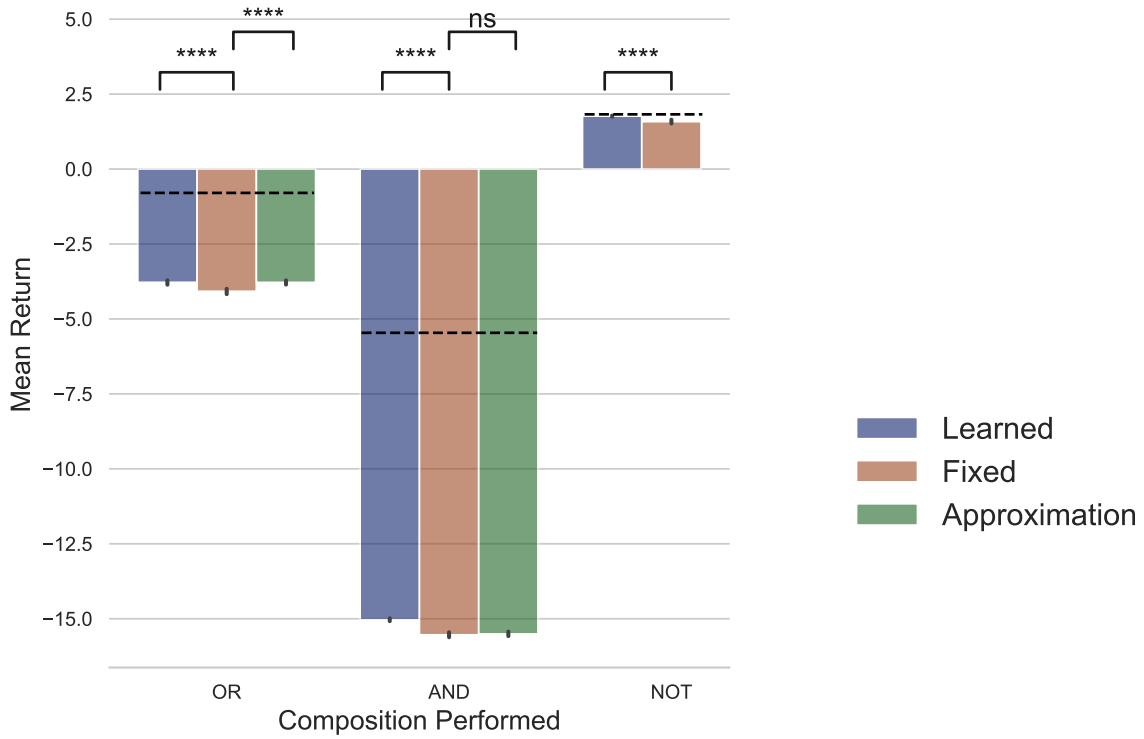


FIGURE 4.9: All 3 fundamental operations in the 13×13 Four Rooms domain. Results taken over 10 runs. Error bars represent 95% confidence interval. Value functions of approximately 50% optimality are used. Horizontal lines represent theoretically optimal performance. A significance marker of **** indicates a p-value ≤ 0.0001, while ns indicates the advantage is not significant.

Next, we investigate whether learning composition operators leads to superior returns when the state and goal spaces of the environment are scaled up in size. Naturally, this also increases the number of base tasks and possible compositions between them. We use the 13×13 Four Rooms with 12 goals, and the same reward function as before. Value functions of approximately 50% optimality are used. This marker was chosen to represent a set of suboptimal value functions with mediocre knowledge of the environment, so focus can be placed on the impact of a larger environment.

Figure 4.9 shows the performance of learned operators and differentiable approximations against fixed operators in this setting. Previously, learned operators held a clear advantage when performing conjunction and negation. This advantage holds in the larger environment, while we now see learned disjunction operators outperforming the fixed *max* operator and achieving near identical performance to the approximation. When we move from 4 goals to 12 goals, the number of base tasks rises from 6 to 66 and the number of disjunction tasks rises from 13 to 1083. The Boolean algebra is less equipped to handle this increase in difficulty when conditions are suboptimal. In the case of conjunction, our learned operator is able to achieve a significant advantage, while performance of our differentiable approximation is competitive with the fixed operator.

4.8 Composition with Stochastic Transitions

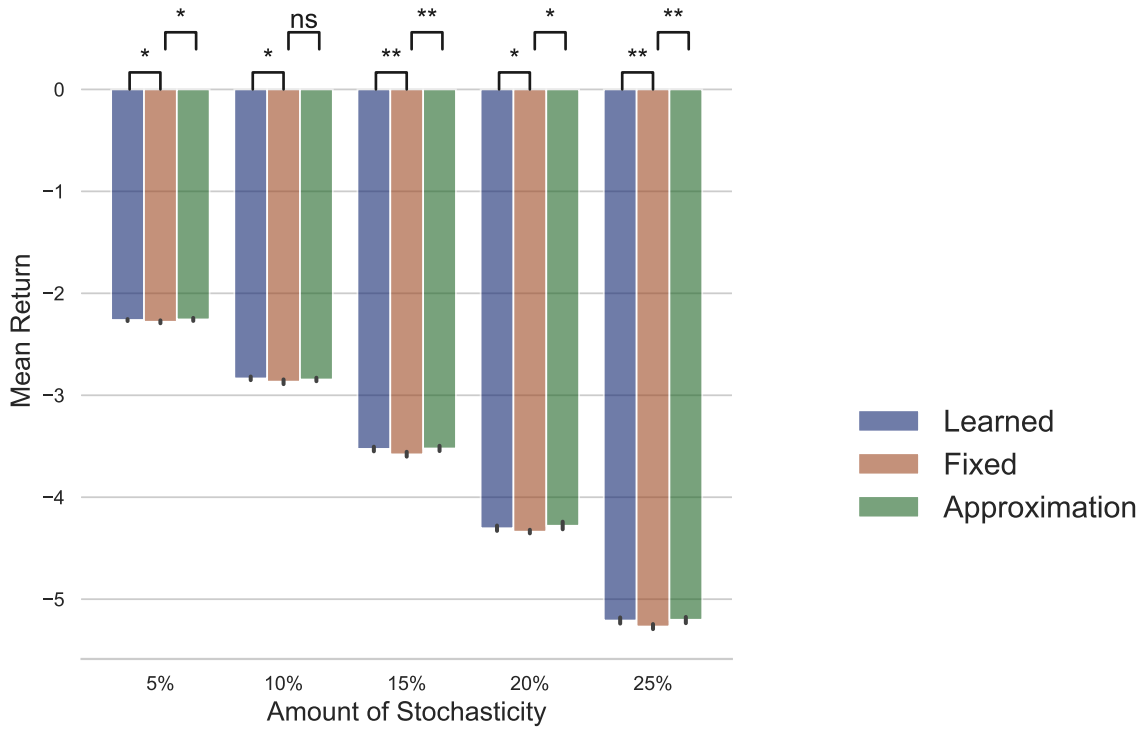


FIGURE 4.10: AND performance for optimal value functions and increasing levels of stochasticity in the 9×9 Four Rooms domain. Results taken over 10 runs. Error bars represent 95% confidence interval.

In this experiment, we investigate the applicability of our method in environments where agent transitions are stochastic. A stochasticity level of $p \in (0, 1)$ means an agent will move in its desired direction with probability $1 - p$, and has equal probability $p/3$ of moving in each of the remaining directions. The 'stay' action is not affected.

Base-task value functions are trained to convergence— although optimal, randomness in the environment leads to unpredictable agent behaviour. Figure 4.10 plots conjunction returns for learned operators and differentiable approximations against the fixed Boolean operators. Stochasticity levels are shown to increase from 5% to 25%. Past this threshold, behaviour deteriorates to the point where the performance of any operator is not informative. Learned operators achieve the greatest returns for all levels, with the approximations being competitive throughout.

4.9 Composition with Non-Viable Reward Functions

We move on to explore composition in the presence of a reward function which is only suitable for learning base tasks—see Section 3.1. Previously, we had used return as our fitness measure. Agents would be driven to act intelligently across the environment, and in the process would achieve composed tasks. However there are faulty reward functions structured such that pursuing composed goals does not maximise return, even with optimal base-task value functions. In this case, we employ success rate as an alternative fitness function for directly promoting the pursuit of composed goals. During each evaluation episode, reaching the correct goal is regarded as a success and +1 is added to the fitness. Any other outcome—be it reaching an incorrect goal or getting stuck—is a failure and adds 0 to the fitness. NEAT is used to produce networks that achieve the greatest number of successful compositions across the evaluation episodes.

We consider the following reward function: 0.1 is given for desirable goals, -0.1 is given for each step, and -0.15 is given for undesirable goals. This reward is sufficient to learn each of the base tasks in our 9×9 Four Rooms. However, we cannot reliably apply the Boolean conjunction and negation operators in this setup.

Figure 4.11 compares our operators learned with NEAT both to the differentiable approximations and to the fixed Boolean operators in terms of mean success rate. In this experiment, optimal value functions are used. With disjunction, we see identical performance between all three operator-types—each having a perfect success rate. Even with a problematic reward function, the relatively easy task of disjunction in a small few-goal environment is achievable without disruption by all sets of operators. The impact of a non-viable reward is observed in the cases of conjunction and negation. Learned operators cope better with this obstacle, and having been tuned directly with success rate are able to complete the composed tasks more frequently. The differentiable approximation for conjunction also has an advantage over the fixed operator, though does not match the learned operator.

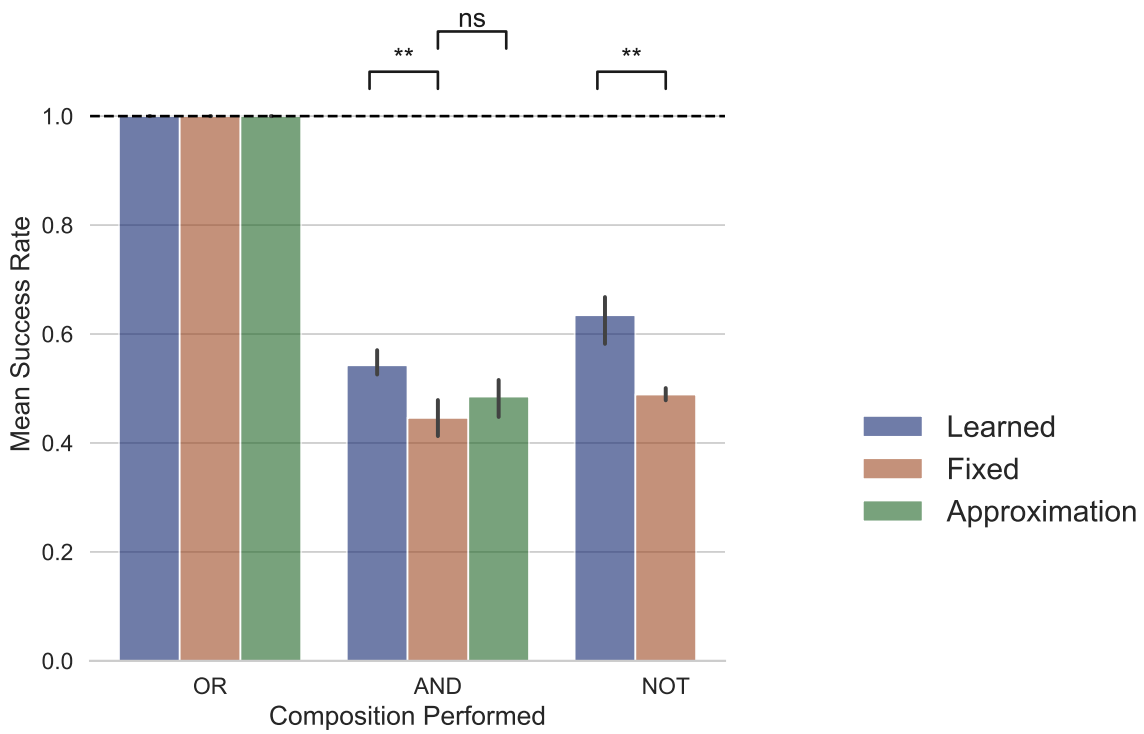


FIGURE 4.11: Mean success rate for all compositions, using optimal value functions and non-viable reward in 4-goal 9×9 Four Rooms. Horizontal line shows perfect success rate. Results taken over 5 runs. Error bars represent 95% confidence interval. A significance marker of ** indicates a p-value ≤ 0.01, *** indicates a p-value ≤ 0.001, and ns indicates not significant.

4.10 Concluding Remarks

Previously we have introduced necessary context surrounding this study, and proposed a technique to learn robust composition operators with NEAT. Here, we demonstrate the viability of our method in the tabular setting. We start in a 9×9 Four Rooms with standard reward function, where transition dynamics are fully deterministic and we possess a suboptimal set of value functions. Modifications to this set-up are made independently by a) enlarging the environment, b) incorporating stochastic transition dynamics, and c) using a reward function that inhibits Boolean composition. Across these experiments, we observe that learned operators possess an advantage over fixed Boolean operators when dealing with common sources of uncertainty found in tabular reinforcement learning problems. To build on these results, we use the following chapter to evaluate the case when our environment requires function approximation techniques.

Chapter 5

Function Approximation

Our experiments have shown the utility of learned composition operators in easily representable tabular environments. However, much of the work being done in RL concerns high dimensional state or action spaces that can only be practically solved using function approximation techniques. We move on to evaluating the performance of learned operators within this type of framework.

5.1 Environment

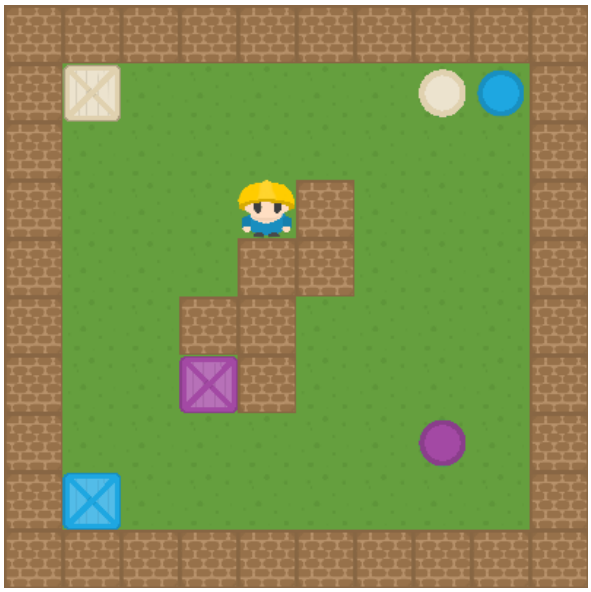


FIGURE 5.1: The Boxman function approximation environment [1]

Function approximation experiments are conducted in an environment called Boxman [1], where the state space is described a higher-dimensional vector. An agent

traverses around obstacles to collect desired objects. There are six total objects—beige circle, blue circle, purple circle, beige square, blue square, purple square. Clearly each object is a colour-shape combination. Base tasks are of the form "collect a blue object", "collect a circular object". Agents can move in the 4 cardinal directions. Additionally, after reaching one of the objects they can choose to collect it. Episodes terminate either when the agent has collected an object, or when it has reached a limit of 12 steps. States are represented as 84×84 RGB images.

A reward of $+2$ is given for collecting the desired goal object, while the reward for taking a step and collecting an undesired goal object is -0.1.

5.2 Learning Base Tasks

It is not feasible to apply Q-Learning in this high-dimensional environment. Instead, we use Deep Q-Learning [8] to generate value functions in the form of Deep Q-Networks (DQNs). Experiments are conducted with a set of value functions estimated to be 50% optimal. These were obtained by training base tasks to the point of optimal performance, observing the number of episodes required for this, and then re-training a set of value functions over half that number of episodes. 50% is chosen to represent a generically suboptimal set of value functions, with the focus of experiments being how our method scales up to this function approximation environment. Future work should assess the viability of our method across the spectrum of optimality in this function approximation setting, as we had previously tested in the tabular Four Rooms setting.

5.3 Learned Composition with Function Approximation

Running NEAT in this higher-dimensional setting is significantly more compute-intensive than in the previous tabular setting. Thus, for feasibility, we scale down the run parameters. A run is set to last for 800 generations, with an initial population size of 80. Fitness is determined from a set of 15 evaluation episodes. These numbers were reached empirically, based on observing the time taken to complete a run. Increasing either the population size or number of generations far beyond

this point would lead to prohibitive training lengths—several days or even weeks when running on a 14-core Intel Core i9 processor, a 24GB NVIDIA RTX 3090 GPU, and 128GB of RAM.

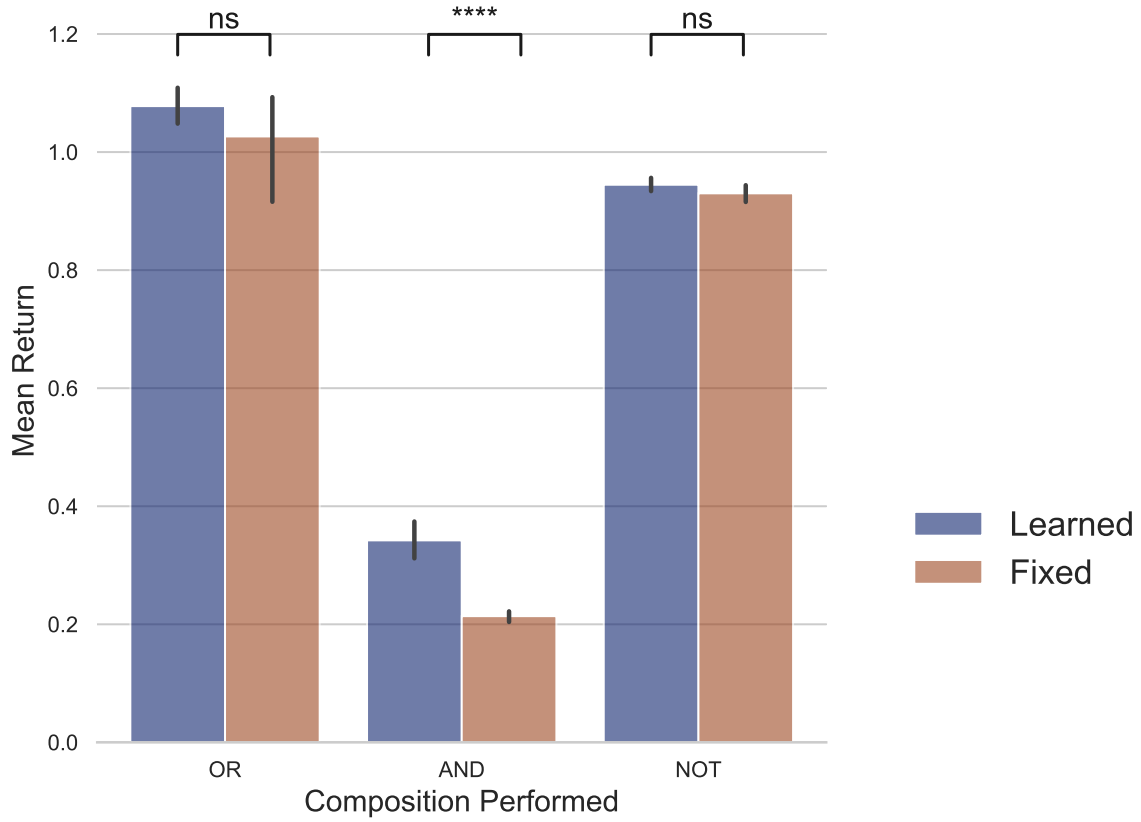


FIGURE 5.2: All three fundamental compositions in Boxman domain. Intermediate value functions trained for 1 million steps are used. Results aggregated over 10 operators produced over three NEAT runs. Error bars represent 95% confidence interval. A significance marker of **** indicates a p-value ≤ 0.0001, while ns indicates the gap is not statistically significant.

Figure 5.2 shows performance in Boxman under this configuration, for all three logical operations. In each case, we see learned operators with an advantage. The greatest gap occurs for conjunction. We again emphasise that these results are subject to the practical restrictions imposed on the experiment. With either the addition of greater extensive computational resources or the subtraction of time concerns, a

full-scale NEAT run should produce more sophisticated operators equipped to navigating in this environment. Current results thus serve to support the viability of our method in large function approximation settings.

5.4 Transfer Between Domains

We have shown separately that learned composition operators can achieve positive performance in both smaller tabular settings and in more complex function approximation settings. Ideally, however, we want flexible operators able to transfer between domains of different state spaces. In particular, we want to learn operators in a tabular setting and apply them in a function approximation setting. Recall from Section 3.4 that operators learned with NEAT are pointwise mappings over $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. This means they are independent of state space specification, and can be applied across environments.

Harnessing existing knowledge in new and more complex domains is an important step for the generalising skills of an RL agent, while also allowing for more efficient resource usage. We consider two cases: i) learning operators in the source domain before zero-shot transferring to the target domain, and ii) learning operators in the source domain before fine-tuning them in the target domain. In this study, Four Rooms acts as our smaller source domain and Boxman is the more complex target domain.

Figure 5.3 compares four approaches when testing conjunction (AND) in the Boxman domain:

1. Learn AND operator fully in Four Rooms.
2. Learn AND operator fully in Boxman.
3. Learn AND operator in Four Rooms, then fine-tune it in Boxman with a scaled-down NEAT run (fewer generations/population members/evaluation episodes).
4. Use the *min* function as AND operator, as defined in the optimal Boolean algebra framework.

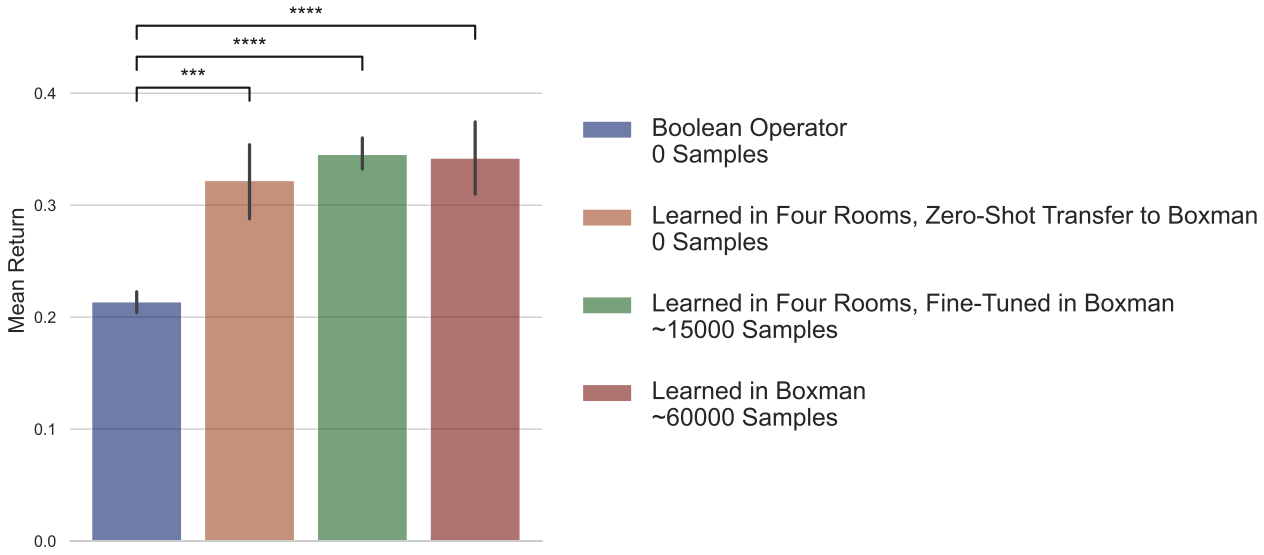


FIGURE 5.3: Mean Return for operators of different sources when performing conjunction. Number of Boxman environment samples needed to produce each operator are indicated. Results taken over 10 runs in each setup. Error bars represent 95% confidence interval. A significance marker of *** indicates a p-value ≤ 0.001, while **** indicates a p-value ≤ 0.0001

The number of Boxman environment samples observed during each case acts as an efficiency measure. The Boolean algebra and zero-shot transfer of a tabular operator require 0 Boxman samples.

All uses of NEAT outperform the *min* conjunction operator. There are two important conclusions that can be drawn from this. Firstly, the zero-shot transfer of operators learned in a tabular domain with differing state space leads to significantly greater returns than the Boolean operator framework. Secondly, minimal fine-tuning of these operators produces roughly equal return to a NEAT run undertaken solely in Boxman, while requiring around $\frac{1}{4}$ as many environment samples.

5.5 Concluding Remarks

The focus of this chapter is the transition from our tabular Four Rooms setting to the higher-dimensional Boxman environment. This means adapting how we learn

our base tasks (now with Deep Q-Learning), but also leads to computationally expensive NEAT runs and requires agents to handle more complex information. We see that our method is viable in this function approximation setting, with learned operators able to achieve competitive performance even subject to computational restrictions on our NEAT runs. However, the most promising result here lies in the ability to learn networks in our previous tabular setting before applying them to the Boxman environment—either by taking these networks as is, or by fine-tuning them in Boxman. Not only does this ease the computational burden of running NEAT in a function approximation environment, it presents an interesting direction in the broader pursuit of fluid knowledge transfer to new domains.

Chapter 6

Conclusion

6.1 Limitations

The chief obstacle to our method comes from sample inefficiency. In fact, sample inefficiency is a known drawback of evolutionary methods when compared to standard RL—resulting in part from the highly sparse nature of evolutionary fitness as a feedback mechanism. Even when optimal Boolean composition does not work as desired, the most efficient solution might be to forego composition altogether and learn the desired tasks individually. In the case of value functions which have not converged, one could just train them for longer. While this point is valid, it does not undermine this work. We may lack access to the original training context or value functions—perhaps we inherited this from an external party. Even with access, modifying these value functions might produce new complications—for example, catastrophic forgetting. There is also the issue of scaling. For a small number of possible compositions, it may be preferable to learn individual value functions. However, as the number of compositions increases, this is not necessarily feasible and having the ability to adapt to imperfect information holds greater importance. Similarly, we point to the flexibility of learned operators—efficiency might suffer for a single problem, but we regain some of this ground by transferring between problems or even across distinct domains. NEAT also holds the implementation advantage of being better equipped than value-based RL methods to evaluate solutions in parallel, helping us gain back some of the ground we lose in terms of sample inefficiency over a single evaluation.

Despite the capacity for superior performance, learned operators carry a degree of volatility. The strong theoretical grounding of the Boolean algebra means that

fixed operators achieve a stable performance baseline across many scenarios, even if we can improve on that baseline. With NEAT, however, the worst performing networks sometimes fail to be competitive.

Operators learned with NEAT are also less intuitive than the well-defined and easily explainable set of fixed operators—recall Section 4.5, where we use genetic programming to express NEAT-produced networks in terms of fixed operators.

As mentioned throughout the report, a run of NEAT is governed by a large set of hyperparameters. Choice of hyperparameters was guided by a combination of empirical observation and intuition—for example, it is clear that a greater population size allows us to explore a greater set of potential composition operators. However, we did not develop a clear correspondence between network performance and NEAT hyperparameters—in part because of the computational strain that would be induced repeated runs of NEAT. Results could be refined by selecting hyperparameters using a method such as F-Race [28]—where we trial various hyperparameter configurations and discard those that are found to cause statistically inferior performance when compared with at least one other candidate configuration, until we converge to the choice that produces the best expected performance.

Another point worth noting is the restrictive nature of the point-wise operators we learn. These perform well for clearly defined tasks—like the composition of standard value functions we consider in this work. However, handling more complex or adversarial scenarios might require us to learn a set of operators whose output carry broader context than a single action-value.

6.2 Final Remarks and Future Work

Composition of learned behaviours represents a growing area within reinforcement learning. It is desirable to manipulate knowledge already possessed by an agent in such a way that the agent gains new capabilities, without having to start the learning process from scratch. This not only aids computational and data efficiency but promotes the development of robust adaptable agents that bring us closer to

representing the way in which humans learn. Optimal Boolean composition operators (conjunction, disjunction, and negation) have been determined given that certain criteria are met—most importantly, we require optimal value functions for the tasks being composed. However, these criteria are not always met, and applying the optimal operators in such a case can lead to faulty results—such as an agent taking invalid actions or getting stuck in a loop. It is thus instructive to learn alternative operators which produce better results in cases where we do not have ideal conditions. We approach this problem primarily through adaptation of the NEAT algorithm.

Results support the viability of our method. We conduct a series of experiments in tabular settings, showing how learned operators are resilient to different sources of suboptimality within an agent's conditions and knowledge-base. Next we consider a larger function approximation environment. Our method is viable in this new setting, and, beyond this, we have the ability to transfer operators between the two domains. This brings us closer to a class of versatile agents that can use learned behaviours and adapt them to solve a variety of new tasks without additional learning.

However, that goal is great in both scope and ambition. There is potential to strengthen the ideas considered here, and to expand into related problems. We hope that future work can address sample efficiency concerns—relevant to cases where safety is a concern and the agent cannot repeatedly traverse its environment without higher stakes.

We also note that the environments considered here are similar in dynamics and agent behaviour. Though Four Rooms contains a tabular state representation and Boxman requires function approximation, both are simple 2D grid-based environments that the agent navigates to reach a desirable grid location. A natural next step would be to investigate the success of our method across a wider range of environments, including the more realistic domains that serve as benchmarks for testing novel RL methods. Such an investigation would also have implications for our experiment regarding transfer of operators between domains, and we would be able to gain a clearer understanding of the generalising abilities possessed by

learned composition operators if we test them across environments which differ in fundamental dynamics as well as state representation.

There is room for further investigation on the evolutionary side of our work. NEAT has lead to a number extensions on top of its core methodology. The most prominent of these is HyperNEAT [29]. Instead of directly evolving each node and connection between nodes, as NEAT does, HyperNEAT evolves an alternative Compositional Pattern Producing Network (CPPN). The CPPN is then queried for the weight of a specific connection. This compact representation allows for more efficient learning of large, logically structured networks. HyperNEAT is particularly applicable to geometric problems that require a sense of spatial awareness, as is the case with the grid-based domains considered in this work. NEAT has served as a strong baseline to exhibit the viability of neuroevolution for learning RL composition operators, but we can potentially boost performance by trialling other evolutionary techniques.

Bibliography

[1] Benjamin Van Niekerk et al. "Composing value functions in reinforcement learning". In: *International Conference on Machine Learning* (2019), pp. 6401–6409.

[2] Geraud Nangue Tasse, Steven James, and Benjamin Rosman. "A Boolean task algebra for reinforcement learning". In: *Advances in Neural Information Processing Systems* 34 (2020), pp. 9497–9507.

[3] Vanya Cohen et al. "Learning to follow language instructions with compositional policies". In: *Association for the Advancement of Artificial Intelligence* (2021).

[4] Geraud Nangue Tasse, Steven James, and Benjamin Rosman. "Generalisation in lifelong reinforcement learning through logical composition". In: *International Conference on Learning Representations* (2022).

[5] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[7] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[8] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[9] Geraud Nangue Tasse, Benjamin Rosman, and Steven James. "World Value Functions: Knowledge Representation for Learning and Planning". In: *Bridging the Gap Between AI Planning and Reinforcement Learning Workshop at ICAPS* (2022).

[10] Peter Auer, Thomas Jaksch, and Ronald Ortner. “Near-optimal regret bounds for reinforcement learning”. In: *Advances in neural information processing systems* 21 (2008).

[11] Geraud Nangue Tasse, Steven James, and Benjamin Rosman. “Composition and Zero-Shot Transfer with Lattice Structures in Reinforcement Learning”. In: *Journal of Artificial Intelligence Research* 82 (2025), pp. 2325–2388.

[12] Andries P Engelbrecht. *Computational Intelligence: An Introduction, Second Edition*. John Wiley & Sons, 2007.

[13] Anita Thengade and Rucha Dondal. “Genetic algorithm–survey paper”. In: *MPGI national multi conference*. Citeseer. 2012, pp. 7–8.

[14] Emanuel Todorov. “Linearly-solvable Markov decision problems”. In: *Advances in neural information processing systems* 19 (2006).

[15] Ching-An Cheng et al. “Rmpflow: A geometric framework for generation of multitask motion policies”. In: *IEEE Transactions on Automation Science and Engineering* 18.3 (2021), pp. 968–987.

[16] Anqi Li et al. “RMP2: A structured composable policy class for robot learning”. In: *arXiv preprint arXiv:2103.05922* (2021).

[17] Wenjie Qiu and He Zhu. “Programmatic reinforcement learning without oracles”. In: *The Tenth International Conference on Learning Representations*. 2022.

[18] Taewook Nam et al. “Skill-based meta-reinforcement learning”. In: *International Conference on Learning Representations* (2022).

[19] Andrew G Barto and Sridhar Mahadevan. “Recent advances in hierarchical reinforcement learning”. In: *Discrete event dynamic systems* 13.1-2 (2003), pp. 41–77.

[20] Richard S Sutton, Doina Precup, and Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[21] Martin Stolle and Doina Precup. “Learning options in reinforcement learning”. In: *Abstraction, Reformulation, and Approximation: 5th International Symposium, SARA 2002 Kananaskis, Alberta, Canada August 2–4, 2002 Proceedings 5*. Springer. 2002, pp. 212–223.

[22] Xue Bin Peng et al. “Mcp: Learning composable hierarchical control with multiplicative compositional policies”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[23] Avi Singh et al. “Parrot: Data-driven behavioral priors for reinforcement learning”. In: *arXiv preprint arXiv:2011.10024* (2020).

[24] Jorge A Mendez and Eric Eaton. “How to reuse and compose knowledge for a lifetime of tasks: A survey on continual learning and functional composition”. In: *Transactions on Machine Learning Research* (2022).

[25] Youngwoon Lee et al. “Composing complex skills by learning transition policies”. In: *International Conference on Learning Representations* (2019).

[26] Alan McIntyre et al. *neat-python*. 2017. URL: https://neat-python.readthedocs.io/en/latest/index.html.

[27] Trevor Stephens. *gplearn*. 2016. URL: https://gplearn.readthedocs.io/en/stable/.

[28] Mauro Birattari et al. “F-Race and iterated F-Race: An overview”. In: *Experimental methods for the analysis of optimization algorithms* (2010), pp. 311–336.

[29] Kenneth O Stanley, David B D’Ambrosio, and Jason Gauci. “A hypercube-based encoding for evolving large-scale neural networks”. In: *Artificial life* 15.2 (2009), pp. 185–212.