Programmatic agents and causal state abstractions

Larsen, Rasmus

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Ph.D. Thesis
Doctor of Philosophy

DTU Compute
Department of Applied Mathematics and Computer Science

Programmatic agents and causal state abstractions

Rasmus Larsen

Kongens Lyngby, Denmark 2021

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

When people think of artificial intelligence, they might imagine the kind of anthropomorphic robots found in science-fiction literature and films. These robots interact with the world in a way that comes naturally to humans, and like humans they have the ability to navigate through new situations by using knowledge and skills that they have previously acquired.

The methods that we use today for training artificial agents that interact with their environment lack several key features, which are needed to achieve the mentioned generalization ability. These features are composition, communication, and abstraction; in short, composition is to form a structure from smaller parts, communication is the ability to be understood by humans and other agents, and abstraction is the ability to look past details and see the bigger picture. When applied to the behavior of agents, these features allow for flexible behavior that combines previously learned skills to new situations, while also being able to tell others about the whys and hows of the behavior. A lot of this is reflected in the natural languages that humans use to communicate, and both these and languages used to write computer programs are structured with composition and communication in mind. The first contribution is a method for learning a representation of agent behavior encoded in a computer language, by imitating an existing policy which does not have a language representation. Since this language representation is composed of smaller parts, and can be read by people or machines, it is a step towards the features of composition and communication.

The work towards the third feature, abstraction, is based on the concept of causality. The basis of causality is straightforward: if event B happens because event A happened, then we say that A caused B. This concept is useful because it can inform us about whether something is important or not, either because it has an influence on something we care about, or because it does not. The second contribution is a method for learning a function that an artificial agent can use to group a bunch of distinct yet similar observations into a single state. For example, when opening the front door of a house, it might be useful for an agent to refer to having a key to the front door of the house simply as “having a key”, and not “having a key while cooking dinner in the kitchen”, or “having a key while cleaning the north-west corner of the living room” – essentially, the causality principle is used to infer what is important to represent, due to it having a causal impact on the goal of opening the door.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for acquiring a Ph.D. degree in Computer Science.

The research work presented in this thesis was supervised by Mikkel Nørgaard Schmidt and Lars Kai Hansen.

Kongens Lyngby, Denmark, August 9, 2021

Rasmus Larsen

Acknowledgements

To everyone who supported me throughout the years.

Contents

CHAPTER 1

Introduction

The holy grail of artificial intelligence is an agent that learns to behave intelligently in complex, varied settings. Such an agent would need to decompose learned behaviors and compose new ones, bringing what it has previously learned into new settings. This is necessary, unless every new environment and behavior must be learned from scratch. Additionally, a group of intelligent agents can benefit from communicating about commonly used behaviors. For example, a recipe is nice to have when learning to cook a meal. However, a recipe is quite abstract – it contains references to other behaviors that the reader must already know how to perform, such as picking up a spoon and filling it with sugar. Despite this difficulty, recipes manage to communicate novel behaviors to humans, who have previously acquired the necessary sub-behaviors.

Reinforcement learning is currently the go-to approach for learning behaviors in interactive settings. Through modern deep learning methods, the approach has been scaled to high-dimensional perceptual settings, where state observations are images. Policies, which are behavioral functions that map state observations to the actions that the agent takes, are implemented as deep neural networks. In the case of image observations, the neural network will probably have a convolutional structure – small filters that are repeatedly applied across the entire image, imbuing the network with an inductive bias towards visual structures such as edges and corners, at a low level, or entire complex objects, at a high level. The purpose of an inductive bias is to be informative about the structure of good solutions; images of the world have structural regularities such as edges, and human brains also perform edge detection in early stages of visual processing. Deep learning has indeed led to greatly improved approaches to perception in fields like reinforcement learning, but approaches to behavioral structure have not experienced the same level of success. Using deep neural networks to represent complicated behaviors lacks some key features: composition, communication, and abstraction.

Composition: Behaviors must be able to be constructed by using other behaviors. Consider this perspective: at a low level, the movements of all animals consist of a small set of possible muscle contractions, stitched together to achieve a higher purpose. The same movement in different contexts can be part of very distinct high-level behaviors. There is some evidence in mammals of what could reasonably be called a grammar of behavior, where behavioral syllables are short, stereotypical movements that when stitched together become the overall movements of the animal. Even high-level behaviors can be composed, such as what humans do when they plan

their day.

Monolithic neural networks, however, are entangled in a way that makes decomposition very difficult. Due to how outputs are calculated, it is generally not possible to attribute a behavior to a specific part of the network, and consequently it is difficult to re-use behaviors and combine them. What is needed is a hierarchical foundation for constructing behavior. Hierarchical reinforcement learning takes an integrated approach, where the existing concepts of reinforcement learning are extended in various ways to support hierarchical structure. As an alternative, entirely different representations and approaches can be used. It is well known that compositionality is a main feature of many programming languages, both imperative and functional; programs can be written and understood in parts, and the meaning of the whole is the combined meaning of the parts. Using programs as structured policies is one of the main concepts considered in this thesis.

Communication: Abstracted behaviors must be able to be communicated to and understood by other agents, including humans. Understandable behavioral policies are important in many applications, for example in safety critical systems where behavior must be verified. Envisioning a future where humans will have more and more interaction with artificially intelligent agents, it could even become a requirement for certain classes of algorithms to be interpretable. For example, it is critical for an autonomous vehicle to behave in a safe and predictable manner, and if it happens not to, being able to understand why is very valuable.

Beyond interpretability to humans, understandable policies can be understood by other artificial agents. Different agents will experience different settings, and thus will learn different useful skills. If skills cannot be shared, agents are bound by a common set of skills that they are "cloned" with, plus any skills subsequently acquired on their own. A programming language represents a structured, shared foundation that agents can potentially communicate by – if two agents understand the same language, and share a set of grounded variables in this language, any related programs can be shared between them. This grounding of variables is one of the subjects of the last feature: abstraction.

Abstraction: Behaviors must be able to be used in different settings. A recipe can be followed at home or in a restaurant kitchen, and it probably does not matter if one uses a plastic or metal bowl to mix the ingredients. Focusing on certain aspects of a situation while ignoring others is central to hierarchical reasoning, where concepts at different levels of the hierarchy can be "hidden" from consideration. In order to call a function, programmers do not generally have to consider how a CPU works, how the specific function is implemented, or even how many lines of code the implementation contains. All such aspects are abstracted away by a hierarchy, and without this hierarchy the programmer would be overwhelmed by the complexity of even simple tasks.

Another perspective on abstraction is through concept learning and perceptual grounding. This is critical to understandability and communication; behavioral policies should preferably make decisions based on concepts that correspond to things in the real world, and choosing the right level of abstraction for such concepts is essential.

In our recipe example, the correct abstraction would probably tell us that the color of our bowl is irrelevant. The wrong abstraction, perhaps learned by cooking only with red bowls, tells us not only that we cannot use a green bowl, but also that a red shoe will do. One way to make sense of this is through causality – would the outcome (successfully cooking a meal) be any different, had the policy been to use any color of bowls instead of only red ones? This perspective leads to the consideration of causal state abstractions, the other main concept of the thesis.

1.1 Contributions

To this end, I have worked in two directions that are useful for learning to decompose and compose communicable behaviors in varied settings:

1. Behavioral policies with compositional, programmatic structure.

2. State abstractions obtained through causal analysis.

More specifically, the work described in this thesis combines concepts from reinforcement learning, program synthesis, and causal analysis. First, by using program synthesis through local search in program space to discover programmatic policies that imitate neural policies. Second, by using causal analysis in a reinforcement learning setting to learn a causal state abstraction that is both useful for maximizing an outcome and attainable by an agent.

The structure of the remainder of the thesis is as follows.

Chapter 2 contains a short presentation of some necessary theory for reinforcement learning, and a discussion of program synthesis basics.

Chapter 3 examines a variety of approaches at the intersection of program synthesis and reinforcement learning. First, some key dimensions of program synthesis and programmatic policy learning are identified. Methods from the field of genetic programming are discussed, before machine learning approaches are discussed, both in relation to the identified dimensions and by expanding each dimension into several classes of approaches. Finally, a selection research from other related fields is presented, such as papers on hierarchical reinforcement learning and generalization. The goal of this chapter is to give both a broad overview of the fields, and some insights into different important aspects of the fields.

Chapter 4 presents the first paper, “Programmatic policy extraction by iterative local search”. The paper introduces a neighborhood structure around a given program under a domain specific language. The neighborhood is used to iteratively search for programs that optimize an imitation loss between the program and an existing policy, such as a neural network. The chapter introduces some necessary theory that is not covered in the paper, before presenting the search method itself.

Chapter 5 presents the second paper, “Reinforcement learning of causal variables using mediation analysis”. In the paper, a method is described for discovering an

abstract causal state variable by interacting with an environment. The abstract state variable has a causal interpretation as a mediator of the outcome of the considered task, and is useful for guiding policy learning and for interpretability. The chapter explains the reasoning behind the method, describing the used causal principles and how to optimize the state variable.

Chapter 6 concludes by summarizing the work done and specific contributions, before discussing some perspectives on the research and potential future work.

Drafts of both papers are included, as part of the thesis, in Appendix A and Appendix B.

CHAPTER 2

Background

This chapter contains necessary background knowledge for the main topics of the thesis.

2.1 Reinforcement learning

Reinforcement Learning (RL) is a machine learning approach to goal-directed behavior. The following exposition is loosely based on the book by R. S. Sutton and Barto (2018), plus additional sources cited in the text. Another book I would recommend is Bertsekas (2011).

2.1.1 Markov Decision Processes

A Markov Decision Process (MDP) is a commonly used formalization of a sequential decision making problem. Although it is a relatively simple idealized representation, many problems that occur in real life can be formalized. Commonly, sequential decision making problems are decomposed into an environment and an agent that interacts with the environment; here, the environment has a set of states $\mathcal{S}$, and the agent has a set of possible actions $\mathcal{A}$. When the agent acts, it has the possibility of receiving a numerical reward from the set of rewards $r \in \mathbb{R}$. The environment has a transition function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$:

$$S_{t+1} = f_t(S_t, A_t, W_t), \qquad t = 1, \ldots, T-1, \tag{2.1}$$

and for a given problem in the environment, there is an associated reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

In the case of a *finite* MDP, the state, action and reward spaces are all finite. The environment transition function is a discrete probability distribution,

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_t = r|S_t = s, A_t = a\}, \tag{2.2}$$

in which we combined the state transition function and reward function into a four-argument probability distribution.

The above also states the *Markov* property, since $p(s', r|s, a)$ completely characterizes the environment transition dynamics; as shown, the *current* state is sufficient information to calculate (the distribution over) the next state, when given an action.

The agent chooses actions according to a policy, or more generally a sequence of policies,

$$\pi = \{\pi_1, \dots, \pi_{T-1}\}, \tag{2.3}$$

where each $\pi_t : \mathcal{S} \to \mathcal{A}$ maps a state to an action. Commonly, the agent has a fixed policy, $\pi_t = \pi$. The actions are constrained to a nonempty subset of the action space, which depends on the current state, $\pi_t(s_t) \in \mathcal{A}_k$.

The goal of the agent is to maximize the expected reward. We define the future expected reward, or the *return* G_t from time t,

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_{T-1} + R_T. \tag{2.4}$$

The steps $k = 1, \dots, T$ constitute an *episode*. Problems that fit into this description are called *episodic*, but some problems are better described as an infinite episode, usually called an *infinite horizon*. In order to handle both finite and infinite horizons in the same framework, it is common to introduce a *discount rate* γ, where $0 \leq \gamma \leq 1$,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{T} \gamma^k R_{t+k+1}, \tag{2.5}$$

which ensures that even when $T = \infty$, the return is finite as long as $\gamma < 1$. The discount rate also determines the agent's time preference, or how much it prefers reward now over reward later.

Associated to a policy is the *state-value function*, which maps each state s to the expected return from following the policy π starting from s,

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} | S_t = s \right], \quad \forall s \in \mathcal{S}. \tag{2.6}$$

There is also the *state-action-value function*,

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right], \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \tag{2.7}$$

which maps a state s and an action a to the expected return from following the policy π after taking action a in state s.

Both the return and the value functions can be written recursively. For the return,

$$G_t = R_{t+1} + \gamma G_{t+1}, \tag{2.8}$$

it is a straightforward rewrite of (2.5).

Using (2.8), the state-value function becomes

$$\begin{aligned} v_\pi &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] && (2.9) \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right], \quad \forall s \in \mathcal{S}, && (2.10) \end{aligned}$$

which can be seen as an expectation over the expression in square brackets. The expression for q_π is similar. (2.10) is usually called the *Bellman equation*, and it is the basis for many reinforcement learning algorithms.

For a given problem, there exist (potentially multiple) *optimal policies*, and their common property is that they have the same, unique *optimal value function*,

$$v^*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S}. \tag{2.11}$$

The same holds for optimal state-action-value functions,

$$q^*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}. \tag{2.12}$$

A central property of optimal policies and value functions is called Bellman's *principle of optimality.*

Theorem 2.1 (Bellman's Principle of Optimality) *If $\pi^* = \{\pi_1^*, \ldots, \pi_{T-1}^*\}$ is an optimal policy, and the agent is at state s_t at time t, wanting to maximize the expected return v_π from time t to time T,*

$$\mathbb{E}_\pi^* \left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} | S_t = s \right].$$

In this case, the partial policy $\{\pi_t^*, \ldots, \pi_{T-1}^*\}$, which is identical to π^* from time t to time T, is optimal.

Simply, the principle indicates that optimal solutions can be constructed by starting from smaller, nested "tail subproblems"; the optimal solution for the full problem can be obtained by recursively solving and combining longer tail subproblems.

2.1.2 Dynamic programming

The optimal value functions can be written recursively, following the principle of optimality,

$$v^*(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v^*(s') \right], \tag{2.13}$$

and

$$q^*(s) = \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma \max_{a'} q^*(s', a') \right]. \tag{2.14}$$

These recursive forms are used as updates in several dynamic programming algorithms that can be used to solve smaller, finite MDPs when the transition function f is known.

The first useful algorithm is called *iterative policy evaluation*, and it is used to calculate the value function for a given policy. Calculating the value function is also called the *prediction problem*. The policy evaluation update is

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right], \quad \forall s \in \mathcal{S}, \tag{2.15}$$

where k refers to algorithm iterations. The update is a fixed point iteration, converging to v_π as $k \to \infty$.

Policy evaluation can be used in an algorithm called *policy iteration*, which iteratively improves a policy until it is optimal. Learning a policy is often called *control*. After evaluating a policy until convergence of the value function, the policy itself can be updated as

$$\pi(s) \leftarrow \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \quad \forall s \in \mathcal{S}, \tag{2.16}$$

which intuitively updates the policy at all states, to select a better action if one is available according to the value function. After each policy update, the policy is reevaluated before being updated again until convergence.

A similar algorithm, *value iteration*, does not iteratively evaluate each policy until convergence. A single iteration (or sweep, as it is usually called when all states are involved) of (2.15) is used to update the value function, before a single iteration of policy improvement by (2.16).

These algorithms are instances of what is called *generalized policy iteration* (GPI), where policy evaluation and policy improvement interact, improving until becoming optimal and therefore consistent. GPI is independent of the details of these two processes, and so almost all RL algorithms can fit into the GPI framework.

2.1.3 Temporal-difference learning

Let us consider different approaches to the prediction and control problems. Specifically, we want to learn value functions and policies without knowing the environment dynamics f. A starting point is the following; After observing a number of episodes consisting of states, actions and rewards following a policy π, we want to estimate $v_\pi(s)$ for the observed states. The conceptually simplest way to do this is by using a Monte Carlo estimate of the return for each state, which can be done by considering only the first visit to a given state, or by considering every visit. A Monte Carlo update might look like

$$v(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t)\right], \tag{2.17}$$

where α is a learning rate and G_t is the observed return after time t.

The Monte Carlo approaches result in state-value estimates that are completely independent for each state, which can be a significant drawback in terms of speed

of learning. Instead, *temporal-difference* (TD) methods *bootstrap*, using estimates of other state values when updating the value of a state. In the simplest form, called *one-step TD*, take (2.17) and expand the return,

$$v(s_t) \leftarrow v(s_t) + \alpha \left[R_{t+1} + \gamma v(s_{t+1}) - v(s_t) \right], \tag{2.18}$$

where R_{t+1} is an observed reward and $v(s_{t+1})$ is the current estimate of the value of s_{t+1}. Thus, a state-value estimate can be updated based on a single state transition, while taking advantage of the estimate of another state's value.

Probably the most well-known TD method is *Q-learning* (Watkins and Dayan, 1992), which besides learning a state-action-value function q, is an *off-policy* method. Being off-policy means that the method learns the optimal value function, even if actions are being selected according to another policy. The Q-learning update is

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t) \right], \tag{2.19}$$

which converges to the optimal state-action-value function under two conditions: all state-action pairs must keep getting updated, and the sequence of learning rates must decrease according to some conditions given by stochastic approximation theory.

Another key aspect of modern reinforcement learning is *function approximation*; that is, using some kind of structured functional representation, most commonly a neural network, to represent value functions and policies. Doing this brings its own set of challenges, for example because updating the value of one state changes the represented value of other values in complicated ways. Nonetheless, function approximation is valuable, allowing for generalisation to unseen states, and also for handling high-dimensional, continuous spaces, and more. With the Deep Q-Network (DQN) architecture (Mnih, Kavukcuoglu, Silver, Graves, et al., 2013; Mnih, Kavukcuoglu, Silver, Rusu, et al., 2015), a lot of new interest was generated towards using TD methods with function approximation. The DQN uses several tricks to stabilize Q-learning with function approximation, and was shown to perform well on games in an Atari 2600 emulator. The Q-network with weights θ is trained by minimizing a sequence of losses,

$$L_i(\theta_i) = \mathbb{E}\left[(y_i - q(s, a; \theta_i))^2 \right], \tag{2.20}$$

where $y_i = \mathbb{E}\left[r + \gamma \max_{a'} q(s', a'; \theta_i^-) | s, a \right]$ is the target, compare to (2.19). Without delving into too many details, there are two things that help to stabilize the DQN learning: experience replay, and the target network. Experience replay stores observations (state, action, reward, next state) in a buffer, and these are sampled during training to calculate (2.20). The target network contains a copy of the weights θ, called θ^-, and these are only periodically synchronized.

2.1.4 Policy gradient learning

An alternative to the methods in the previous section, policy gradient methods, as the name suggests, directly calculate policy updates of the form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\overline{\nabla J(\boldsymbol{\theta})}, \tag{2.21}$$

where $\boldsymbol{\theta} \in \mathbb{R}^d$ is the parameter vector of the policy $\pi(a|s;\boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$, and $\overline{\nabla J(\boldsymbol{\theta})}$ is an approximation to the gradient of the cost function $J(\boldsymbol{\theta})$ that is close to the true gradient in expectation. In the episodic case (which we will stick to), we can use the true value function as cost function,

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0). \tag{2.22}$$

The gradient of the value function with respect to the policy parameters, without involving the derivative of the state distribution, is given by the *policy gradient theorem*.

Theorem 2.2 (Policy Gradient)

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\ &\propto \sum_s \mu_\pi(s) \sum_a q_\pi(s,a)\nabla\pi_\theta(a|s) \\ &= \mathbb{E}_{S_t,A_t\sim\pi}\left[q_\pi(S_t,A_t)\nabla \ln \pi_\theta(A_t|S_t)\right],\end{aligned}$$

where $\mu_\pi(s)$ is the state distribution that follows from choosing actions according to $\pi_\theta(a|s)$.

The first expression for the gradient given by the policy gradient theorem is exactly proportional to the true gradient, but it contains a sum over all actions. Instead, the final expression in the theorem gives us an expectation. Using this expression in an update of the form of (2.21), and using $E_\pi\left[G_t|S_t,A_t\right] = q_\pi(S_t,A_t)$, we get the *REINFORCE* update,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \nabla \ln \pi_\theta(A_t|S_t). \tag{2.23}$$

This update can be implemented in an automatic differentiation framework by constructing an objective whose gradient is the policy gradient estimator,

$$L(\boldsymbol{\theta}) = \mathbb{E}_\pi\left[G_t \ln \pi_\theta(A_t|S_t)\right]. \tag{2.24}$$

REINFORCE is perhaps the most simple policy gradient method, but it is not a very good choice in practice. For example, the gradient estimator can have a very large variance. It can also be quite expensive to obtain new on-policy experience for each update. There has been a lot of research on improved policy gradient methods, including *actor-critic* methods that also learn a value function, resulting in algorithms

like DDPG (Lillicrap et al., 2015), TRPO (Schulman, Levine, et al., 2015), PPO (Schulman, Wolski, et al., 2017), and SAC (Haarnoja et al., 2018).

Let us take the related algorithms TRPO and PPO as examples. Trust Region Policy Optimization (TRPO) maximizes a "surrogate" objective under a KL divergence regularization constraint which ensures that a single optimization step does not change the policy too much,

$$\text{maximize}_\theta \quad \mathbb{E}\left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}\hat{A}_{\theta_{old}}(S_t, A_t)\right] \tag{2.25}$$

$$\text{subject to} \quad \mathbb{E}\left[\mathrm{KL}\left[\pi_{\theta_{old}}(\cdot|S_t), \pi_\theta(\cdot|S_t)\right]\right] \leq \delta. \tag{2.26}$$

Instead of the return G_t, the surrogate objective uses the estimated *advantage* $\hat{A}$, which has a learned state-dependent baseline subtracted from it to reduce variance. Also, the surrogate objective uses an importance sampling estimator, allowing multiple updates with the same data, or outdated asynchronous updates, at the cost of higher variance.

Proximal Policy Optimization (PPO) simplifies TRPO, retaining its tendency for monotonic improvement. Instead of the KL divergence constraint, PPO uses a clipped objective function,

$$L^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}\left[\min\left(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}\hat{A}_{\theta_{old}}(S_t, A_t), \mathrm{clip}(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}, 1-\epsilon, 1+\epsilon)\hat{A}_{\theta_{old}}(S_t, A_t)\right)\right], \tag{2.27}$$

where ϵ is a hyperparameter that clips the importance ratio to between $1-\epsilon$ and $1+\epsilon$. Commonly, a state-value function (critic) is learned simultaneously, and the objective is further augmented with a policy entropy bonus to ensure exploration, resulting in an overall objective,

$$L^{PPO}(\boldsymbol{\theta}) = \mathbb{E}\left[L^{CLIP}(\boldsymbol{\theta}) - c_1(V_\theta(S_t) - V_{target})^2 + c_2 H[\pi_\theta](S_t)\right], \tag{2.28}$$

where c_1 and c_2 are constant hyperparameters.

2.1.5 Hierarchical RL

Many practical problems can be decomposed into subproblems. Doing so has many advantages, such as faster learning, better generalization, and transfer of knowledge to new problems by composing already known solutions.

Frameworks that have been studied since the late 90s include: options (R. S. Sutton, Precup, and S. Singh, 1999), MAXQ (Dietterich, 1999), and hierarchical abstract machines (HAMs) (Parr and S. J. Russell, 1998). A common theme among these, is that they are based on a generalization of MDPs called *semi-MDPs* (SMDPs). Whereas an MDP has no reference to the *time* an action takes, in an SMDP the time an action takes is generally a random variable. Temporally extended, higher-level actions thus fit into the SMDP framework; for example, a fixed sequence of primitive

actions (i.e. an open-loop plan) could be a higher-level action that takes time equal to the number of primitive actions it contains. In all cases, there is an underlying MDP containing only primitive actions, and all the usual RL algorithms can be extended to SMDPs. However, more interesting are the hierarchical approaches mentioned before.

The options framework (R. S. Sutton, Precup, and S. Singh, 1999) extends the set of primitive actions in an MDP with a set of temporally extended policies, called options. A *Markov option* is a triple (I_o, π_o, β_o) where I_o is the initiation set, a set of states that the option can be initiated in, $\pi_o : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the stochastic policy for the option, and $\beta_o : \mathcal{S} \to [0, 1]$ is the termination condition.

The MAXQ framework (Dietterich, 1999) consists of a decomposition of the MDP and the state-action-value function $q(s, a)$, into a hierarchy of smaller MDPs and an additive combination of value functions from the smaller MDPs. An interesting distinction raised by the original paper is the difference between *hierarchically* and *recursively* optimal policies. Hierarchical optimality refers to policies achieving the highest reward possible within the hierarchy, essentially the global maximum given the hierarchical constraints. Recursive optimality is weaker, describing nodes whose subpolicies are optimal given the fixed policies of their children. This allows subpolicies to be learned without referring to the overall context they're executed in, which could make it easier to transfer and re-use these subpolicies in different settings.

Hierarchical Abstract Machines (HAMs) (Parr and S. J. Russell, 1998) are finite state automatons that partially specify policies. There are four types of machine states: *action* states execute a primitive action, *call* states execute another machine, *choice* states select the next machine state according to a probability distribution, and *stop* states halts the current machine and returns to the machine that called it, if any. Learning takes place in the choice states, where it is possible to learn the choice probability distribution the way a usual stochastic policy is learned in reinforcement learning.

2.2 Program synthesis

The objective in program synthesis is to generate a program that matches a given specification. This is a general concept, where aspects such as what the "programs" are and how the specification is given varies greatly. Broadly, I would classify the approaches towards solving this objective into three major fields: Logical methods, genetic programming, and machine learning. While there is certainly overlap, especially in more modern research, each field has characteristic approaches to synthesizing programs. A good introductory resource is the book by Gulwani, Polozov, and R. Singh (2017), which surveys the field as of 2017. In Chapter 3, I survey the field from a different perspective which includes reinforcement learning, and of course also including new research that has been published since 2017. This section serves as a small introduction into the overall field, before the survey in the next chapter.

The program synthesis problem is in the general case intractable, or even undecideable, such as with a Turing complete language and some arbitrary constraint.

Search methods are used across pretty much all practical methods, whether they're more or less informed about the search space.

2.2.1 Logical synthesis

The logical synthesis methods consist of deductive and inductive methods. Deductive synthesis is an axiomatic approach, in which the problem description is given by a logical specification – although some newer frameworks use deductive search with an example-based specification. Programs are synthesized top-down, by using a theorem prover or axiomatic transformations to build a proof of the specification, in which the program is contained. This is generally very efficient and the resulting program will be correct by design, assuming that the specification is itself correct. Writing the specification can be as difficult or even more difficult than writing the program itself, but it might also be easier to reason about or prove properties of. An example of the deductive approach is Denali (Joshi, Nelson, and Randall, 2002), a code generator for machine code, whose intended use is to generate critical code in for example loops that would normally be hand-written. This task is generally referred to as superoptimization.

Inductive synthesis is not based on axioms, and is sometimes called syntax-guided synthesis. Instead of a full specification, only partial specifications are used, which can for example include incomplete program sketches or input/output data. In many ways, the class of "inductive synthesis methods" subsumes the other approaches to program synthesis, but I use the classification specifically to encompass methods based on formal logical systems and similar. An example of this is counterexample-guided inductive synthesis (CEGIS), which was first presented in the thesis by Solar-Lezama (2008); synthesis is split into a solver and a verifier, where the solver produces a program that satisfies the current partial specification, and the verifier checks whether the program satisfies a complete specification. If the program does not satisfy the specification, the verifier produces a counterexample, which is added to the partial specification before the solver produces another solution candidate. Although the verifier does need a complete specification, verification is much easier than synthesis.

2.2.2 Genetic programming

The field of genetic programming has a very long history, with the first example of the approach probably being the book by Holland (1992), the first edition of which was published in 1975. Appealing to evolution and natural selection in nature, the fundamental problem of finding structured solutions to a wide variety of well-defined, complex problems is discussed. The term *adaptation* is used to encompass these problems and their solutions, and the way adaptation happens is through an algorithmic process reminiscent of evolution.

Genetic programming as it is used today, is an evolutionary framework, which is used to solve problems thaT can be defined by specifying problem-specific: terminals

(constants etc.), primitive functions, fitness measure, termination criterion, and hyperparameters. With a problem specification, the general steps taken by a genetic programming algorithm are:

1. Sample an initial population of programs, called generation 0.
2. Repeat the following steps, creating a new generation in each iteration, until termination:
   a) Evaluate the fitness of each program in the population.
   b) Create a new population by combining and mutating the best individual programs in the current population.
3. When terminated, the best individual across all generations is usually designated as the result of the algorithm.

While the general structure described above usually holds, there are of course many variations across different algorithms. One major direction of variation is in the combination and mutation operators used to generate new programs. Commonly used operators are, as usual, somewhat biologically inspired, at least by name; reproduction, crossover, and mutation are three such operations. Reproduction simply copies a program to the next generation, while crossover combines random, consistent parts of two programs, and mutation makes random changes to a single program. Many other operators can be defined. Clearly, there is a lot of stochasticity involved in almost every algorithmic step and operator. This is usually necessary due to the combinatorial program space, with the drawback that the result of any particular run could be quite suboptimal.

2.2.3 Machine learning

The successes of deep learning in pattern recognition led to renewed interest in applying it to other problems, such as program synthesis. There are a couple of quite different approaches emcompassed by what is here called machine learning. The first approach is based on guiding the search in program space, by learning a probability distribution over programs or something similar. Another approach which compliments the guided search is to learn new language primitives, resulting in a language with a stronger bias towards good solutions. There are also approaches based on neural program interpreters (NPIs), although this is usually called program induction instead of synthesis. The programs are continuous vectors, such as weights in a neural network, and they are executed by a neural network architecture that might resemble a Turing machine or similar. The programs are usually learned end-to-end from example data. Another approach is to learn program representations with useful features. An example might be to learn a function that maps different programs that are semantically equal to the same vector. Finally, there are differentiable programming languages, which have a lot in common with neural interpreters. The program

representation is closer to code, such as a probabilistic programming language with differentiable control flow. This can be a great advantage, since interpretability often is one of the reasons we would use program synthesis methods at all.

The next chapter examines, in much further detail, common approaches to program synthesis, programmatic policies and more.

CHAPTER 3

Approaches to program synthesis and structured reinforcement learning

The field of program synthesis is very broad and varied. Unlike a field such as reinforcement learning, where there is some very useful, fundamental theory, there are many completely different approaches to the program synthesis problem. While I believe that general fundamentals might be established eventually, the current state of the art is distributed over an intersection of logical methods, genetic programming and machine learning methods. The purpose of this chapter is to review a relatively broad set of more or less modern approaches to program synthesis, with a focus towards reinforcement learning, through the lens of some key *dimensions* described in Section 3.2. This is by no means an exhaustive list of approaches, but hopefully it serves as an overview of some major approaches and an analysis of interesting variations between these.

3.1 Programs and reinforcement learning

Not much research actually tackles reinforcement learning with programs, but the more popular approach of hierarchical reinforcement learning is in many ways similar to a programmatic approach. An example is the Hierarchical Abstract Machine (HAM) method (Parr and S. J. Russell, 1998), where policies are state machines that can take low-level actions, call other state machines, choose the next state dynamically, or return to the calling state machine. The advantages of a programmatic approach have been mentioned multiple times, and the hierarchical RL methods are essentially a different, perhaps more weakly structured, approach to achieving some

of the same advantages. Accordingly, the following covers a good deal of interesting hierarchical RL research.

Even when the hierarchical RL methods *do* use more explicit program representations, such as in the HAM extension called Programmable HAMs (PHAMs) (Andre and S. J. Russell, 2001), they do not perform program synthesis. In PHAMs, learning happens in the joint semi-MDP induced by the program specification and the environment. It's possible to use variations of the Bellman equation, as described in Section 2.1, to learn the choice points of such a programmatic policy. The programmatic specification is then merely a prior specification of hierarchical structure, and what is learned must follow the specified structure. Other approaches have different tradeoffs, but the integration of program synthesis with reinforcement learning is needed to reap the full benefits of a programmatic representation.

The complication, then, is finding the right program structure. The Bellman equation is unfortunately not directly helpful, and neither is the policy gradient theorem. Program semantics (i.e. behavior) is generally quite sensitive to syntactical changes, in the sense that a small change to a program's syntax can have large effects on its behavior. Reasoning about the effects of such changes to programs is traditionally the domain of logical methods, but the integration of logic and reinforcement learning is itself a significant challenge. The intractability of applying program synthesis naively to reinforcement learning can be understood through a simple example. Consider the GPI framework described in Section 2.1.2; we could attempt to define program evaluation and improvement operators. If the improvement operator is a (local) search in program space, and the evaluation operator runs the programmatic policies that the search in the environment, then we have an iterative algorithm for program improvement. However, the evaluation operator does not use the *current* program, it uses all programs that are searched. Since evaluation is a process of environment interaction, the algorithm just described requires environment interaction proportional to the size of the program space per improvement step.

3.2 Dimensions in program synthesis

Gulwani (2010) defines three "dimensions", which captures a lot of the variation between different program synthesis methods. These are:

Program representation Can vary from full Turing complete, high-level programming languages, through low-level machine code, to more or less limited Domain Specific Languages (DSLs), decision trees, state machines, and so on. In some cases, the representation can also be something that isn't normally thought of as a program, but which has program-like properties, such as a neural network with specific structure.

Intent specification Synthesizing a program requires a description of the requested program. This ranges from formal logical specifications to input-output examples, or even just properties of the output.

Search method The way in which the program space is explored varies from enumeration of a well-defined space of programs, through biologically inspired functions that combine and mutate programs, to sampling from a learned distribution over programs.

Each of these dimensions contain multiple categories of approaches, some of which are dependent across dimensions. One example of this is a logically specified intent and a search method based on deduction; as mentioned previously, the deductive methods usually require a formal specification. Consequently, the space of possible methods is certainly not to be seen as some sort of cartesian product over the dimensions.

It is of course possible to consider further dimensions, although these might have diminishing returns in terms of explanation. For example, it could be worthwhile to consider the *fitness measure* as a dimension, especially in certain applications. Many approaches only consider *correct* programs, ones that fully satisfy either a partial or complete specification, as viable solutions. However, it can be a good idea in some cases to also include partially correct programs, at the very least as intermediary steps towards a solution. One reason for this is that partially correct programs can be useful stepping stones towards correct programs; this is often encountered in genetic programming. A related concept is to sometimes accept worse solutions than the current one while searching, for example to help escape local maxima.

Another related concept, which could be considered a dimension, is the *source of intent*. Although the overall intent probably always stems from a human, that perspective seems too reductive. Certainly, the intent is commonly sourced directly from a human user of the system. Sometimes the intent is an initially complete specification, sometimes it is a human-in-the-loop situation. There are however settings where the intent is not directly sourced from a human. Even settings where it might on the surface seem like a human is the source, the human is not always the immediate source. An example is when supplying input-output data as a specification, and the data is obtained by running an existing program; while one or several humans might have written the program, the immediate source of the intent specification is then the program. This dimension is very much related to the partial correctness dimension, since the source of the intent might also inform us of how we should treat the intent. It is possible that the specified intent is not actually what is wanted. It could be that observed input-output pairs are noisy, or even faulty. Humans can make mistakes in creating such data manually, or in creating the program that outputs the data. Machine learning commonly deals with such noisy data, but the concept is not often considered when applied to program synthesis.

Both of the above mentioned new dimensions will be considered in addition to the usual three that were listed.

3.3 Genetic programming

Genetic programming (GP) has been used to generate many kinds of programs, including policies for games and controllers for industrial systems. The most distinguishing feature of these approaches is that they maintain a population of candidate solutions. The population is evolved in an explorative, non-greedy fashion, which is therefore able to overcome locally optimal solutions.

In terms of the directions, the search space or program representation can vary greatly. Since the search does not rely on specific properties of the representation, but merely on operations that modify and combine them, any structured representation is possible. Intent specification is also flexible, although in the case of control it is common to use a system model to evaluate policies. This has implications for the fitness measure and source of intent, since a simulated system, whether predefined or learned, is likely to be inaccurate or wrong in some ways.

3.3.1 Program representation

Program representation is indeed a major axis of variation, with one overall distinction being the genotype-phenotype representation. As in most other methods, there are GP algorithms that directly represent and manipulate programs as abstract syntax trees. The programs in GP are called the phenotype, as an obvious reference to the "observed characteristics" in biological organisms. Some GP methods instead manipulate a genotype, which is an encoding of the program into a simpler, separate form. Using a genotype-phenotype mapping, a genotype is translated into a phenotype: an executable program.

An example of the separation of genotype and phenotype is Cartesian Genetic Programming (CGP) (Miller, 2011). The genotype is a string of integers, called genes. The genes encode a partially connected feed-forward graph of a program, with the associated functions of the nodes, connections between nodes, and so on. One benefit of this separation is that the genotype can be significantly more simple than the phenotype, while only the genotype is passed on to future generations. Another benefit of this representation is that the genotype can contain genes that are not used in creating the phenotype, but their modification can impact future generations if they become used, which can lead to solutions that otherwise wouldn't be found.

Montana (1995) found that types were useful in GP. The basic GP methods assume that nonterminals can accept any terminal or nonterminal as arguments, which essentially means that everything must be the same type. By using multiple types, the efficiency and interpretability of the search can be improved. Harding et al. (2012) extend CGP with types. Their method, Mixed Type CGP (MTCGP), supports scalar reals and vectors of real numbers, as it is described in the paper. It should be possible to extend the method to additional types if needed.

Wilson et al. (2018) use (MT)CGP to synthesize policies for Atari 2600 games. The program input is the game screen, which at a pixel level is quite a lot of inputs even with downscaling. By using a vector or matrix type, it becomes possible to

add functions to the library that process the entire image instead of single pixels. Although it is mentioned in the text that computer vision functions were not used, it seems like it would be useful to include functions such as edge and blob detection. Not only would this be entirely possible with a type system, the efficiency could be improved by having more specific types for different kinds of vectors and matrices such as images.

Another genotype-phenotype representation is Gene Expression Programming (GEP) (Ferreira, 2001). Like in CGP, programs are encoded in genes. Each gene is a string consisting of a head and a tail, encoding a syntax tree in level order. Each symbol in the head represents a terminal or non-terminal in the syntax tree, while each symbol in the tail can only represent a terminal. This structure ensures that a gene always encodes a proper syntax tree, as long as the tail is long enough to fill in any needed terminals at the bottom of the tree. A gene can contain a coding part, called the open reading frame (ORF), and a non-coding part following it. This means that a fixed-length gene can code for syntax trees of different sizes, containing a number of nodes up to the length of the head plus the tail. As in CGP, non-coding parts of genes can be useful during program search. An advantage of the GEP representation is that any gene following the simple head-tail grammar described will result in a valid phenotype. This simplifies mutation and crossover operators because constraints aren't necessary as with CGP genes.

Multiple genes can be used to solve a problem with GEP, and several kinds of linking functions can be used to combine multiple genes into an overall solution. By letting genes code for subprograms, it becomes possible to hierarchically learn and combine solutions to subproblems, reusing previous solutions if they are found to be useful. For example, each gene could code for a part of a sequential policy, and when combined each part of the sequence solves a sub-problem leading towards an overall solution.

In terms of application, GEP has not been used much for synthesizing controllers in the literature. In his thesis, Mwaura (2010) describes the synthesis of modular robot behaviors with GEP. An obstacle avoidance controller is synthesized, using a simple language consisting of a single predefined function, if less than or equal to, and a number of sensor variables and motor outputs. More complicated controllers are also evolved, such as a wall-following controller consisting of multiple linked sub-behaviors.

3.3.2 Program evaluation

Evaluation of programs in GP is another area of variation. Miller (2011) uses probabilistic tournament selection, where a winner is selected among a small set of competing individuals.

Sipper (2011) describes evolutionary algorithms and genetic programming through the lens of games. Using both classical board games such as checkers and chess, and more involved game environments such as Robocode, a number of evolutionary

algorithms are described. These algorithms can produce high-performing agents, in some cases even winning Robocode competitions against complicated hand-coded agents. Pawlak, Wieloch, and Krawiec (2015) in some way inverts the execution of programs, in order to make it easier to discover useful intermediate states that lead to the wanted program state. Kelly and Heywood (2017) learn to play multiple Atari games by evolving what they call Tangled Program Graphs. These graphs organize what the authors describe as teams of programs, which is a concept where each program describes a context in which an action could be taken, and a bidding process decides which action to actually take in each state. The graph emerges through evolution, since programs can generate either an atomic action, or a reference to another team. As a follow-up, Kelly and Heywood (2018) evaluate the Tangled Program Graphs on more Atari games, with a single evolved policy showing good multitask performance on up to 5 games. The algorithm is also simplified, using an elitism objective instead of a Pareto objective.

3.3.3 Search methods

The search methods used in GP are usually quite similar, with mutation and crossover operators progressing the search in combination with the selection criteria. Still, there are approaches that differ, while retaining the evolutionary approach that is unique to GP.

An example of this is to introduce derivative information. Differentiable CGP (Izzo, Biscani, and Mereta, 2016) extends CGP by using a generalized dual number representation to calculate derivatives of programs. The output of a program in dCGP is a truncated Taylor polynomial, containing information about the program input and all derivatives up to the truncation order with respect to chosen parameters. One application of this is to discover the values of numerical constants used in the programs, something that is traditionally a difficulty when using GP. The usual approach involves *ephemeral constants*, which is forming new constants from functions applied to pre-defined constants.

Salomon (2003) discusses an inefficiency arising from resampling when using genetic algorithms. Especially, algorithms that have a high crossover probability and a low mutation probability have this deficiency, since they tend to stay in the same part of the search space for longer. An algorithm that only uses mutations suffers a constant runtime complexity increase due to resampling, while crossover algorithms have a runtime increase that is logarithmic in the dimension of the search space. Since this is not due to the operators themselves, but rather the random application of them, the author presents a deterministic genetic algorithm that does not apply the operators randomly.

Kamio, Mitsuhashi, and Iba (2003) attempt to integrate reinforcement learning and GP, using GP to synthesize programs that solve the required task in a robotic simulator. Afterwards, the GP solutions are used to initialize a reinforcement learning approach, where the state space is divided according to the programs. While it can

be said that their actual approach is not much of an integration of RL and GP, the authors do describe a more general iterative framework in which reinforcement learning happens on the results of GP, and the results from reinforcement learning affect the next round of GP synthesis.

3.3.4 Source of intent & fitness function

In GP, the source of intent is usually encoded in a selection function, giving a computational criteria for selecting the individuals from the population that will be involved when applying the evolutionary operators. The selection function is usually probabilistic, although it can take many different forms, allowing for programs that do not perform as well as others to still have a chance at propagating to future populations. Schmidt and Lipson (2009) identify nontrivial physical equations from observed data. The authors argue that finding invariances is not enough to find good conservations laws; the key point is that synthesized equations should link the derivatives of different groups of variables over time. The authors call this concept the predictive ability, and while GP has often been used for symbolic regression tasks, it seems that this insight is very useful for finding mathematical expressions of natural laws.

An interesting variation on the usual fitness or selection function is multi-objective GP. Multiple objectives can be combined into a scalar fitness function, for example to achieve a tradeoff between accuracy and program size (B.-T. Zhang and Mühlenbein, 1995). Alternatively, objectives can be kept separate, in which case the concept of Pareto dominance is relevant. If a solution is not worse than another solution on any objective, while being better on at least one objective, the former solution is said to Pareto dominate the latter. Jong and Pollack (2003) optimize for both performance, size, and diversity using a Pareto approach.

There are many approaches that utilize information beyond whether a particular example is correctly solved or not. The difficulty of various tests can vary, and compensating for this by weighing the different tests can be useful (McKay, 2000). This is especially the case when tests are automatically generated by for example sampling from a distribution; a uniform distribution over tests is unlikely to result in a constant difficulty, or even a uniform distribution of difficulty. Krawiec and Lichocki (2010) consider non-additive effects that occur due to the composition of skills. If multiple, separate learned capabilities are more useful in combination, the fitness function should reflect this.

3.4 Machine learning

3.4.1 Program representation

Due to the large variety of machine learning methods applied to program synthesis, there are also quite a few different program representations. Some of these represen-

tations can even be very different from what is usually thought of as a program; for example, some methods represent programs as vectors in $\mathbb{R}^n$. Still, such representations must retain some properties of what would more straightforwardly be called programs, such as a well-defined semantics or composability. This is also the case for the other approaches, such as those that attempt to learn a program representation.

Of course, many approaches use concrete syntax, abstract syntax, or similar, as their representation. A particular instance of this is "natural" code (Hindle et al., 2016), which is written by humans, and has structure reminiscent of written natural languages. Many "big code" methods use statistical language models like n-grams or RNNs, requiring large amounts of training data, in order to learn a distribution over natural code (Allamanis, Barr, et al., 2018). This can be useful for many tasks such as code similarity, decompilation, and static analysis. However, big code methods have not found much application to program synthesis; because of this, it is worth distinguishing between methods whose main representation is simply some form of syntax, and the big code methods. One reason for this is how difficult obtaining the right training data is. Although it is possible to obtain billions of lines of natural code, such as the data the Codex model uses from GitHub (Chen et al., 2021), the application of this approach to inductive program synthesis seems difficult. Nonetheless, in experiments the Codex model managed to generate functionally correct docstring-conditioned code. It should be noted that, in such a setting, it is up to the human end-user of the system to verify correctness of the code; this is discussed further in the section on intent specification. This approach has also raised concerns about the legal implications of training a model on code of unknown origin and licensing. Parts of the training data could be reproduced by the model, and used in ways not permitted by the original license.

Regarding the syntax representations, the main variation consists of the choice of language and related features. These range from low-level computer code, such as the RISC-V assembly in Simmons-Edler, Miltner, and Seung (2018), to high-level general purpose languages, such as Python in Yin and Neubig (2017). DreamCoder (Ellis, Morales, et al., 2018) uses the abstract syntax of the lambda calculus as a representation, and learns a probabilistic context-free grammar from which task-relevant programs can be sampled.

A large set of methods take advantage of gradient information to guide program search. While this is not possible directly in the space of (abstract) syntax, various code-like representations make this possible. Gaunt et al. (2016) present a DSL for describing inductive program synthesis problems. This allows separating the problem specification from the solver, which can be gradient descent, linear program relaxations, satisfiability modulo theory solvers, or Sketch (Solar-Lezama, 2008). Although one of the described optimization methods is gradient descent, the final program representation can be natural, readable code. However, an intermediate representation based on a differentiable interpreter is used during synthesis, which is similar to the method in Bošnjak et al. (2016). The continuous representation can be discretized into source code, at least if the discovered representation converges to something that is almost deterministic; as shown in their analysis, this is not always the case for

the gradient descent method, which can converge to a suboptimal, stochastic solution. Shah et al. (2020) introduce the idea of using neural networks as permissible heuristics for searching the space of differentiable programs. Their method substitutes neural networks for missing expressions in partial programs, and because the programs are differentiable the network can be trained on the problem-specific end-to-end loss. Pierrot et al. (2020) train a set of neural policies, and associated predictive models. This representation allows the primitive policies to be composed into programs, which sequentially execute several policies, and furthermore these programs can be used by a planning algorithm to create a sequential plan towards a given goal state. Tian et al. (2019) learn to generate graphics programs that draw 3D shapes. The programs are both generated by and executed by neural networks, and interestingly the program structure is defined by a grammar instead of continuous embeddings.

Some methods take the approach of extending recurrent neural networks with external memory, turning them into differentiable interpreters. This concept was introduced with Neural Turing Machines (Graves, Wayne, and Danihelka, 2014), which uses attention-based indexing to interact with the external memory, while programs are represented by the weights of the neural network. The idea of controlling an external memory with a neural network is much older (Das, Giles, and G.-Z. Sun, 1992), and was used to learn context-free grammars. Many papers extend this approach, such as Neural RAM (Kurach, Andrychowicz, and Sutskever, 2015), which uses a more advanced external memory model where memory cells can be interpreted as pointers to other memory cells. The architecture is shown to be able to learn several low-level computational tasks, such as reversing an array or searching a list. Bošnjak et al. (2016) describe a differentiable interpreter for the Forth programming language. The interpreter allows one to write program sketches before optimizing the programs with gradient descent. This is achieved with a continuous representation of the machine stack, heap, and program counter, along with differentiable read- and write-operations and differentiable implementations of Forth "words", which is what Forth calls subroutines. Essentially, everything is implemented with an attention-like architecture, with vectors indexing into the memory representations. D. Xu et al. (2017) learn neural programs, represented as a set of "APIs" which can be called by a meta-controller. Complicated tasks are thus decomposed into a set of neural modules, which in turn consist of a set of problem-specific low-level APIs. Specifically, the used low-level robotic primitives can move the gripper, activate the gripper, and release the gripper. The method takes advantage of many of the properties of a hierarchical task decomposition, such as learning to hide information that isn't useful for a given subtask.

Another class of methods take the approach of learning a program or similarly structured representation. This kind of representation is often used as a form of knowledge representation (Davis, Shrobe, and Szolovits, 1993), for which the purpose is to reason about the underlying structure of the programs. While many of these methods were not applied to program synthesis problems, learned representations could potentially be very useful for program synthesis. A good example of this is found in Allamanis, Chanthirasegaran, et al. (2017), where representations of se-

mantic equivalence are learned. The difficulty is that small syntactic changes can result in large semantic differences. Such a model might beneficially be applied to program synthesis, where semantical understanding of syntactical transformations is often missing in machine learning methods. Jetchev, Lang, and Toussaint (2013) learn to extract relational symbols from continuous observations. These symbols must be useful for an agent (robot) in terms of abstractly modelling the environment, and also for goal-directed planning in the resulting world model.

Some relevant methods do not learn a program representation, but instead another structured representation such as an object-based one. They are interesting to consider jointly, if not just because structured visual learning methods present a viable link between high-dimensional visual observations and structured, program-like representations. Steenkiste, M. Chang, et al. (2018) describe a method for learning to discover objects and their interacting physics in visual scenes. By learning to decompose the scenes in an unsupervised manner, the algorithm also directly learns to predict future frames of the scenes, even generalizing to scenes with more objects. Greff et al. (2019) describe a related model, learning to decompose static scenes by segmenting and representing objects jointly. The scenes are more complicated than demonstrated in Steenkiste, M. Chang, et al. (2018), but while the authors do apply their method to dynamic scenes, it is done by setting the number of "refinement" iterations to the number of timesteps in the data, which seems somewhat limited. Zuidberg Dos Martires et al. (2020) learn to anchor perceptions in an object-based probablistic model. This model is coupled to a rule system based on statistical relational learning, in which rules are learned for reasoning about the objects. Experiments demonstrate that the system can learn to reason about the perceived objects, such as maintaining belief about the existence of previously seen, but now occluded, objects.

There are methods which use less powerful formalisms than general programs. They can be easier to use and train, since the space of possible structures is much smaller. An example of this is Inala et al. (2020), where policies are represented as small state machines, with each state containing a separate neural policy and transitions between states consisting of predicate functions. Topin and Veloso (2019) also use state machines, in the form of Markov chains, to represent policies over an abstracted state space. The purpose is to learn a simple, explainable policy, not only by extracting the state machine structure, but also by grouping MDP states into abstract states.

3.4.2 Intent specification

Machine learning enables some unique ways of specifying intent. More abstract specifications are possible, due to advances in areas such as natural language and image processing. Input/output examples are also very relevant, since they are often the most readily available specification. Also, abstract specifications like natural language descriptions can run into issues such as ambiguity or synthesized programs being difficult to verify.

Nonetheless, text descriptions can be very end user friendly, since they are easier to write than the code itself. Yin and Neubig (2017) use natural language descriptions together with a neural syntax-guided model to synthesize Python programs. Chen et al. (2021) use Python docstrings to condition a language model to generate relevant code snippets. Ling et al. (2017) synthesize programs that operate over multiple-choice math problems, outputting both a choice and a text rationale for the choice.

Specification by input/output examples is, as in GP, probably the most common approach. Most of these approaches are straightforward – the intent is to find a program whose outputs match the given outputs on the given inputs. For example, Bošnjak et al. (2016) optimize a cross-entropy loss between program output and target output. DeepCoder (Balog et al., 2016) also uses input/output examples, feeding them to a neural network that predicts useful properties of the program that produced the examples. Specifically, the neural network is trained to predict the probability of a set of functions appearing in the code of the relevant program, and this can be used to guide the search. Lázaro-Gredilla et al. (2019) describe a visual-cognitive architecture, which is used for learning concepts represented as programs to be run by a robot. The architecture consists of an interpreter for a specially designed language that handles visual attention, robot hand control and indexing scene objects. Concepts are specified by pairs of schematic drawings, showing a configuration of objects before and after the concept is applied.

It is also possible to give the synthesis more supervision than just inputs and outputs, by also using intermediate states as targets. This is usually referred to as program or execution traces, and the purpose is usually to make learning more data efficient. One example is the method by D. Xu et al. (2017), which uses program traces to train a neural interpreter for hierarchical policies. Additionally, their approach uses meta learning in order to obtain additional intent at test time, where a single expert demonstration is used to condition the generation of the policy. Ellis, Ritchie, et al. (2017) use hand-drawn images as specifications, and learn to generate graphics programs whose outputs resemble these images. These programs do not take inputs, and therefore a single image corresponds to the output of a single program. Young, Bastani, and Naik (2019) extend this approach to generate more complicated images that have repetitive structure, such as photos of building facades. Burke, Penkov, and Ramamoorthy (2019) describe a hybrid robotic architecture, first fitting several proportional gain controllers. Then, a sequence of controllers (a trace) that solves the relevant task is used as intent in a simple search-based program synthesis approach, generating a program with loops and conditionals. The authors show that their system can learn visuomotor reaching tasks from demonstrations.

An interesting approach is to consider programs as intent. The intent could be given by a partial program, such as in Sketch (Solar-Lezama, 2008), but it could also be a complete program. It could even be something less structured, such as a reward function; Natarajan et al. (2020) introduce a variation on the Programming By Example framework, called Programming By Rewards. Similarly to reinforcement learning, the framework uses a black-box reward function that describes a problem-specific preference of behavior. Given a set of inputs, the objective is to synthesize a

program that maximizes the reward obtained by executing the program on the inputs. Specifically, programs from a language for decision trees based on simple if-then-else expressions are synthesized by using continuous optimization methods. The opposite of using a black-box reward, Christakopoulou and Kalai (2017) use a programmatic scoring function, calling these glass-box loss functions. Having access to all details of the scoring function is obviously more informative than a black-box scoring function which can only be observed at given inputs. The authors demonstrate an approach for taking advantage of this information, by learning PCFGs to solve glass-box synthesis problems.

A similar approach is to use a control policy or dynamical system as intent. For example, Penkov and Ramamoorthy (2017) perform program synthesis to discover programmatic system dynamics or programmatic policies, with the goal being that generated programs are more interpretable than the source of intent. Superoptimization methods also use programs as intent. Here, the idea is to find an optimized implementation of the given program. STOKE (Schkufza, Sharma, and Aiken, 2013) uses stochastic local search from a prototype non-optimized, compiled x86 binary, and sometimes discovers implementations that outperform fully optimized compiler outputs. Inala et al. (2020) describe a method for synthesizing state machine policies for sequential decision problems. Although state machines are quite simple, they are more structured than for example neural networks, which could lead to better inductive generalization. The state machines are constructed using an adaptive teaching approach, in which a teacher learns to solve problems by being parameterized as a state machine with a fixed sequence of states, each running for a fixed amount of time. The student then imitates the teacher as a probabilistic state machine, finally producing a deterministic state machine Topin and Veloso (2019) describe an algorithm for generating what the authors call Abstract Policy Graphs, the purpose of which is to explain a complicated neural policy through an abstracted state space. The Abstract Policy Graph is a state machine-like graph structure where nodes are abstract states that map to a number of low-level states, and edges are abstract actions that transition between abstract states. States are grouped based on a feature importance function, which can for example be a learned value function.

3.4.3 Search method

Machine learning methods approach program search by either directly searching the program space, or by learning to search.

Sometimes the program representation also makes it clear what search method to use. Continuous representations such as the neural interpreters can take advantage of gradient methods, which might reduce the computational cost of finding a good solution. D. Xu et al. (2017) propose Neural Task Programming, a method that learns to learn in order to generate a hierarchical policy from a single demonstration at test time. The architecture consists of an LSTM neural interpreter, running program embeddings. Bošnjak et al. (2016) optimize an end-to-end loss based on the

cross-entropy between program output and target output, with a mask that ignores irrelevant components on the stack. Shah et al. (2020) substitute neural networks for holes in partial programs. The authors show that the networks can learn a close-to-admissible heuristic, and by using a heuristic search algorithm such as A*, this can speed up program synthesis. Pierrot et al. (2020) combine a number of methods in order solve continuous control problems containing hierarchical structure. Their approach uses Monte Carlo tree search in the form of AlphaZero, to learn a neural interpreter for programs consisting of compositions of previously learned neural policies. Critically, learned models of the primitive policies enables the tree search planning in the interpreter. R. Singh and Kohli (2017) discuss a meta-approach to program synthesis, consisting of two parts: a specification encoder, and a program generator. The spec encoder learns to understand partial input-output specs from examples, and the program generator learns to search the program space conditioned on spec vectors from the encoder. They are jointly trained on a large set on samples of programs from a DSL with corresponding specifications.

Tree search and enumeration are common approaches, and statistical methods are useful, for example, for learning to guide the search. DreamCoder (Ellis, Morales, et al., 2018) is one such example, consisting of a probabilistic model for programs given a DSL. The algorithm learns not only a probabilistic grammar, but also a neural network recognition model which produces a posterior over programs given input/output data. Given a set of program synthesis problems, it also learns to expand the initial DSL with new functions, composed of functions already in the language; this is an alternative way of making the search more efficient, by acquiring functions that have a good problem-specific inductive bias. A precursor to DreamCoder, the Explore-Compress algorithm (Dechter et al., 2013) also learns both a distribution over programs from a DSL, and new programs for the DSL that maximally compress previous solutions. Yin and Neubig (2017) use a neural generative model to synthesize programs in a general purpose language, such as Python. Programs are generated in steps, applying a single production rule or generating a single token at a time, and the generation probabilities are conditioned by a natural language description of the requested program.

The setting in Ling et al. (2017) is interesting, because not only must a succesful program answer the given multiple-choice question, but it must also output a natural language rationale, including math, for the correct choice. The model generates programs by sampling sequences of instructions that output words or perform calculations. Young, Bastani, and Naik (2019) apply a neural generative model to images, using programs to describe repetitive spatial structure. Sampled graphics programs are used to generate training data with known spatial structure, and a generative model is learned that infers the structure of a given image. Odena, Shi, et al. (2021) demonstrate a method for learning bottom-up program synthesis, by training a classifier that predicts whether an intermediate value produced by executing a subprogram is going to be part of the overall solution. Such a model needs to be quite fast, quite reliable, or a mix of both, since evaluating a neural model is generally orders of magnitude slower than executing a small (sub-)program. Thus, the model is batched

across hundreds of subprograms, also called intermediate values, and property signatures (Odena and C. Sutton, 2020) are also employed. This combination allows the classifier to be both performant and accurate enough at the same time, in order to speed up synthesis.

It is also possible to learn to guide the search for programs by reinforcement learning. For example, Zaremba and Sutskever (2015) use a reinforcement learning algorithm to train Neural Turing Machines that interact with discrete memory interfaces. This differs from the original Neural Turing Machines (Graves, Wayne, and Danihelka, 2014), where attention-based memory was used in order to make the interaction differentiable. Simmons-Edler, Miltner, and Seung (2018) frame the synthesis of RISC-V machine code as an MDP, with actions corresponding to lines of code. Hence, the "agent" must output a program that maximizes the reward, which is based on input/output examples.

Meta-learning approaches generally perform learning-to-learn, discovering models that can use additional problem-specific data at test time to condition the synthesis of a solution. D. Xu et al. (2017) describe a meta-learning algorithm, where task demonstrations are used to condition policy generation at test time. During training, a hierarchical task decomposition is learned, such that new tasks can be solved compositionally in testing.

Some approaches combine machine learning with another method for program synthesis. Quite commonly, Sketch (Solar-Lezama, 2008) is integrated with other methods. One example of this is found in Ellis, Ritchie, et al. (2017), where the method learns to generate LaTeX graphics programs from hand-drawn illustrations containing simple shapes. Their method uses a neural network to generate a trace, which in this case is a set of graphics commands, then uses the Sketch framework to synthesize the trace into a program. However, they also train a simple probabilistic search policy and show that it outperforms Sketch when searching for the minimum cost program.

3.4.4 Source of intent & fitness function

It is common to use differentiable loss functions in machine learning, and this has mostly translated to fitness functions in program synthesis. Sources of intent are a bit more varied, and some interesting approaches to source intent can be found.

An interesting source of intent can be found in papers that automatically generate specifications, usually for training purposes. Especially neural models can benefit greatly from such an approach, since they require a lot of training data, which in some cases can be automatically generated. Young, Bastani, and Naik (2019) sample programs whose outputs are used as training data with known programmatic structure. Since structural generative information isn't available for real images, sampling rich graphical programs with known structure is a valuable source of data for the neural model. Ellis, Ritchie, et al. (2017) synthesize minimum-cost programs for their set of hand-drawn images, using the programs as a training set for a search policy over such

graphics programs. Similarly, Ellis, Morales, et al. (2018) use program samples that they call "fantasies" to learn a task-conditional distribution over programs. These programs should represent tasks that could realistically be encountered, in order to contribute meaningful information to the training. R. Singh and Kohli (2017) sample a large set of programs from a DSL, and use these together with their corresponding specifications to train a neural specification encoder and accompanying neural program generator.

Another source of intent is an existing control policy. This could be a neural policy obtained through reinforcement learning, or it could have another form and be obtained in another manner. Usually, the purpose of these methods is interpretability of the resulting programmatic policy, but generalization ability can also be a goal. The main reason that the source of the policy matters, is that a trained policy is likely to be suboptimal, especially in off-policy states. However, a policy defined in some other way could be optimal, or otherwise have desired properties. This should probably be considered when designing synthesis methods that deal with existing policies, and the consideration applies to most if not all the approaches that specify intent as previously learned policies. For example, Penkov and Ramamoorthy (2017) combine gradient descent and structure search in order to find functional programs that can explain e.g. a reinforcement learning policy. Their structure search uses A* search guided by gradient information in order to select program proposals.

Somewhat similar to having a policy, is having a programmatic reward function as intent. This does not seem to have been used much, but Christakopoulou and Kalai (2017) present such reward functions as "glass-box" loss functions, since it is possible to look inside the reward function and understand it. The authors experiment with manually defined glass-box loss functions, but it seems interesting to consider if it could be useful to synthesize glass-box loss functions.

One distinction for the fitness function is whether syntax or similarity is used for learning. This is for example discussed by R. Singh and Kohli (2017), where it is argued that while syntactic similarity measures offer rich supervision and are thus easy to optimize, they can penalize many good programs that are semantically similar, but syntactically distinct. Meanwhile, input/output measures are consistent with semantic similarity, but offer less supervision, for example due to not being defined for partial programs. Methods based on language models often use syntactic similarity. For example, Alon et al. (2020) learn to generate missing pieces of source code with a structured language model operating on abstract syntax trees.

3.5 Other relevant areas

This section is less structured than the previous ones, but includes many important topics that are of relevance. A lot of the following research is not less relevant to the topics considered in this thesis than the research described in the previous sections, but it does not fit into the same descriptive framework.

3.5.1 Logical methods

There is a lot of important and interesting work combining logical reasoning, programming language theory, and program synthesis. Since it has not been a major focus of my studies, however, it has been relegated to a small subsection, that is nonetheless included for the sake of mentioning the topic. The methods are very relevant for the program synthesis field as a whole, but they seem difficult to apply to realistic reinforcement learning settings, at least because it is common to only consider programs valid if they perfectly satisfy the constraints given by e.g. input/output examples.

Sketch (Solar-Lezama, 2008) uses a verifier, which produces counterexamples. During synthesis, a candidate produced by the synthesizer is put through a verification process. If verification fails, a counterexample is produced, which is an input on which the candidate fails to produce the correct output; in that case, synthesis is performed again, with the additional information from the counterexample. Polozov and Gulwani (2015) introduce a methodology that they call data-driven domain-specific deduction. The most interesting thing is the program search strategy, which is deductive search combined with enumerative search – potentially very relevant! Deduction is quite efficient, but I think it requires fitting all the i/o data perfectly. They also view their work as a unification of much previous work on PBE/inductive synthesis. Feser, Chaudhuri, and Dillig (2015) describe λ^2, a method that first generalizes i/o data to hypotheses, before synthesizing programs matching the hypotheses by combining deduction and best-first enumerative search. Osera and Zdancewic (2015) demonstrate a method for synthesizing recursive functional programs given input/output data and type information. The paper contains a good walk-through of how types can guide synthesis and formalizes many of the concepts in doing so.

3.5.2 Automated code feedback

Keuning, Jeuring, and Heeren (2018) review approaches to automatically generating feedback for programming exercises. Generating feedback on code requires deep understanding of how task descriptions relate to code structure, which would also be very useful knowledge to apply to program synthesis. Piech et al. (2015) learn to embed so-called Hoare triples, describing how a piece of code changes the program state, as a feature vector which can then be used as a "one-step" predictor of program behavior.

3.5.3 Other reviews and resources

Besold et al. (2017) review the field of neuro-symbolic learning, in which the goal is to integrate connectionist neural models with symbolic reasoning methods such as logic based systems. A wide array of researchers in the field present their personal thoughts on the topic. S. Zhang and Sridharan (2020) review methods for doing sequential decision making under uncertainty (e.g. reinforcement learning) while leveraging (reasoning with) declarative knowledge. Several issues in the intersection

of these fields are discussed, such as which representations to use, how to learn declarative knowledge incrementally, and how to combine reasoning, learning, and control. The program synthesis book by Gulwani, Polozov, and R. Singh (2017) is a nice introduction to the field as a whole, and contains information on several approaches that are only superficially considered here.

3.5.4 Model-based reinforcement learning

An alternative to model-free reinforcement learning, which is just called reinforcement learning in this thesis, is model-based reinforcement learning. The approaches are mostly based on supervised learning, and by learning a model of the world, decisions can be made by planning in the model instead of using a trained policy.

Chiappa et al. (2017) learn recurrent predictive models for visual environments, including models that do not need to predict the high-dimensional observations at every step. M. B. Chang et al. (2016) introduce a Neural Physics Engine, factorizing a visual scene into objects and showing that it can predict object movement. Kipf et al. (2018) describe a neural architecture for learning to predict trajectories of systems of interacting objects, such as a set charged particles. Jaques, Burke, and Hospedales (2019) describe a combined neural inverse-graphics and differentiable physics engine. The approach structures prediction by decomposing scenes into objects and their interactions, which for example is promising in terms of higher quality long-term predictions. Hasselt, Hessel, and Aslanides (2019) explore nuances in using an explicit (parametric) model, versus using experience replay. While experience replay is often seen as being a model-free method, it can in fact be viewed as a model. It is argued that experience replay methods are at least as good as model-based methods, when the model is used to generate ”imaginary” training data. However, a model might be better used for forwards or backwards planning.

3.5.5 Evolutionary strategies and reinforcement learning

Salimans et al. (2017) experiment with using a black-box optimization algorithm instead of RL methods such as temporal differences or policy gradients. By sampling random policies nearby the ”current” policy, and performing rollouts with these slightly varied policies, an approximate gradient can be obtained. Wang et al. (2019) simultaneously learn progressively more difficult environments and their corresponding policies. They use continuous parametrizations of both environments and policies, but it might be neat for one or both of them to be programs. Mania, Guy, and Recht (2018) demonstrate how a local search algorithm in the space of policies can be competitive with policy gradient methods that explore in the space of actions. Small perturbations of a policy can be used to estimate an unbiased gradient, and the basic algorithm can also be seen as a simple variant of Evolutionary Strategies (Salimans et al., 2017). Barreto et al. (2020) present a generalization of policy evaluation and iteration to multiple tasks and policies at once. The generalization allows for

composition of policies in a natural way, solving multiple tasks with less experience. The generalized equations can be seen as a framework which can be implemented in extended in multiple ways. Diuk, Cohen, and Littman (2008) introduce an MDP representation based on objects and their relations. Inspired by object-oriented programming, the state representation is a set of objects, each an instance of a class from a set of problem-dependent classes. The purpose of such a representation is to improve several aspects of reinforcement learning, such as sample complexity and generalization ability. Guo et al. (2014) demonstrate an alternative way to use planning in solving sequential decision problems; namely, to generate training data. The best results were obtained by training a classifier to predict the actions found through planning, and by running the planner on experience obtained in the environment by the partially-trained classifier, in order to better match the data distributions observed by the planner and the classifier. Justin Fu et al. (2019) explore the problem of describing reward functions in RL through natural language. By using inverse RL, the method allows language-conditioned reward functions to be represented by a neural network, which doesn't work by naively conditioning the policy on the natural language description.

3.5.6 Causality and reinforcement learning

Dasgupta et al. (2019) demonstrate that causal reasoning can emerge from training agents using traditional, model-free reinforcement learning. Bengio et al. (2019) describe a method for meta-learning causal structures. Through the assumption that the right causal structure leads to faster adaptation to distributional changes, it is shown to be possible to learn continuously parametrized causal structures in an end-to-end manner. Shalizi and Crutchfield (1999) describe a fundamental theory for representing and learning causal state representations, called ϵ-machines. The representation has several good properties, and the authors describe relations to other fields such as time series modeling and decision theory.

3.5.7 Hierarchical reinforcement learning

Hierarchical structure is a central concept of this thesis, and there exists an entire field of research that incorporates modularity and hierarchy into mostly traditional reinforcement learning methods. Some of this research even uses programs or program-like structures, but it does not contain what I would describe as actual program synthesis.

S. P. Singh (1992) describes Compositional Q-learning, one of the earliest attempts at combining Q-learning with a modular architecture that achieves transfer learning. This method learns solutions to a set of elemental (not decomposable) and composite decision problems, while being more efficient than learning to solve each problem separately. Another early approach to modular reinforcement learning is Karlsson (1997), where each module has its own state space and reward function. In order to

select actions, modules are arbitrated by a central algorithm which can be defined in different ways, such as selecting the action with the greatest combined expected utility. The main difficulty in using this approach is that useful state spaces and reward functions have to be defined for each module. There are attempts to alleviate aspects of this, such as Rothkopf and Ballard (2013), where an algorithm is presented for learning the individual modules and how to use them. Their algorithm allows learning the modules despite only observing a global reward, which is the sum of each module reward. Schmid (1999) describes a system for integrating planning with inductive program synthesis, in order to solve tasks with recursive structure. Planning is used to generate a program that solves an instance of the problem of small size, and a synthesis algorithm is then applied which results in a program that generalizes to larger instances of the recursive problem.

The MAXQ framework introduced by Dietterich (1999) decomposes decision problems by defining local termination predicates and reward functions. The presented MAXQ-Q learning algorithm learns recursively optimal policies, even under state abstraction. However, as noted in the paper, recursive optimality is weaker than hierarchical or global optimality, where solutions might need to be locally recursively suboptimal. Andre and S. J. Russell (2001) present an extension to Hierarchical Abstract Machines (Parr and S. J. Russell, 1998), which is a framework for specifying partial policies as state machines with learnable policies in certain states. The extension adds more general programming language constructs, including parametrised policies which could lead to better generalisation. Learning is achieved with standard reinforcement learning methods, while the programs merely specify prior knowledge about the structure of the problems.

In theory, modular policies allow for state abstraction, using only relevant state variables in each module. Andre and S. J. Russell (2002) demonstrate an approach to state abstraction in the setting where a program sketch is given, describing a partial policy. Specifically, the paper discusses safe state abstraction under hierarchical optimality, which means that the ignored state variables are indeed irrelevant. Barto and Mahadevan (2003) reviews the major approaches to hierarchical reinforcement learning, including the frameworks and approaches still being researched today. Also included is a discussion of relevant future work, which included (dynamic) representations, learning of hierarchies, and application to larger, realistic problems. Silver and Ciosek (2012) introduce a method for recursively composing option models, which enables planning over multiple layers of hierarchy. Using their generalized Bellman equation, it is possible to simultaneously learn option models and plan over these options to solve additional tasks.

Liu and Jie Fu (2019) propose a combination of options with temporal logic; for example, given two options that maximize the probability of two separate outcomes, what are the options that maximize the probability of outcome 1 *and* 2, or outcome 1 *or* 2. Beyond the conjunction and disjunction operators, their temporal logic also includes "next" and "until" operators. Bagaria, Crowley, et al. (2020) introduce an algorithm called Deep Skill Graphs, which constructs hierarchical skills to navigate the state space in an unsupervised manner. S.-H. Sun, Wu, and Lim (2019) specify

tasks/goals as programs containing control flow and learnable behavioral subtasks. The subtasks correspond to neural network policies, and are trained using standard reinforcement learning after being selected by evaluating the current program state.

Simpkins and Isbell (2019) describe a method for enabling composition of reinforcement learning modules, by solving the problem of misaligned reward scales between different modules. Their Arbi-Q algorithm for action arbitration consists of another reinforcement learner, which learns the arbitration policy on a state space that is potentially separate from the modules' state spaces. Andreas, Klein, and Levine (2016) present a method for describing prior procedural information, in form of a policy sketch containing a sequence of named subtasks. The subtasks are solved by constructing a policy corresponding to each named subtask, and learning these policies across multiple tasks at once. More complicated sequences are solved by using curriculum learning, which allows some of the subtasks in the complicated sequences to be solved before having to solve a full complicated sequence. Lin, Mausam, and Weld (2016) describe a programming language for implementing POMDPs; essentially, it is a language with choice points – language states where the behavior of the program can be optimized. A program is compiled into a set of states and a HAM (Parr and S. J. Russell, 1998), after which the POMDP is constructed from these. Running a program requires solving the POMDP, which can for example be done with an online Monte-Carlo planner.

Verma, Murali, et al. (2018) introduce the Programmatically Interpretable Reinforcement Learning framework, an approach to using programmatic policies. The main difficulty with programmatic policies is learning them, and the framework thus takes an indirect approach; by first learning a neural policy with standard algorithms, it becomes possible to synthesize a programmatic policy (supervised) imitation learning. This is demonstrated by using a neurally guided search method for policy imitation. Following up on this framework, Verma, Le, et al. (2019) demonstrate a method for learning programmatic policies by iteratively learning a neural policy with a policy gradient method, and imitating the learned policy with a program through program synthesis. The authors analyze this approach under the framework of mirror descent, viewing the imitation step as a projection operator. One practical application of this approach is the REVEL method (Anderson et al., 2020), which uses the programmatic policies to allow for provably safe exploration in reinforcement learning, avoiding the expensive verification of neural policies.

Moerman (2009) mainly presents the HABS algorithm, which modifies an earlier algorithm called HASSLE. Both algorithms are based on a set of uncommitted subpolicies, but they differ in how these subpolicies are learned and used. HASSLE has a set of high-level states or subgoals which abstracts the actual, lower-level state space, and by using an uncommitted set of subpolicies, subgoals are reached by dynamically assigning subpolicies to subgoals. In a sense, HASSLE uses subgoals as high-level actions. HABS does not keep these high-level states, but organizes itself dynamically by classifying subpolicies into behaviors, which however complicates training, since in HASSLE the subgoals are used in calculating rewards. HABS solves this by using a self-organization principle, learning an abstract state representation and using

clustering to find similar behaviours that are needed to solve the overall task. Devin et al. (2019) introduce compositional plan vectors, a way to represent trajectories as compositions. The vectors enable an agent to learn to solve sequential tasks consisting of several subtasks, and to then compositionally reuse solutions to the subtasks without any additional supervision.

Levy et al. (2017) introduce the Hierarchical Actor-Critic, an approach to learning multiple layers of behavioral hierarchy simultaneously. It is argued that this is important in order to best make use of hierarchical learning, but that it is difficult due to instabilities occuring from training the different levels. Higher-level policies are trained as if lower levels are already optimal, which reduces the instability. Bagaria and Konidaris (2020) present a method for learning options in high-dimensional continuous state spaces, by combining skill chaining (i.e. the termination condition of an option is the initiation condition of another option) with deep reinforcement learning. Their deep skill chaining method is compared to the method in Levy et al. (2017), generally outperforming it in a set of simulated continuous tasks. Eppe, Nguyen, and Wermter (2019) present an integration of symbolic planning with reinforcement learning which is quite different from hierarchical approaches. The integration requires hand-engineering of predicate-subgoal mappings, which grounds the planning in the low-level observations. However, such hand-engineering could be well worth it, since the method shows good performance in some simulated environments that require some level of causal reasoning.

In Hangl et al. (2020), skills are learned by autonomous playing, while simultaneously learning an environment model. Initially, narrow skills are learned from demonstrations, and their domain is then extended through active playing, guided by the model. Holtz, Guha, and Biswas (2020) demonstrate an approach for synthesizing high-level policies that compose lower-level policies by selecting one (also called arbitration). The approach uses a specific DSL that includes physical dimensions in the type system, and which otherwise consists of a sequence of if-then-else statements with a set of simple numerical predicates and some mathematical functions. The synthesis algorithm is quite complex and layered, for example using an SMT solver in order to solve contraints for parameter synthesis. Lyu et al. (2018) describe their Symbolic Deep RL (SDRL) framework, integrating symbolic logic with hierarchical deep reinforcement learning, for the explicit purpose of explainability of subtasks. The framework consists of a planner, a controller, and a meta-controller, which respectively performs subtask scheduling, subtask learning, and subtask evaluation. The symbolic representations consist of manually defined domain knowledge based on the action language $\mathcal{BC}$. Extending SDRL, Ma et al. (2021) replace the symbolic planning meta-controller with an inductive logic programming controller. The method seems complicated, employing Transformer networks for attention together with a manually designed symbolic space in order to represent and extract objects in the tested environments. First-order logic rules extracted from the system are used for action selection, and these rules also constitute the explainability element of the approach.

3.5.8 Generalization and abstract reasoning

Bahdanau et al. (2018) compare two different models by how they systematically generalize on natural language (VQA) tasks. Finding that modular architectures generalize better, they try to learn the layout or the parametrization end-to-end, which doesn't work well. Different ways to learn layouts seem necessary. Uesato et al. (2018) demonstrate a method for evaluating safety critical policies, by learning adversarial evaluation settings while still being able to estimate failure probabilities. This demonstrates the difficulty of properly evaluating black-box policies when safety is critical. Barrett et al. (2018) explore the capability of neural networks to learn and perform abstract, symbolic reasoning. Popular deep neural architectures generalize poorly on Raven matrices, a kind of IQ test. A new relational architecture performs better, especially if also trained to provide symbolic explanations. Similarly, Hill et al. (2019) explore learning analogical reasoning in neural networks. Using a visual analogy domain that is quite similar to Raven matrices used in intelligence tests, they find that it's important to choose the right data, and to present it to the model in the right manner, which they call "learning analogies by contrasting abstract relational structure".

Steenkiste, Greff, and Schmidhuber (2019) discuss generalization in model-based agents, arguing that structured models should be inferred on the fly. A key part of this structure is decomposing the world into objects, which should be represented in a way that fulfills a number of requirements in order to facilitate generalization. Indeed, with such a representation, it is argued that compositional reasoning can lead to powerful generalization beyond what current models can achieve. Gerstenberg and Tenenbaum (2017) explore the flexibility of human thinking, through what the authors call intuitive theories. These theories are domain-specific concepts and causal laws relating them. The theories do not simply describe what happens, they are more abstractly interpreting evidence through the lens of the intuitive theory. They are modelled through generative models, and supported by causal reasoning such as counterfactuals. T. Xu, Li, and Yu (2020) analyze the error associated with imitation learning of reinforcement learning policies. As would be expected, simple behavioral cloning has worse bounds on the value gap between expert and imitation policy, as compared to an approach that tries to correct the compounding errors associated with behavioral cloning (in this case generative adversarial imitation learning). Interestingly, it is also seen that such imitation learning approaches could be useful for model learning, since most methods use a behavioral cloning-like approach to model learning.

3.5.9 Natural behavioral hierarchies

Wiltschko et al. (2015) provide evidence of and a method for identifying simple modules of animal behavior. It is hypothesized that overall, complex animal behavior is hierarchical and consists of these simpler modules. Using 3D imaging of mouse behaviors together with an autoregressive hidden Markov model, behavioral mod-

ules are identified and a grammar of behavior is described for mouse body language. Additional evidence is presented by Berman, Bialek, and Shaevitz (2016), where a behavioral hierarchy in fruit flies is identified. Through an algorithm that decomposes images into a low-dimensional basis, a set of behavioral states and transitions between these states are discovered. Overall, the results show that behaviors consist of a deep hierarchy of many time scales, displaying memory for up to approximately 20 minutes.

CHAPTER 4

Paper 1: Programmatic policy extraction by local search

Learning programmatic policies is challenging, as shown by the many methods described in Chapter 3. Genetic programming approaches the problem heuristically, using evolutionary operators that amount to random search and recombination of high-fitness solutions. While GP has been shown to perform well in many cases, it seems unsatisfactory to rely on heuristics when reinforcement learning enables informed policy updates and thereby efficient policy learning. Accordingly, it would be beneficial to integrate a programmatic policy representation with existing modern reinforcement learning algorithms. Many methods take an approach where environment interaction is reduced, since this is the main bottleneck; in general, every candidate program must be evaluated through potentially expensive environment interaction. Approaches that reduce environment interaction include learning a parametric environment model (Hein, Udluft, and Runkler, 2017), evaluating fewer programs by learning to search more efficiently (Ellis, Morales, et al., 2018), and imitating an existing policy (Verma, Murali, et al., 2018; Bastani, Pu, and Solar-Lezama, 2018). The work presented here takes the idea of imitation, and proposes a search method that takes advantage of being able to search through a large number of programs.

As a starting point, consider the methods described in Verma, Murali, et al. (2018) and Verma, Le, et al. (2019). The former involves imitating a previously learned policy, turning programmatic policy learning into a supervised imitation problem. The paper contains a description of an algorithmic framework, and experiments with an instantiation of this framework where programs consistent with a predefined sketch are searched through iteratively. While a neighborhood is mentioned both in the pseudocode and in the text, it is either left open how to define it, or in the case of the mentioned instantiation, loosely defined as a set of program templates that are

structurally similar.

The latter paper extends the setting to one where the oracle policy is not trained first, but instead trained in steps that alternate reinforcement learning and program synthesis. By considering a joint policy space where the RL policy and programmatic policy are combined by adding them together, and by considering program synthesis a projection operator from this joint space to program space, the method is analyzed as a variant of mirror descent. While the former method is mostly useful for interpretability, since the starting point is a successfully trained policy, the latter iterative framework could have many more advantages. Due to the integration of programmatic and neural policy learning, I argue that it could be possible for program synthesis to speed up learning, or even to help discover better solutions; while this requires more research on the integration of knowledge obtained from program synthesis in the subsequent RL step, this work could be considered a step in that direction. The authors again instantiate the suggested framework, but in a quite limited manner that one could argue is not actually program synthesis. Using a language whose main construct is a PID controller, the projection step is performed by either fitting a regression tree (Breiman et al., 2017) or by finding some weights using Bayesian optimization (Snoek, Larochelle, and Adams, 2012).

The work described here is an attempt to define and implement a policy imitation method, which is more powerful because it can synthesize potentially complicated programs from typed domain specific languages by using the neighborhood heuristic. The point is that such a method would be very useful in the iterative RL setting described above, by discovering useful programs in a language with a strong problem-specific inductive bias. This chapter is based on work done for the paper "Programmatic policy extraction by iterative local search", of which a draft version is in Appendix A. Some additional topics that are not fully covered in the paper are also discussed.

The contributions in this work consist of: a complete definition of a neighborhood structure for programs from DSLs in a polymorphic lambda calculus, using the Hindley-Milner type system; discussion of relations to (deterministic) GP and Very Large-Scale Neighborhood search; initial experiments with the defined neighborhood structure for policy imitation; and (here) also a critique concerning aspects of programmatic policy imitation.

4.1 Lambda calculus and types

Before getting into the contents of the paper, an introduction to the topic of the lambda calculus seems necessary. Since this topic is readily found in dedicated textbooks (e.g. Pierce, 2002; Barendregt, Dekkers, and Statman, 2013), it is only briefly mentioned in the included draft. Nonetheless, the necessary aspects are covered here in a quite compact manner, with contents inspired by the textbook by Pierce (2002). I leave out many subtleties that are comprehensively covered by textbooks.

The lambda calculus is a formal system which can be used to describe computations, by using the operations of function definition and function application. While

the basic lambda calculus only uses these two operations, which makes it simple but also quite impractical, it is possible to extend the calculus with various convenient features such as numbers, tuples, standard library functions, and so on. Starting with the most basic, untyped lambda calculus, everything is constructed out of three different kinds of *terms*: a *variable* $\mathtt{x}$; an *abstraction* of a variable $\mathtt{x}$ in a term $\mathtt{t}$, $\lambda\mathtt{x}.\,\mathtt{t}$; and an *application* of a term $\mathtt{t}_1$ to another term $\mathtt{t}_2$, $\mathtt{t}_1\,\mathtt{t}_2$. These terms are part of the abstract syntax as described in Table 4.2. There is no need for considering any concrete syntax here, as synthesis happens directly in the abstract syntax.

In the untyped lambda calculus, with no extensions, running a program amounts to rewriting terms by applying functions to arguments. This is done by a substitution, written as $(\lambda\mathtt{x}.\,\mathtt{t}_{12})\,\mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_{12}$. The meaning of this notation is to substitute all free occurences (free meaning not bound by an abstraction) of $\mathtt{x}$ in $\mathtt{t}_{12}$ by $\mathtt{t}_2$. This rule is usually called the β-reduction rule, and a term matching the form on the left hand side is called a *redex*. Depending on the evaluation strategy, several other *congruence* rules are defined to fully specify the evaluation order. In the case of *call-by-value* evaluation, as used here, a redex is only reduced if the $\mathtt{t}_2$ part is a *value*. Values are syntactical elements which by definition cannot be reduced further, and in the pure lambda calculus, abstractions are the only values. With extensions, things such as numbers and other built-ins are likely to be considered values as well. The call-by-value rules are (E-App1), (E-App2) and (E-AppAbs) in Table 4.2.

4.1.1 Simple types

Programs in the untyped lambda calculus are not always able to be evaluated. Evaluation can get *stuck*, or it can continue forever. Especially the latter is an issue in program synthesis, but getting stuck on an invalid program is also a waste of computation. A solution to this problem is the simply typed lambda calculus, also called $\lambda^{\rightarrow}$, where valid programs are guaranteed to be able to be evaluated. The simply typed lambda calculus extends the untyped lambda calculus with *base types*, such as booleans (`Bool`) and natural numbers (`Nat`), and a type constructor $\rightarrow$ that is used to construct function types by combining base types or other function types. For example, $\mathtt{Bool} \rightarrow \mathtt{Nat}$ is a type describing a function that takes a boolean argument and produces a natural number as a result. The base types are inhabited by constant values, such as `true` and `false` for `Bool`. Using these types, it is possible to *statically* determine whether a program can be evaluated, that is, without actually evaluating it. Although this is considered a feature, it does limit the terms that are typeable; even if a program can be evaluated to a consistent type, the type might not be statically determined. A term `t` having type `T` means that it definitely evaluates to a value of type `T`.

While constants have their associated type, variables bound by abstractions must be *annotated* with types. This means that every leaf of an abstract syntax tree has a type, since it is either a constant or an annotated variable. Given a term containing constants, variables, abstractions, applications, and corresponding type

annotations, we can calculate the type of the term. The rules for this *type checking* are somewhat obvious, essentially checking that applied arguments match the annotated type, and so on. The first typing rule is for variables, (T-Var) in Table 4.2. There is a convention of using a *type environment* (or *context*) which is called Γ, consisting of a set of variables and corresponding types. The comma operator $\Gamma, \mathtt{x} : \mathtt{T}$ extends the environment with a new variable $\mathtt{x}$ and its type $\mathtt{T}$. The typing rules for abstractions and applications, (T-Abs) and (T-App), are also in Table 4.2. Evaluation is performed the exact same way in the typed calculus, and it is even possible to perform *type erasure* by removing everything type-related, since it is only used for type checking.

4.1.2 Type inference

In the above, explicit type annotations were described, but they are not strictly necessary. Type inference or reconstruction makes it possible to calculate the most general type, called the *principal type*, for a term with missing annotations. For example, the term $\lambda \mathtt{x}.\, \mathtt{x}\, \mathtt{1}$, which takes an argument $\mathtt{x}$ and applies it to the number $\mathtt{1}$, must obviously have a type of the form $(\mathtt{Nat} \to \mathtt{X}) \to \mathtt{X}$. Here, $\mathtt{X}$ is a *type variable*, which can take on the value of any type. Similarly to variables in terms, a *type substitution* is defined as a mapping from type variables to types. For example, the mapping $\sigma = [\mathtt{X} \mapsto \mathtt{Bool}]$ is applied to the previous type to obtain $\sigma((\mathtt{Nat} \to \mathtt{X}) \to \mathtt{X}) = (\mathtt{Nat} \to \mathtt{Bool}) \to \mathtt{Bool}$.

There are two views of type variables, which lead to different concepts. In the first view, any type can be substituted for a type variable. When concrete types are not substituted for type variables during typechecking, different types can eventually be substituted into the variables, and the term can be used in different type contexts. This is called *parametric polymorphism* and is described in the next section. In the second view, we instead ask if there even exists a substitution that makes a term well-typed. If it is possible to choose a substitution which makes the term well-typed, we will find the principal type that does so. This is type inference, and it can be described in two parts: constraint generation, and unification.

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \,|_{\varnothing}\, \{\}} \qquad \text{(CT-Var)}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2 \,|_{\mathcal{X}}\, \mathcal{C}}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}_1.\, \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2 \,|_{\mathcal{X}}\, \mathcal{C}} \qquad \text{(CT-Abs)}$$

Table 4.1: The constraint typing rules for variables and abstractions.

Constraint generation involves generating a set of equations that must necessarily be satisfied for any solution to the type inference. This is similar to type checking, except constraints are recorded instead of checked. For the sake of intuition, generating constraints amounts to generating equations between two trees, where leaves can contain constants, variables, or the slightly more complicated case of a subtree.

For example, the constraint typing rules for variables and abstractions are shown in Table 4.1. There are additional constraint typing rules for other parts of the language, such as applications and language extensions. The notation is a bit involved, but $\Gamma \vdash \mathtt{t} : \mathtt{T} \,|_{\mathcal{X}}\, \mathcal{C}$ should be read "term $\mathtt{t}$ has type $\mathtt{T}$ given context Γ when constraints $\mathcal{C}$ are satisfied." Further, the notation $|_{\mathcal{X}}$ is essentially a notice that type variables in each derivation must be freshly generated – they must be distinct from the type variables in other derivations. It might be easiest to read the constraint typing rules bottom-up, since in this direction, they describe the algorithm that produces $\mathtt{T}$, $\mathcal{C}$ and $\mathcal{X}$ such that $\Gamma \vdash \mathtt{t} : \mathtt{T} \,|_{\mathcal{X}}\, \mathcal{C}$, given $\mathtt{t}$ and Γ – furthermore, this algorithm does not fail, since it is always possible to generate the constraints.

Unification is a method for calculating the most general solution for the constraint set. For a constraint set $\mathcal{C} = \{\mathtt{S_i} = \mathtt{T_i}\}$, a substitution σ *unifies* an equation $\mathtt{S} = \mathtt{T}$ if the instances $\sigma\mathtt{S}$ and $\sigma\mathtt{T}$ are equal, and σ unifies $\mathcal{C}$ if it unifies every equation in $\mathcal{C}$. The algorithm is shown in Listing 4.1. Here, $FV(\mathtt{T})$ is the set of type variables used in $\mathtt{T}$, and it is used as an *occurs check*, ensuring that an infinite type isn't created by a substitution like $[\mathtt{X} \mapsto \mathtt{X} \to \mathtt{X}]$.

```
1  function unify(C)
2    if C = ∅ then []
3    else let {S = T} ∪ C' = C in
4      if S = T then unify(C')
5      else if S = X ∧ X ∉ FV(T) then unify([X ↦ T]C') ∘ [X ↦ T]
6      else if T = X ∧ X ∉ FV(S) then unify([X ↦ S]C') ∘ [X ↦ S]
7      else if S = S1 → S2 ∧ T = T1 → T2 then unify(C' ∪ {S1 = T1, S2 = T2}
8      else no solution exists
```

Listing 4.1: Unification

4.1.3 Polymorphism

The simply typed lambda calculus that was described can be too restrictive, since every type $\mathtt{T}$ is either a type constant or a function $\mathtt{T_1} \to \mathtt{T_2}$ composed of these. The second view of type variables, where they are held abstract, can be used to gain more flexibility. When using the same function in multiple parts of a program, it might be applied to different types. Even if the type of this function is a variable, conflicting constraints would be generated for the same type variable; instead, what we need is *universal quantification*. Then, for the identity function $\mathtt{id} = \lambda\mathtt{t}.\mathtt{t}$, we can say that it has the type $\forall\mathtt{X}.\,\mathtt{X} \to \mathtt{X}$. Here, $\mathtt{X}$ is called *generalized*, and it is important to be careful about which type variables are generalized. Commonly, a `let` expression is introduced of the form `let x = t1 in t2`, which can be used anywhere a term is expected. Allowing types to be generalized in functions defined by `let` expressions is called *let-polymorphism*, and when used in this manner in the lambda calculus it is often referred to as the Hindley-Milner type system.

Syntax

$\mathtt{t} ::=$		terms:
	$\mathtt{x}$	variable
	$\lambda \mathtt{x} : \mathtt{T}.\, \mathtt{t}$	abstraction
	$\mathtt{t}\, \mathtt{t}'$	application
	$\mathtt{let}\, \mathtt{x} = \mathtt{t}'\, \mathtt{in}\, \mathtt{t}$	let binding
$\mathtt{v} ::=$		values:
	$\lambda \mathtt{x} : \mathtt{T}.\, \mathtt{t}$	abstraction value
$\mathtt{T} ::=$		types:
	$\mathtt{X}$	type variable
	$\mathtt{T} \to \mathtt{T}$	function type
$\Gamma ::=$		contexts:
	$\varnothing$	empty context
	$\Gamma, \mathtt{x} : \mathtt{T}$	variable binding

Evaluation (call-by-value) $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\, \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\, \mathtt{t}_2'} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\, \mathtt{t}_2 \longrightarrow \mathtt{v}_1\, \mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda \mathtt{x} : \mathtt{T}_{11}.\, \mathtt{t}_{12})\, \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

$$\mathtt{let}\, \mathtt{x} = \mathtt{v}_1\, \mathtt{in}\, \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}_2 \quad \text{(E-LetV)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{let}\, \mathtt{x} = \mathtt{t}_1\, \mathtt{in}\, \mathtt{t}_2 \longrightarrow \mathtt{let}\, \mathtt{x} = \mathtt{t}_1'\, \mathtt{in}\, \mathtt{t}_2} \quad \text{(E-Let)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}_1.\, \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\, \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{let}\, \mathtt{x} = \mathtt{t}_1\, \mathtt{in}\, \mathtt{t}_2 : \mathtt{T}_2} \quad \text{(T-LetPoly)}$$

Table 4.2: Definition of the lambda calculus with Hindley-Milner type system, in which we construct DSLs and synthesize programs. Some simple extensions are left out, such as natural or floating point number values, which can be easily implemented depending on application. Note that while `let`-expressions are not synthesized in the presented method, types of functions in the DSL are generalized, and this is considered during synthesis.

4.2 Local search

The paper (Appendix A) defines the neighborhood structure, but I also describe some details here. A DSL $\mathcal{D}$ is defined in the typed lambda calculus, containing: a set of base types, constants associated with the base types, and functions annotated with base types or type variables. The DSL could be implemented in any way necessary, such as by extending the lambda calculus with additional features. The main benefit of the Hindley-Milner type system is obtained here, since the DSL can contain generically typed functions, while neighborhood generation takes advantage of the annotated types to limit the search space.

The neighborhood for a program P from the DSL $\mathcal{D}$ is obtained by exhaustively applying all possible *edits* at all paths of the AST of the program. An edit replaces a term with a newly generated term from the DSL, ensuring that the program remains well-typed. The set of all possible edits to a term is limited by a predicate, based on the newly generated term. The predicate used in the paper limits the size of the new term, resulting in a depth-limited search. It is important to note that "all paths" indeed means all paths, including paths to internal nodes of the AST. With this definition, all subtrees of the AST, including the entire program itself, are considered for edits. For this reason, the term to be replaced by an edit is temporarily added to the DSL, allowing the search an opportunity to construct a program not only by refactoring small parts of it, but also by reusing an existing term as part of something new.

4.2.1 Synthesizing function types

A perhaps surprising aspect of the method is that it does not synthesize lambda abstractions. This is not due to a theoretical limitation; abstractions could straightforwardly be synthesized by the presented method. To synthesize an abstraction when a function of type $\mathtt{T}_1 \to \mathtt{T}_2$ is requested, simply generate $\lambda \mathtt{x} : \mathtt{T}_1 . \mathtt{t}$ and recursively synthesize `t` with the requested type $\mathtt{T}_2$. However, in ad-hoc testing this was not found to produce useful programs, despite adding significantly to the neighborhood size.

Instead, functions can be synthesized from the DSL, using any available functions that can yield the requested type. This could either be an exact match, a polymorphic match (e.g. $\forall \mathtt{X}. \mathtt{X} \to \mathtt{X}$ can be used if $\mathtt{Bool} \to \mathtt{Bool}$ is requested), or even a partially applied polymorphic match: if the requested type is $\mathtt{Bool} \to \mathtt{Int}$, a function in the DSL with type $\forall \mathtt{X}. \forall \mathtt{Y}. \mathtt{X} \to \mathtt{Y} \to \mathtt{X}$ will work, by applying it to a single `Int` which can be recursively synthesized. While this approach was found to produce fewer, more relevant candidates, it seems unsatisfactory to completely disregard the synthesis of abstractions. Finding a way to synthesize useful abstractions is a challenging, but probably worthwhile, future endeavor. Synthesizing abstractions with generalized types, such as through `let` expressions, is probably more difficult but would take full advantage of the type system.

4.3 Distributed search

Since the local search algorithm is a very large tree search, obtained by viewing neighborhood enumeration as the application of edits to terms and then recursively to terms in the edits, it is possible to parallelize the search using strategies for searching trees. Naively, we could try to decompose the search by expanding the search tree up to some depth and then parallelizing across subtrees. Unfortunately, the search trees were observed in experiments to be very irregular; when decomposed in this way, a majority of valid programs were observed to belong to only one or a few

subtrees. What is needed is an algorithm that adaptively allocates computation to expand nodes, irrespective of their location in the tree. It's also possible to use an algorithm that adaptively searches through subtrees, namely the Distributed Tree Search algorithm in Ferguson and Korf (1988), which could be slightly more efficient by reducing overhead. The decomposition at a node level is more straightforward, however, and unlike the application to branch-and-bound as presented in that paper, deciding to evaluate a node does not depend on information in other parts of the search tree.

The algorithm used to parallelize neighborhood generation and search works by distributing *node expansion* across a set of processes, which can exist on the same physical machine, or across several physical machines on the same network. There is a main process which coordinates the search, and worker processes only need to be connected by the network to the master process. The search consists of node expansion and program evaluation, and both are distributed as follows. Given a program, the main process calculates all paths in the AST, and submits a job for each path. Worker processes that are idle can take an available job, with each job consisting of the algorithm in Listing 4.2. The function $editCandidates(s)$ generates the next level of the search tree from the current node s, by selecting all candidates from the DSL for which a constraint between the type of s and the type of the candidate can be unified. Since the node is only expanded one level, arguments in the candidates are not recursively synthesized yet, but are left as typed holes, corresponding to the nodes at the next level of the search tree. Programs without any holes are *complete*, and can be evaluated.

```
function expandAndEvaluate(s)
  leaves = []
  for ns ∈ editCandidates(s)
    if complete(ns)
      add ns to leaves
    else if depth(ns) < maxDepth
      create a new job expandAndEvaluate(ns)
    else
      skip ns
  evaluate leaves and return
```

Listing 4.2: Expand a node and evaluate programs

4.4 Discussion

As presented, the neighborhood structure is essentially a suggestion for further experimentation and research. To my knowledge, only some very basic experiments have been performed with the imitation-projection framework. Performing policy projection in the typed lambda calculus would be interesting, since it represents a major upgrade to the expressive power of the language compared to the PID controllers or decision trees that have been used previously. One challenge in this regard is to select

a task of suitable interest and complexity, and to design a corresponding DSL that is also interesting and expressive enough. The most interesting kinds of task/DSL combinations would be ones where with some sort of programmatic structure to the optimal solution, and where multiple different base types are involved in order to fully take advantage of the type system.

An aspect of the presented method that needs work is the synthesis of functions. I do believe that the method for synthesizing functions strictly using the DSL can be valuable, but also that the generation of new functions should not be ignored. Specifically, I imagine that it might not be very important to synthesize abstractions during a single iteration – not much reusable code can be discovered in just one iteration. Across iterations however, acquisition of new functions could play a more important role, by reducing the effort needed to generate similar concepts subsequently.

Several methods that were discussed in the survey treated the acquisition of new functions. While enumerating a neighborhood is another way to approach the search for programs, many aspects are at least somewhat independent of the search method. For example, an existing method for acquiring useful functions could likely be integrated, at least in between synthesis iterations. Other machine learning methods could also be integrated with the neighborhood search, such as learning a distribution over programs in the neighborhood, which could be used for enumeration and instead of the depth limit.

Finally, the idea of policy imitation needs to be discussed. It seems to me that there is fundamental problem in treating a neural policy as an oracle, notably in early stages of learning when it is practically guaranteed that no part of the policy is optimal. If the goal is to discover simple programs that are interpretable, or that generalize beyond the current performance of a neural policy, then it seems counterproductive to treat the neural policy as an oracle while attempting to imitate it as well as possible. Interactive imitation does not help; adding more data to overfit on only exacerbates the problem, and reinforcement learning algorithms tend to learn mostly on-policy, even if they are capable of learning from off-policy data. The experiments described in the paper did show a glimmer of hope, such as discovering the simple but performant program in iteration 4 of the neural imitation. Further iterations demonstrate the problem, though, by the program getting more complicated while fitting the neural policy increasingly well despite little gain in performance. Based on this, it seems likely that the problem should be framed differently, such that whatever program synthesis algorithm is used, it is not rewarded for fitting arbitrarily bad parts of a supposed oracle. Neighborhood search does not strictly depend on imitation learning; any approach where a very large number of programs can be evaluated at low cost can be used, such as learning a cheap model of the environment in which programs are evaluated.

CHAPTER 5

Paper 2: Reinforcement learning of causal variables

This work differs a bit from other topics considered in this thesis, since no program representations are involved. Nonetheless, causality and models thereof are similar to programs, in that simple but structured representations can contribute in important ways to the learning of intelligent behavior. There is a well-known discrepancy between the noisy, limited observations humans receive about the world, and the powerful, general models humans nonetheless manage to construct to guide their behavior (Tenenbaum et al., 2011). In reinforcement learning, this discrepancy can be seen as the difference between learning at the level of individual states of an MDP, and acquiring abstract knowledge which can be applied across states or even across MDPs. There is probably only one way to actually learn generalizations *beyond* observed data: obtain additional information from another source. One approach to learning abstract knowledge in reinforcement learning is state abstraction, which has been researched under the guise of function approximation since the inception of the field. In its basic form, the purpose of function approximation in reinforcement learning is to allow learning to scale to large state spaces, for which tabular representations are unrealistic. This makes function approximation of some kind practically necessary when applying reinforcement learning to larger, more realistic tasks.

Of more relevance here is the abstract knowledge dimension of function approximation. Whether it is simple state aggregation, or a value function represented by a deep neural network, the state space is abstracted. The potential benefits and disadvantages are well known; for example, although observed data can be used to update many similar states that are grouped together, this can also destabilize learning if the update does not fit some of the grouped states. There is also an obvious connection to hierarchical reinforcement learning, where, for example, abstracted states become

states in a semi-MDP related to the original MDP (R. S. Sutton, Precup, and S. Singh, 1999).

This chapter is based on work done for the paper "Reinforcement learning of causal variables using mediation analysis", and a draft of the paper can be found in Appendix B. I worked on this paper with Tue Herlau, who had been working on the idea for a while before we collaborated on completing it. The main contribution of the paper is an approach to learning an abstract state variable, which is in a causal relationship with the outcome (i.e. the reward). The variable can be learned online using a stochastic approximation algorithm, while simultaneously training a standard reinforcement learning policy to reach the causal state.

5.1 Direct and indirect effects of policies

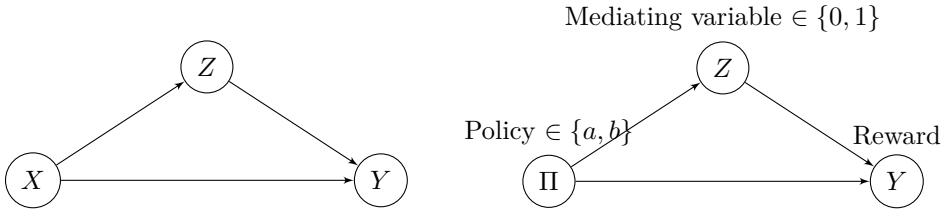
The basis for this work is the definition of direct and indirect effects by Pearl (2001). That paper also defines more general path-specific effects, where only a selected set of paths instead of nodes remain active, but the conditions for identification are much stricter than in the case of the direct and indirect effects. Specifically, we consider the simplest possible mediation model, containing the paths $X \to Y$ and $X \to Z \to Y$, as seen in Figure 5.1a. In this model, the direct effect of X on Y quantifies how changing X changes Y only through the path $X \to Y$, while the indirect effect quantifies how changing X changes Y only through the path $X \to Z \to Y$. The natural indirect effect (NIE) in this model, if X had the value x' instead of x, is (Pearl, 2001)

$$\mathrm{NIE}_{x'}(Y) = \sum_z \mathbb{E}[Y|x,z]\left(P\left(z|x'\right) - P\left(z|x\right)\right). \tag{5.1}$$

In order to use this model, we take the view shown in Figure 5.1b. Thus, $X = \Pi$ denotes the choice of a policy, $Y = G_0$ is the return (2.5), and Z is some unknown variable that can be identified in the underlying MDP. To keep it as simple as possible, we consider two policies $\Pi = \{\pi_a, \pi_b\}$, and Z is binary. The interpretation of Z here is that it is an indicator of whether some underlying condition has been made true in the state space of the underlying MDP, and for that reason we define it as a stopped process,

$$Z = \max\left(Z_0, Z_1, \ldots, Z_T\right), \tag{5.2}$$

where $Z_t = \{0, 1\}$ for $t = 0, 1, \ldots, T$. Hence, $Z = 1$ if, during an episode, the condition defined by the Z_ts happens. In this simple case, where the variables are binary, the



(a) Simple mediation model. (b) Binary policy mediation model.

Figure 5.1: The mediation model used in the paper.

NIE simplifies as

$$
\begin{aligned}
\mathrm{NIE}_{x'}(Y) = & \quad \mathbb{E}[Y|Z=0,\pi_a]\left(P\left(Z=0|\pi_b\right)-P\left(Z=0|\pi_a\right)\right) \\
& + \mathbb{E}[Y|Z=1,\pi_a]\left(P\left(Z=1|\pi_b\right)-P\left(Z=1|\pi_a\right)\right) \\
= & \quad \mathbb{E}[Y|Z=0,\pi_a]\left(\left(1-P\left(Z=1|\pi_b\right)\right)-\left(1-P\left(Z=1|\pi_a\right)\right)\right) \\
& + \mathbb{E}[Y|Z=1,\pi_a]\left(P\left(Z=1|\pi_b\right)-P\left(Z=1|\pi_a\right)\right) \\
= & \quad \mathbb{E}[Y|Z=0,\pi_a]-\mathbb{E}[Y|Z=0,\pi_a]P\left(Z=1|\pi_b\right) \\
& - \mathbb{E}[Y|Z=0,\pi_a]+\mathbb{E}[Y|Z=0,\pi_a]P\left(Z=1|\pi_a\right) \\
& + \mathbb{E}[Y|Z=1,\pi_a]\left(P\left(Z=1|\pi_b\right)-P\left(Z=1|\pi_a\right)\right) \\
= & \quad \mathbb{E}[Y|Z=0,\pi_a](P\left(Z=1|\pi_a\right)-P\left(Z=1|\pi_b\right)) \\
& + \mathbb{E}[Y|Z=1,\pi_a]\left(P\left(Z=1|\pi_b\right)-P\left(Z=1|\pi_a\right)\right) \\
= & \left(\mathbb{E}[Y|Z=0,\pi_a]-\mathbb{E}[Y|Z=1,\pi_a]\right)(P\left(Z=1|\pi_b\right)-P\left(Z=1|\pi_a\right)). \quad (5.3)
\end{aligned}
$$

The expression in (5.3) is a bit easier to interpret. The first term of the product is the change in expected outcome, keeping X at the reference level x while changing Z from $z=0$ to $z'=1$, while the second term is the change in probability of seeing $Z=1$ when changing X from the reference of $x=\pi_a$ to $x'=\pi_b$. Thus, the NIE has the property that it attains a large value only when 1) observing z' leads to a larger reward *and* 2) the policy π_b is able to act such that z' is more consistently achieved than under π_a. This is the motivation for using the NIE, since the right choice of Z (read: that attains a large value of the NIE) should be an identifiable condition in the MDP that is both useful and achievable.

5.2 Optimizing the indirect effect

Our goal is to discover a useful mediating variable Z, which we find by maximizing the NIE with respect to Z. To do so, we define the distribution of Z as Bernoulli($\Phi(s)$) using the parameterized function $\Phi(s)$. Thus, for each state s in the MDP, $\Phi(s)$ is the probability of z'. Additionally, we define the combined policy

$$\pi(s_t) = \begin{cases} \pi_a, & \text{if } \Pi = a \\ (1 - Z_{0:t})\pi_b + Z_{0:t}\pi_a, & \text{if } \Pi = b, \end{cases} \tag{5.4}$$

where $Z_{0:t} = \max\{Z_0, \dots, Z_t\}$. This choice is made in an attempt to decompose the trained behavior of π_b, which is discussed below.

Learning Z by optimizing the NIE can be made pretty similar to standard reinforcement learning, by introducing recursions similar to the Bellman equation (2.10) which converge to the terms in (5.3). First, we need the value of Z from time t, $Z_t^\infty = \max\{Z_t, Z_{t+1}, \dots\}$, while using the convention $Z_t = 0 \quad \forall t \notin \{0, \dots, T\}$, and we can then define the "value functions"

$$v_t^\infty(s_t) = P(Z_t^\infty = 1 | S_t = s_t, Z_{t-1} = 0), \tag{5.5}$$

$$v_t^z(s_t) = \mathbb{E}\left[G_t | S_t = s_t, Z_t^\infty = z, Z_{t-1} = 0\right], \tag{5.6}$$

which are both conditional on $Z_{t-1} = 0$. These should be interpreted as follows, starting from a state s_t: v_t^∞ is the probability that Z happens in the future given that it hasn't happened yet, and v_t^z is the expected return given that Z has not happened yet, and that it either will ($z = 1$) or won't ($z = 0$) in the future. That is, v_t^z is actually two value functions v_t^1 and v_t^0. Importantly, $v_0^z = \mathbb{E}\left[G_0 | Z = z, s_0\right]$ and $v_0^\infty = P(Z = 1 | s_0)$ correspond to terms in (5.3).

The value functions satisfy the recursions

$$v_t^\infty(s_t) = \Phi(s_t) + \bar{\Phi}(s_t)\, \mathbb{E}\left[v_{t+1}^\infty(S_{t+1}) | s_t\right], \tag{5.7}$$

$$v_t^1(s_t) = \frac{v_\pi(s_t)\Phi(s_t)}{v_t^\infty(s_t)} + \frac{\bar{\Phi}(s_t)}{v_t^\infty(s_t)}\, \mathbb{E}\left[v_{t+1}^\infty(S_{t+1})\left(R_{t+1} + \gamma v_{t+1}^1(S_{t+1})\right) | s_t\right], \tag{5.8}$$

$$v_t^0(s_t) = \frac{\bar{\Phi}(s_t)}{v_t^{\bar{\infty}}(s_t)}\, \mathbb{E}\left[v_{t+1}^{\bar{\infty}}(S_{t+1})\left(R_{t+1} + \gamma v_{t+1}^0(S_{t+1})\right) | s_t\right], \tag{5.9}$$

where $\bar{x} = 1 - x$ and $v_\pi(s_t)$ is the value function as usually defined (2.6). The supplementary material in Appendix B contains a proof of the corresponding Bellman operators being contraction mappings.

These recursions all have the familiar form

$$v_t(s_t) = \mathbb{E}_\mu\left[H_t(s_t, S_{t+1}) + G_t(s_t, S_{t+1}) v_{t+1}(S_{t+1}) | s_t\right], \tag{5.10}$$

where the expectation is over the observed behavior generated by a policy μ. These updates could be used directly, but practically we chose to use the n-steps V-trace

```
input: initial policy $\pi_a(s)$ and $\pi_b(s)$
input: initial value $v_\pi(s)$, $v_\pi^\infty(s)$, $v_\pi^1(s)$, and $v_\pi^0(s)$
input: initial causal variable $\Phi(s)$
repeat
  collect experience using $\pi_a$
  train $\pi_a$ using e.g. an actor-critic method
  train $\pi_b$ using the reward from (5.11)
  train value functions $v_\pi(s)$, $v_\pi^\infty(s)$, $v_\pi^1(s)$, and $v_\pi^0(s)$ using (5.7)
  train causal variable $\Phi(s)$ by maximizing (5.3)
until convergence
```

Listing 5.1: Learning a causal mediating variable

target (Espeholt et al., 2018), which should lead to better off-policy updates when sampling experience from a replay buffer.

To estimate the NIE, we parameterize $v_\pi(s)$, $v^\infty(s)$, $v^1(s)$, and $v^0(s)$. Then, the NIE is obtained by plugging the relevant values into (5.3). The parametrized Z (i.e. Φ) can be trained to maximize the NIE, for example using gradient ascent. The policy π_b is trained directly to maximize $P(Z = 1)$, but $P(Z_t)$ is multiplicative across steps. In order to obtain an additive value that can be used directly as a reward, we decompose it using a stick-breaking construction,

$$r_{t+1}^b = \Phi(s_t) \prod_{k=0}^{t-1} (\bar{\Phi}(s_k)), \tag{5.11}$$

which results in $\sum_{t=0}^{\infty} r_{t+1} = P(Z = 1|\tau)$ where τ is any trajectory of states and actions. The overall approach is described algorithmically in Listing 5.1. Results from some basic settings are included in the draft of the paper in Appendix B.

5.3 Discussion

The presented method is an alternative way of learning a simple, abstracted state variable, which isn't a latent part of a generative model, but instead is in a descriptive relationship to some underlying phenomenon. This results in a qualitatively different model, by making a different set of *ontological commitments* (Davis, Shrobe, and Szolovits, 1993). As mentioned in the introduction, it is necessary to obtain additional information beyond observed data, in order to also generalize beyond it. Our approach obtains a bit of extra information through a simple causal graph, which allows the discovery of a mediating variable that is both useful and achieveable according to the NIE.

There is some relation between this work and work on dynamic treatment regimes in medicine. The field of optimal dynamic treatment regimes is related to reinforcement learning in general, and Q-learning is commonly applied in this field, even on observational data (Schulte et al., 2014). Further, work on optimizing a dynamic

treatment regime only with respect to a path-specific effect instead of the overall outcome (Nabi, Kanki, and Shpitser, 2018) is similar to optimizing the indirect effect. To be clear, the setting here is quite different since learning is performed online, on a larger problem, while the mediating variable is simultaneously learned. The main similarity is thus the learning of a policy, but in our work this is achieved with reinforcement learning on a reward defined by using the mediating variable.

Hopefully, learning such a descriptive causal variable can be useful in learning intelligent behavior. Mediation analysis is popular in fields such as medicine and psychology, where it can be used to gain additional insight into interventions (Whittle et al., 2017; Agler and De Boeck, 2017). At minimum, an intermediate causal state can act as an interpretable goal by which an MDP can be decomposed. Several such abstracted states can facilitate planning, in a fashion similar to hierarchical reinforcement learning (Silver and Ciosek, 2012; Bagaria and Konidaris, 2020). Extending the method to discover multiple causal states would be interesting, and would perhaps help in making the approach more practically viable. There are several ways in which multiple causal states could be incorporated, for example as sequences of causal states obtained by iteratively discovering a new causal state using the previously discovered one as the new outcome.

An aspect of the method that needs further research is the training procedure itself. While running experiments for the paper, we found that it could be difficult to learn some of the functions depending on parameterization and other tricks. These should be mostly described in the paper, such as not directly optimizing (5.3), but instead optimizing an alternate form of it which is also regularized. When published, the code will be available as well, and all the more minor things that might not be mentioned in the paper can of course be found there.

Finally, a critical aspect of the method is that the optimal mediating variable cannot be identified if π_a is optimal. This fact can be understood by considering that an optimal mediating variable Z by definition is highly associated with the optimal outcome $\mathbb{E}[Y|Z=1]$. If π_a is optimal, it already optimizes $P(Z=1|\pi_a)$ because it is needed to achieve the optimal outcome. As discussed in the paper, depending on the perspective we choose to take on causality, this could be seen as a natural consequence of how we define causality. When considering an MDP at the lowest level, it is difficult to consider anything as a cause except the actions that lead to the following states. Similarly, at a molecular level it would not be straightforward to link smoking to cancer; the immediate low-level cause of cancer would be, for example, the particular chemical interaction that causes a genetic mutation, transforming a normal cell into a cancer cell. Only at a much higher, abstracted level can we call this cause "smoking". Considerations such as this are not just philosophical, but have a direct influence on how we can apply causal principles, as this issue demonstrates. Interestingly, the very basics of causality are still being discussed by philosophers (B. Russell, 1912; Ross and Spurrett, 2007).

CHAPTER 6

Discussion

The presented research tackles the key features of intelligent agents that were identified in the introduction. Programmatic policies are extracted from existing neural policies by applying multiple steps of a greedy search algorithm over a neighborhood structure defined in a typed programming language. Specifically, the neighborhood is defined based on a given program and domain specific language (DSL), and contains all the well-typed programs obtained by substituting terms from the DSL for all possible sub-terms of the program. Since a DSL can generate an unlimited number of larger and larger terms, the size of the neighborhood is limited by, for example, considering only small terms from the DSL and a small number of simultaneously substituted terms. Despite these limits and the greedy search, it was demonstrated that the method can discover useful programmatic policies that are too complicated to discover in one iteration of search. Even when the required program contains discrete structures that cannot be found in one step, it is at least sometimes possible to discover them. For example, an `if`-expression with branches that are too complicated to be contained in a single neighborhood, can be discovered in two ways: by first discovering the code for the branches before finding the `if` and condition in a subsequent step, or alternatively by finding an `if` with simplified terms in the branches first. Interestingly, both of these approaches to the synthesis problem come naturally from the definition of the neighborhood.

Abstract causal state variables are learned by maximizing the causal effect of a policy on the reward through the state variable. That is, the optimal function mapping states to an abstracted variable is identified. By optimizing this so-called indirect effect of the policy, the abstract variable corresponds to a situation which is causally linked to the agent's reward function, while simultaneously being something that the agent can obtain through its actions. We argue that this interpretation of the indirect effect is qualitatively different from other ways of learning state abstractions, such as predictive latent variable models, since it is not associated with a generative model of the world. For example, in a simple setting where an agent has to pick up a key to open a door before obtaining a reward, it was demonstrated that the method identified the key as a causal variable, despite obtaining no reward from it directly.

6.1 Perspective and future work

6.1.1 Programmatic policy search

Extracting programmatic policies from neural policies is a straightforward way to attain composition while using standard, state of the art reinforcement learning algorithms. Programs discovered in this way are communicable, and have several other potential advantages to neural policies: for example, simple programs with a good inductive bias are likely to generalize better to states that have not been observed during training, and they are cheaper to evaluate and use at test time. Of course, these advantages are by no means free to obtain; designing a good DSL is, as the name suggests, domain specific and must currently be done by a human expert. Additionally, searching through a large program space is computationally expensive, and happens after an also expensive reinforcement learning phase. This last point was discussed in the paper, and it is not strictly the purpose of the method to be used after reinforcement learning. Rather, the point is to jointly learn a neural and a programmatic policy. For example, programmatic policy extraction could be performed at some interval during reinforcement learning, while first making sure that the neural policy behaves well on the chosen imitation states. As a result, structured policies that generalize could be discovered in early stages of learning, taking advantage of inductive bias in the DSL.

There is no reason to expect that the search can generally find the right program; local optima are always a main consideration in local optimization. Due to the greedy search, it is necessary that useful structures can be discovered within the limits of the neighborhood iterations, if they are to be discovered at all. When and how this is possible should be examined further, in order to maximize the potential of program search. One point, perhaps a bit obvious, is that the search depth as defined must be at least the maximum arity of functions in the DSL. Even then, this results in the function(s) with highest arity only being synthesized with simple constants or variables as arguments. This is merely due to the way the search depth is limited, and could be changed; the question is simply how. One approach could be to train a probabilistic model over edits, and enumerate all programs above a given probability. However, one of the benefits of the neighborhood as defined is that no model is trained, which could be expensive in itself, and a different model would be needed for each DSL, and probably even for each setting that the same DSL is applied to.

6.1.2 Causal state abstraction

Learning a causal state abstraction seems like a good fit for the kind of abstraction that was discussed in the introduction. The purpose of such an abstraction is to not only ignore irrelevant aspects of the world (relative to the current task), but also to act as a concept that can be communicated, and that can be used in a wide variety of scenarios. Since causal states are linked to outcomes, and can be manipulated by an

agent, it seems that they would also be useful for other agents. In terms of generalization to unseen states, however, the method still depends on a good parameterization of the function mapping states to abstract states. It would be interesting to explore how well the causal state representation can actually generalize in commonly used environments, and to explore ways in which this generalization could be improved.

6.1.3 Combined approaches

Ideally, an agent would be able to compose skills, communicate with others, and abstract skills and knowledge. The presented methods address these three aspects, but not jointly. The programmatic policies operate with a small set of input variables, and although a DSL could be designed to handle high-dimensional perceptions, this seems less than ideal. Instead, it would be useful if program inputs could be discovered with a method such as causal state abstraction. One possibility is to search directly for programmatic policies that maximize the indirect effect as described. One concern in this regard is that abstracted state variables are not useful for low-level control. Programs with abstracted state inputs would essentially operate at a different level of the skill hierarchy, and a different representation, or at least an entirely different set of inputs, would have to be used for low-level policies. From my perspective, this modularity could be seen as a strength of the approach – low-level control policies are likely to vary between different agents, and only programs operating over abstract, grounded variables make sense across agents and situations.

APPENDIX A

Draft of paper 1

This paper, with the tentative title “Programmatic policy extraction by iterative local search”, is at the time of writing under review for the AAIP workshop at IJCLR 2020-21.

Programmatic policy extraction by iterative local search

Rasmus Larsen and Mikkel N. Schmidt

Department of Applied Mathematics and Computer Science,
Technical University of Denmark.

Abstract

Reinforcement learning policies are often represented by neural networks, but programmatic policies are preferred in some cases because they are more interpretable, amenable to formal verification, or generalize better. While efficient algorithms for learning neural policies exist, learning programmatic policies is challenging. Combining imitation-projection and dataset aggregation with a local search heuristic, we present a simple and direct approach to extracting a programmatic policy from a pretrained neural policy. After examining our local search heuristic on a programming by example problem, we demonstrate our programmatic policy extraction method on a pendulum swing-up problem. Both when trained using a hand crafted expert policy and a learned neural policy, our method discovers simple and interpretable policies that perform almost as well as the original.

1 Introduction

While neural policy representations are by far the most common in modern Reinforcement Learning (RL), other representations are worth considering. Programmatic policies provide a number of potential benefits: For example, a program might be read and understood by a human, something that generally is not possible with a neural network. Programs are also inherently compositional, which allows for not only reuse of policies in new combinations, but also compositional reasoning about their behavior.

However, learning programmatic policies is challenging. The structured, discrete space of programs does not allow for the gradient based optimization that neural policies benefit greatly from. Compared to a more standard inductive synthesis setting, programmatic policies must be evaluated in an environment that, whether simulated or real, is expensive to interact with. Several approaches exist that attempt to handle this interaction issue, such as learning a parametric environment model (Hein et al., 2017), imitating an existing policy (Bastani et al., 2018; Verma et al., 2018), or evaluating fewer programs by learning to search more efficiently (Ellis et al., 2018). Furthermore, Verma et al. (2019) extend the imitation setting by providing a framework for intertwining RL and programmatic policy imitation.

This imitation-projection framework brings us a step closer to programmatic RL, where programs can be learned gradually through interaction with the environment. Essentially, this allows similar sample efficiency when compared to policy gradient methods, since the imitation-projection step is performed offline by scoring programs according to an imitation learning objective. One could even plausibly imagine that the inductive bias in a problem-specific policy language could lead to improved learning.

The framework leaves many choices open in terms of how the policy update and programmatic policy projection steps are performed, as well as in terms of defining the program space. Verma et al.

(2019) perform experiments with a specific choice of update and projection, using two tailored program spaces based on PID controllers with either decision tree regression or Bayesian optimisation over some parameters as the projection operator.

In this paper we experiment with a more general program space based on Domain Specific Languages (DSLs) implemented in a typed lambda calculus. We demonstrate a method for re-using projections by local search around a previous projection, potentially reducing the required computational effort while allowing much longer programmatic policies to be found. Since imitation-projection greatly reduces environment interaction, the presented method takes advantage of this and performs relatively large searches in program space. Demonstrating the method on the pendulum swing-up task, we show that a simple and effective programmatic policy can be found by imitating a learned neural policy.

2 Methods

Our framework is based on previous work on imitation-projected and programmatically interpretable reinforcement learning (Verma et al., 2018, 2019). In contrast to previous work, our program space is a DSL defined in a general typed lambda calculus, that allows us to do program synthesis by a local type-directed search.

2.1 Program synthesis by type-directed search

In this paper, we represent programs in the polymorphic lambda calculus, also called System F (Pierce, 2002). This choice allows for quite expressive programs, without requiring too complicated code for defining, executing or synthesizing them. One especially useful feature of this program representation is the type system, which can reduce the space of programs to be searched. While other program representations could also be used, a powerful type system makes the program search more scalable, by pruning candidate subtrees that do not type check. Using this representation, DSLs can be defined as a set of typed functions and constants, which together represent the space of possible programs to be searched.

Our starting point for program synthesis is enumerative search. The space of all programs in a DSL forms a tree, with the empty program at the root. Internal nodes are partial programs, with each branch being a candidate substitution for a hole in a partial program. Enumerating through this search tree results in generating all valid programs, according to the DSL grammar, as the leaves of the tree. Since the search tree for most DSLs will be infinite, we limit the search by defining a stop criterion for the search. Here, we choose the stop criterion to be a maximum search depth, resulting in a depth-limited search algorithm.

We can take advantage of the typed language to reduce the search space. Instead of yielding all syntactically valid programs, as explained above, we want to yield only well-typed programs. The enumerative type-directed search algorithm is the same as used by e.g. Ellis et al. (2018) to draw programs from a prior distribution, but it does not include probabilities over the grammar productions. To expand a node in the search tree, a typed hole (an empty program with a type annotation) in the corresponding partial program is selected for synthesis. Then, valid candidates are selected from the set of all DSL candidates by type unification; the resulting context of the type unification is propagated to the following synthesis, ensuring that any constraints are satisfied. All candidates that can produce the correct type are considered, even if they would need arguments applied to them first.

Algorithm 1 Iterative local programmatic policy imitation

input: oracle policy f
optional input: initial program $p_{init} = \varnothing$
output: program p_T
collect N on-policy trajectories using f:
  $\tau_0 = \left(\left(s_0^0, f(s_0^0), s_1^0, f(s_1^0), \dots\right), \dots, \left(s_0^N, f(s_0^N), s_1^N, f(s_1^N), \dots\right)\right)$
create supervised dataset $\Gamma_0 = \{(s, f(s)) \,|\, s \in \tau_0\}$
derive p_0 from Γ_0 by local search from p_{init} // algorithm 2
for $i = 1, \dots, T$
  collect M on-policy trajectories using p_{i-1}:
    $\tau_i = \left(\left(s_0^0, p_{i-1}(s_0^0), s_1^0, p_{i-1}(s_1^0), \dots\right), \dots, \left(s_0^M, p_{i-1}(s_0^M), s_1^M, p_{i-1}(s_1^M), \dots\right)\right)$
  create supervised dataset $\Gamma' = \{(s, f(s)) \,|\, s \in \tau_i\}$
  aggregate datasets:
    $\Gamma_i = \Gamma_{i-1} \cup \Gamma'$ // or $\Gamma_i = \Gamma_0 \cup \Gamma'$, which is cheaper
  derive p_i from Γ_i by local search from p_{i-1} // algorithm 2
end

Algorithm 2 Depth-limited local search (typed neighborhood)

input: domain specific language $\mathcal{D}$
input: imitation dataset Γ
input: initial program P
output: best program in typed neighborhood p^*
function $N_n^d(\mathcal{D}, P, l)$ // generates the neighborhood for location l in P
  return $\varnothing$ **if** d = 0
  $T = \text{type}(P, l)$ // type of expression at l
  $C = \{e \,|\, e : t \in \mathcal{D} \wedge T \text{ can unify with yield}(t)\}$ // everything valid from DSL
  $P' = \{\text{edit}(P, l, c)) \,|\, c \in C\}$
  // return all complete programs, and recursively generate remaining partial ones
  return $\{p \in P' \,|\, p \text{ is complete}\} \cup N_n^{d-1}(\mathcal{D}, p', l') \quad \forall p' \in \{p' \in P' \,|\, p' \text{ is partial}\}$
    where l' is the location of the first hole in p'
end
$p^* \leftarrow \varnothing, v^* \leftarrow \infty$ // best program and imitation loss
foreach $l \in$ set of all paths in P
  $E_l = \text{expression}(P, l)$ // expression at location
  $\mathcal{D}' = \mathcal{D} \cup (E_l, \text{type}(E_l))$ // extend DSL
  foreach $p \in N_n^d(\mathcal{D}', P, l)$
    evaluate p on Γ and update p^* and v^*
  end
end

2.2 Local policy imitation

We frame the policy synthesis problem as imitation learning. Like previous policy imitation methods, an interactive dataset aggregation method such as DAgger (Ross et al., 2010) is used: Instead of imitating only on states that the expert experiences, which is called behavioral cloning, some experience from the imitation policy is periodically added to the set of states considered. This allows subsequent imitation iterations to correct mistakes that otherwise wouldn't be observed, since the expert policy never experiences these mistakes. However, it is possible that the expert also makes mistakes on states that are not usually observed, and for this reasons it is not always a clear benefit. The iterative imitation approach is described with pseudocode in alg. 1.

The described iterative imitation algorithm can employ different program synthesis methods. Basic enumerative search, as described in section 2.1, searches through a tree with the empty program at the root. To construct an enumerative search that is able to synthesize much larger programs, while also benefiting from work performed in previous iterations of the search, we propose a local search heuristic. Here, the local search is defined by a neighborhood around a given program,

Algorithm 3 Imitation-Projected Programmatic Reinforcement Learning with Local Synthesis

```
input: initial policy π_0
optional input: initial program p_0 = ∅
output: trained policy π_T, program p_T
for t = 1, ..., T
  π_t ← Update(π_{t−1})   // reinforcement learning, e.g. policy gradient
  p_t ← Project(π_t, p_init = p_{t−1})   // program synthesis by algorithm 1
end
```

more specifically by a parametric neighbourhood based on a tree edit operation.

Define the edit operation $\texttt{edit}(P,\, l,\, P')$ as replacing the subexpression at location l in program P with the expression P'. Given a typed DSL $\mathcal{D}$, containing functions, constants, and their (potentially polymorphic) types, the neighborhood of the program P at location l is the set of programs obtained by generating all well-typed expressions P' contained in $\mathcal{D}$, written $N_n^d(\mathcal{D},\, P,\, l)$. The definition of a location is the root-to-expression path in the abstract syntax tree (AST) of the program. Here, we use the concept of a location in a generalized manner that can encompass multiple simultaneous locations, that is, a location l can represent multiple paths in the AST that are to be simultaneously replaced with independent expressions. The neighborhood of a program P is thus the union of the neighborhoods at all locations, $N_n^d(\mathcal{D},\, P) = \bigcup_{l \in L(P)} N_n^d(\mathcal{D},\, P,\, l)$, where the neighborhood has been parameterized with a maximum depth d of the replacement expressions, and with the number of simultaneous edits n.

Furthermore, in practice, the expression being edited is dynamically added as a candidate to the DSL, and for the depth evaluation this candidate counts as having a depth of 1. This allows an edit not just to replace an expression, but to also extend an expression by using it as part of the new expression despite the result being too large otherwise. The size of the neighborhood $|N_n^d(\mathcal{D}, P)|$ is quite sensitive to all involved parameters $\mathcal{D}$, P, n, and d, but these can be flexibly chosen based on the problem and available computational resources. The neighborhood search algorithm is given as pseudocode in alg. 2.

One purpose of this search algorithm is to fit into the full imitation-projection framework from Verma et al. (2019). A simple, modified version of this framework is shown in alg. 3. The difference consists of the projection step, which now also depends on the previous projection.

3 Experiments

We present three different program synthesis experiments: The first is a programming by example (PBE) task with sampled ground truth programs, demonstrating the efficacy of our local search heuristic. The second is a policy extraction task where the ground truth is a hand-coded policy. In these two first experiments, the DSL used for the search contains the true program used to generate observational data. Finally, in our third experiment we examine if we can learn learn a simple yet effective policy by imitation from a more complicated neural network policy which is trained using an existing reinforcement learning algorithm.

3.1 Programming by example with local program search

As a first evaluation of the method, we used a straightforward PBE task, whose purpose is to show that the described iterative local search is capable of synthesizing nontrivial programs from input-output specifications.

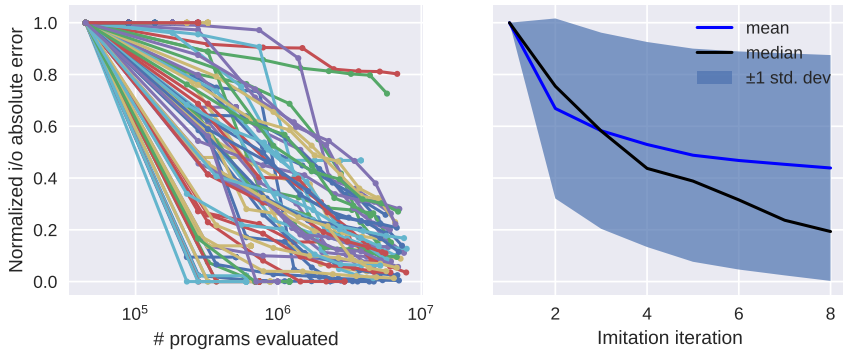


Figure 1: Experiment on 100 sampled programs. Left: For each program, the absolute error (normalized wrt. first iteration) by number of programs evaluated. Right: Mean, median, and standard deviation by search iteration.

DSL: We defined a language containing the set of constants $\{-1, 0, 0.5, 0.8, 1, 3, 5, 6, \texttt{true}\}$, which are all `Floating` point numbers except `true` which is `Boolean`, and the functions with associated type signatures $\{\texttt{if} : \texttt{Bool} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0}, > : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Bool}, \wedge : \texttt{Bool} \Rightarrow \texttt{Bool} \Rightarrow \texttt{Bool}, \oplus : \texttt{Bool} \Rightarrow \texttt{Bool} \Rightarrow \texttt{Bool}, - : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}), * : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}, \cdot^2 : \texttt{Float} \Rightarrow \texttt{Float}\}$.

Data: The observation space (i.e. input) to these programs consists of three `Floats`, which are distinct variables that can be used just like constants. 10 sets of these numbers were randomly sampled as inputs to be used during synthesis. Ground truth programs of some length, as a simple proxy for complexity, were sampled from a weighted distribution over the DSL. In order to obtain samples that have a reasonable length, we designed a distribution on the abstract syntax of our DSL that puts more probability on higher-arity functions. Further, the probability of sampling `true` was weighted significantly down, while the probability of sampling an input variable was weighted higher. Since program length is not the best measure of complexity, samples were rejected on other criteria too. Programs were discarded if: the length of the program (number of tokens) was less than 8, program output was constant, or an input-output equivalent program, on some randomly chosen inputs, existed within a depth 4 search of the DSL.

Results and discussion: Results from $d = 4$ local search on 100 sampled ground truth programs can be seen in fig. 1, which shows that for many of the programs an exact fit is found on the given inputs. Even for programs where an exact solution is not found, most of the searches show significant progress through the iterations, although a few make no progress at all. Since the search is deterministic, if no improvement is made in an iteration, further iterations will not lead to better results. It should also be noted that a single iteration of search with $d = 5$ in this setting corresponds to evaluating about as many programs as 20 iterations with $d = 4$.

3.2 Imitation of a programmatic pendulum swing-up policy

Next we examined if we were able to discover a ground truth programmatic policy by imitation learning.

Task: We based the experiment on a simple, classical control problem, the pendulum swing-up task. The state space consists of the angle and angular velocity of the pendulum, and the

action space is the torque applied to the base of the pendulum, normalized to the interval [-1, 1]. This two-dimensional state space allows us to easily display and visually compare policies. The simulator is discretized with a time step of 0.05s, and an episode is 200 steps long. The reward function is $r(\theta, \dot{\theta}, a) = ((\theta + \pi \pmod{2\pi}) - \pi)^2 + 0.1\dot{\theta} + 0.001a^2$, which results in a reward of 0 if the pendulum is perfectly balanced with no torque applied, and a reward of $-\pi^2$ when the pendulum is pointing straight down while not moving. While the state space of the pendulum task is $(\theta, \dot{\theta})$, the observation space supplied to policies is $(\sin\theta, \cos\theta, \dot{\theta})$.

DSL: We used a simple DSL with primitives suitable for solving the RL task, containing the constants $\{-6, -1, 1, 0.5, 0.6, 8, 10\}$, and the functions $\{$`if` : `Bool` $\Rightarrow$ `T0` $\Rightarrow$ `T0` $\Rightarrow$ `T0`, $>$: `Float` $\Rightarrow$ `Float` $\Rightarrow$ `Bool`, $-$: `Float` $\Rightarrow$ `Float` $\Rightarrow$ `Float`, $+$: `Float` $\Rightarrow$ `Float` $\Rightarrow$ `Float`, $*$: `Float` $\Rightarrow$ `Float` $\Rightarrow$ `Float`, `sign` : `Float` $\Rightarrow$ `Float`, $\cdot^2$: `Float` $\Rightarrow$ `Float`$\}$.

Ground truth: We hand crafted a ground truth policy in the DSL, capable of swinging up the pendulum from any starting state. The policy achieves an average reward of approximately -211 and a maximum of -113.

Synthesis: At each iteration of the local search we used a search depth of $d = 4$ which was found to be enough to discover the `if` expression that switches between swing-up and balancing. As training data we used $N = 5$ state trajectories from the ground truth policy and $M = 2$ trajectories from the latest programmatic imitation policy. All training states were expert labelled with actions from the ground truth policy. As test we simulated 100 rollouts from uniformly random states in the range $\frac{\pi}{2} \leq \theta \leq \frac{3\pi}{2}$, $-1 \leq \dot{\theta} \leq 1$, which is the pendulum below horizontal with relatively low velocity.

Results and discussion: The results of the experiment is shown in fig. 2 (left column). After four iterations of imitation learning a simple policy was found, capable of balancing the pendulum and swinging up from some states. After approximately ten iterations the policy could effectively swing up and balance the pendulum from any state. The imitation learning did not find the ground truth programmatic policy by iteration 10, likely due to the small number of observations in certain areas of the state space. Nonetheless, it managed to synthesize an effective policy which is quite similar to the ground truth.

3.3 Imitation of a neural pendulum swing-up policy

Finally, we examined if we were able to discover a simple, interpretable policy in a more realistic setting, where we synthesized by imitation learning from a neural policy. The task, DSL, and synthesis procedure were as described in the previous experiment, with the ground truth policy as the only difference.

Ground truth: The neural ground truth policy was found by TD3 (Fujimoto et al., 2018), using feed-forward neural networks with 2 hidden layers of 24 neurons for both the actor and critic. Training was run for 5 million steps with a learning rate of 10^{-4} to ensure relatively good convergence.

Results and discussion: The results of the experiment is shown in fig. 2 (right column). After four iterations of imitation learning, a simple imitating policy capable of swinging up and balancing the pendulum is found. After several more iterations, a significantly more complicated programmatic policy was found which resembles the neural policy more closely but yields only a minor performance improvement.

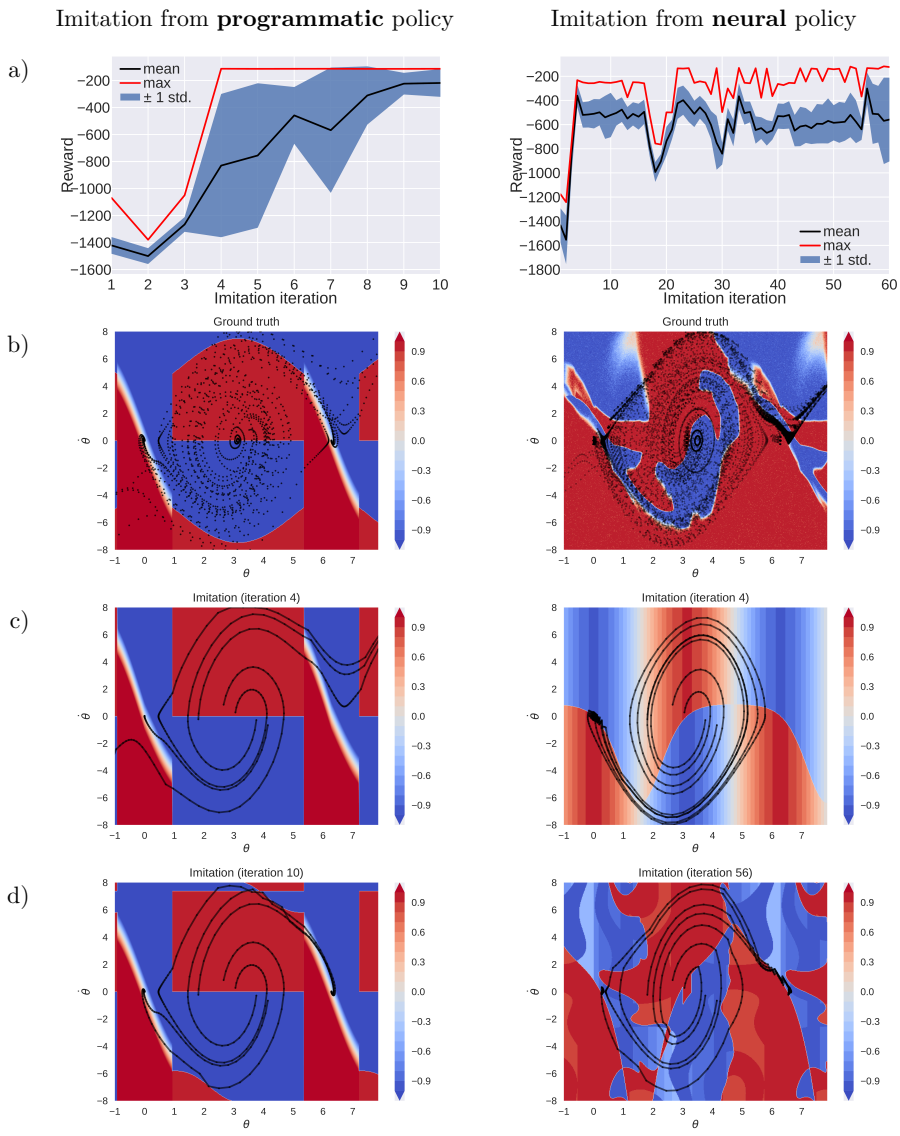


Figure 2: Pendulum swing-up imitation learning of a programmatic policy from a ground truth programmatic (left) or neural (right) policy. Policies are visualized as a heat map; the state space is pendulum angle, θ, and angular velocity, $\dot{\theta}$, and the action is pendulum torque. The goal state is $\theta = 0 \pmod{2\pi}$, $\dot{\theta} = 0$. a) Cumulative reward of test trajectories. b) Ground truth programmatic/neural policy. Points indicate all states seen during training. c) Programmatic policy found after four iterations of imitation learning, with five test trajectories shown. d) Programmatic policy found after several more iterations, with five test trajectories shown.

4 Discussion

We have presented and evaluated our method with simple experiments, and much remains to be done. As mentioned, one goal is to integrate the local search with reinforcement learning as described by alg. 3. While simple, we believe that the presented results show potential, especially through the programs that were discovered in only a few iterations. In particular, it would be interesting to evaluate this approach on more structured tasks, where neural networks might struggle with generalization while a program could be found that immediately generalizes. In such a setting, we could also take better advantage of type-directed search, with more complicated DSLs containing e.g. logic, matrix or computer vision functions potentially still remaining computationally tractable.

It should also be mentioned that local, iterated synthesis as a concept remains orthogonal to several other improvements in program synthesis; for example, enumerating or sampling programs according to a learned probability distribution as in e.g. Ellis et al. (2018) is possible. Instead of depth-limited search, it would be possible to limit the search to programs above a certain likelihood. However, it seems unclear how this distribution would be effectively learned for policies.

4.1 Related work

There has been previous work on synthesizing programmatic policies at the intersection of RL and GP, such as GPRL (Hein et al., 2017). Their method is based on offline GP, performed in a previously learned parametric model of the system of interest. Indeed, they include a comparison with behavioral cloning, which is simply imitating the actions of a trained policy directly. Their method performs better on the actual (simulated, but not learned) system. It is well known that behavioral cloning can lead to poor performance, e.g. Ross et al. (2010), which could lead to the observed performance gap. The authors do not analyze why the model-based method performs better when given the same training data as behavioral cloning, but it seems likely that interaction with the model can overcome some of the distributional issues arising from behavioral cloning.

In RL, it might be preferable to not learn a parametric model if it is used for credit assignment (i.e. policy learning) (van Hasselt et al., 2019). How these results might impact programmatic policy learning is an open question, but it seems possible that similar considerations could be made.

Gupta et al. (2020) proposed a method for using program repair in neural program synthesis. After neural synthesis, the resulting program might not be correct or even satisfy the input-output relation. The authors propose to learn a neural debugger that outputs so-called edits which correct potential errors in the program. The relation to this work is apparent in how we use an edit operator to define the neighborhood of a program.

Kamio et al. (2003) describe a way to integrate GP, RL and simulated systems. By first synthesizing a policy using GP in the simulated system, it can later be adapted and fine-tuned through RL, allowing the policy to function on a real robot.

The presented neighborhood search method can be considered an instance of the (Very) Large-Scale Neighborhood Search framework (Pisinger and Ropke, 2010). The discussion surrounding these methods is very relevant, such as choice of neighborhood structure, determining a good size of the neighborhood, finding more efficient ways to search the neighborhood, and so on. Deterministic versions of genetic algorithms have been considered before, such as in Salomon (2003).

References

Bastani, O., Pu, Y., and Solar-Lezama, A. (2018). Verifiable reinforcement learning via policy extraction.

Ellis, K., Morales, L., Meyer, M. S., Solar-Lezama, A., and Tenenbaum, J. B. (2018). Dreamcoder: Bootstrapping domain-specific languages for neurally-guided bayesian program learning. In *Neural Abstract Machines & Program Induction Workshop at NIPS*.

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods.

Gupta, K., Christensen, P. E., Chen, X., and Song, D. (2020). Synthesize, execute and debug: Learning to repair for neural program synthesis. In *Advances in Neural Information Processing Systems*.

Hein, D., Udluft, S., and Runkler, T. A. (2017). Interpretable policies for reinforcement learning by genetic programming.

Kamio, S., Mitsuhashi, H., and Iba, H. (2003). Integration of genetic programming and reinforcement learning for real robots. In *Genetic and Evolutionary Computation — GECCO 2003*, page 470–482. Springer Berlin Heidelberg.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pisinger, D. and Ropke, S. (2010). *Large Neighborhood Search*, page 399–419. Springer US.

Ross, S., Gordon, G. J., and Andrew Bagnell, J. (2010). A reduction of imitation learning and structured prediction to No-Regret online learning.

Salomon, R. (2003). The deterministic genetic algorithm: implementation details and some results. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. IEEE.

van Hasselt, H. P., Hessel, M., and Aslanides, J. (2019). When to use parametric models in reinforcement learning? In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Verma, A., Le, H., Yue, Y., and Chaudhuri, S. (2019). Imitation-Projected programmatic reinforcement learning. In Wallach, H., Larochelle, H., Beygelzimer, A., dÁlché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 15752–15763. Curran Associates, Inc.

Verma, A., Murali, V., Singh, R., Kohli, P., and Chaudhuri, S. (2018). Programmatically interpretable reinforcement learning.

APPENDIX B

Draft of paper 2

This paper, with the tentative title “Reinforcement learning of causal variables using mediation analysis”, is at the time of writing in preparation for submission to the AAAI 2022 conference.

Reinforcement learning of causal variables using mediation analysis

Anonymous Authors[1]

Abstract

We consider the problem of acquiring causal representations and concepts in a reinforcement learning setting. Our approach defines a causal variable as being both manipulable by a policy, and able to predict the outcome. We thereby obtain a parsimonious causal graph in which interventions occur at the level of policies. The approach avoids defining a generative model of the data, prior preprocessing, or learning the transition kernel of the Markov decision process. Instead, causal variables and policies are determined by maximizing a new optimization target inspired by mediation analysis, which differs from the expected return. The maximization is accomplished using a generalization of Bellman's equation which is shown to converge, and the method finds meaningful causal representations in a simulated environment.

1. Introduction

Hard open problems in reinforcement learning, such as distributional shift, generalization from small samples, disentangled representations and counter-factual reasoning, are intrinsically related to causality (Schölkopf, 2019). Furthermore, causal representations have been emphasized as central to concept acquisition and knowledge representation (Tenenbaum et al., 2011).

Statistical causal analysis, as developed and popularized by Judea Pearl, assumes the data arises as transformations of independent noise sources according to the causal graph (Pearl, 2009). From a practical perspective, the approach of describing data generatively, as arising from nonlinear transformations of i.i.d. noise, underlies the most successful machine learning models today (Shrestha & Mahmood, 2019). That same approach has been successfully applied for fast concept acquisition (Tenenbaum et al., 2011; Lake et al., 2015), as well as control (Deisenroth & Rasmussen, 2011; Levine et al., 2016). However, there are two key differences: First, the causal graph cannot be learned from observational data alone (Pearl, 2009); secondly, and more important for our purpose, these approaches differ from causal analysis in the *scale of modeling*, a term coined by Peters et al. (2017):

Traditional examples of causal modeling, such as the SMOKING → TAR DEPOSITS → CANCER example (Pearl & Mackenzie, 2018), *do* offer a generative process of the few variables included in the analysis, but they *do not* offer a generative process of the underlying temporal phenomena (patient history). The variables are said to be in a *descriptive* relationship to the underlying phenomena, to emphasize that they are not identified as high-level variables in a generative process. The reduction of the data-generating process to a few abstract primitives in a causal relationship, is central to concept acquisition (Tenenbaum et al., 2011), and more broadly to knowledge representation (Davis et al., 1993).

In this article, we aim to answer the following question: Can we automatically learn a parsimonious causal model which is descriptive, rather than generative, of the underlying problem, yet still embodies relevant causal knowledge?

As an illustration, consider the DOORKEY environment, fig. 1. The agent must pick up the key, open the door and go to the goal state in order to receive the reward. Instead of identifying a generative process of the maze, our approach identifies a binary causal variable (for instance, whether the door is opened or not) and builds a small causal graph representing the causal relationship between the identified variable, policy choice and return. The agent is thereby imbued with the causal knowledge that the identified variable is in a causal relationship with the return.

Our approach[1] has two central features: First, that we do not identify a causal variable as a latent variable in a generative model of the data, or as a latent factor which arises from maximizing the expected return $\mathbb{E}_\pi [G_n \mid S_n = s_n]$ with respect to the policy. Instead, we replace the expected return with an alternative maximization target, the *natural indirect effect* (NIE), which is maximized to identify a causal variable. Second, the approach naturally ensures a candidate causal variable represents *a feature of the environment the*

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

[1]Code available at . . .

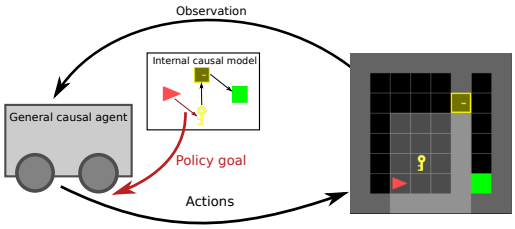


Figure 1. In the DOORKEY environment, the agent (red) must learn to pick up a key to open the door and get to the goal. Our causal agent learns a small, coarse-grained causal network, and uses it when training its policy.

agent can manipulate, thereby ensuring the information is relevant for the agent. This distinguishes between *relevant* causal concepts and irrelevant ones. In the DOORKEY example (fig. 1), a variable corresponding to being one step away from the goal would be a necessary cause for completing the environment. However, it would be no easier to manipulate such a variable than simply reaching the goal state.

To optimize the NIE in a reinforcement learning setting, we apply suitable generalizations of Bellman's equation. This allows us to apply most actor-critic methods, and specifically, use an off-policy method based on the V-trace estimator (Espeholt et al., 2018).

Related work: Determining causal variables has previously been examined in image data from a latent-variable perspective (Besserve et al., 2020; Lopez-Paz et al., 2017) and time-series signals, using (temporal) state aggregation (Zhang et al., 2015). However, these approaches apply a latent-variable criteria which is distinct from ours. The problem of determining causal variables has also been investigated from a fairness-perspective, see (Zhang & Bareinboim, 2018).

In a reinforcement-learning setting, the option-critic architecture considers state-dependent policies similar to ours, but from a non-causal perspective Bacon et al. (2017), and Zhang et al. (2019) learn a state representation using sufficient statistics criteria. Determining latent states to best explain the observations is closely related to the reward machine architecture (Camacho et al., 2019; Icarte et al., 2018), which learns binary feature-vector representations in a logical, rather than causal, relationships. Our approach is different, since we learn both the causal variables and manipulation policies jointly using a causal criteria.

In recent work, reinforcement learning has been applied for causal discovery in graphs with pre-defined variables, using meta-learning (Dasgupta et al., 2019) and active learning (Amirinezhad et al., 2020), for example. Wang et al. (2020) consider confounded observational data in a reinforcement learning setting, and their approach is noteworthy as they suggest a modified Q-learning update. These approaches, however, consider just a handful of variables that can be observed (and manipulated), which is a different setup than the one considered herein.

2. Methods

We consider a general γ-discounted episodic Markov decision problem in which states/actions at time steps $t = 0, 1, \ldots, T$ are denoted S_t/A_t, and the goal is to maximize the expectation of the return $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$. The expectation is with respect to the behavior policy $\pi(a|s) = \Pr(A_t = a|S_t = s)$. For easier interpretation, the examples involve sparse reward $+1$, given at the end of the episode in case of successful termination.

2.1. Mediation analysis

Mediation analysis (Alwin & Hauser, 1975; Pearl, 2012) deals with decomposing the total causal effect, $p(Y = y|\mathrm{do}(X = x))$, a treatment variable X exerts an outcome variable Y into different causal pathways, which may pass through intermediate *mediating variables*.

In the simplest setting (see fig. 2, left), X could be whether a school pupil received extracurricular studies, while Y is their academic performance at the end of the year, and the mediating variable Z could correspond to extra study time.

Mediation analysis allows us to quantify the extent to which a third variable, such as Z, mediates the effect of X on Y. The most important measure is the *natural indirect effect* (Pearl, 2012; 2001), which measures the extent to which X influences Y, solely through Z. For a transition of $X = x$ (starting value) to $X = x'$ (manipulated value), it is defined as expected change in Y, affected by holding X constant at its natural value $X = x$, and changing Z to the value it would have attained *had* X been set to $X = x'$. This quantity involves a nested counter-factual, and cannot be estimated in general; however, for specific causal diagrams, it has a closed-form expression. For instance, in the simple case given in fig. 2 (left), it is defined as (Pearl, 2001):

$$\mathrm{NIE}_{x\to x'}(Y) = \sum_z \mathbb{E}\left[Y|x,z\right]\left[P(z|x') - P(z|x)\right]. \quad (1)$$

The NIE has intuitively appealing properties: It is large when Z is highly influenced by our choice of manipulation $X = x, x'$, meaning that Z is easy to manipulate, and the first term reflects that Z should influence the outcome Y. The product implies a trade-off between these two effects. In our application, we let $X = \Pi$ denote our choice of policy, and then use the NIE to index good (versus bad) choices of the observable variable Z and policies Π. Hence, we hypothesize that by maximizing the NIE (rather than

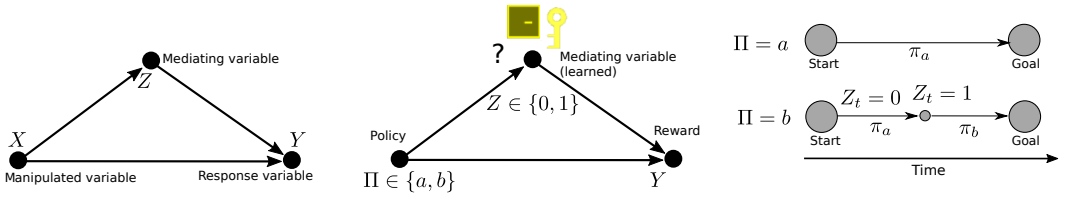


Figure 2. Left: Example of a mediation analysis diagram, in which a cause/effect $X \to Y$ is being mediated by a variable Z. Center: Application to reinforcement learning: The manipulating variable X corresponds to a choice between two policies, whereby the effect Y is the return, and Z is a (learned) variable which influences Y. We wish to quantify how the policy choice influences Y through the variable Z. Right: The value $\Pi = a$ indicates that we follow a baseline policy π_a. This is compared to a policy $\Pi = b$ obtained by following the policy π_b until the first time t where $Z_t = 1$ occurs, in which case we follow π_a. Both π_a, π_b, and the distribution of Z_t, have to be learned.

expected return) we can determine relevant causal variables, which correspond to useful concepts for the agent.

2.2. Causes and effects in reinforcement learning

The most natural causal variable to include in a causal diagram is the expected return $Y = G_0$, since manipulating Y, and therefore learning which variables are relevant for manipulating Y, should remain the eventual goal of the agent.

Since we consider causal variables as aggregates of many individual states, no single action can reasonably be considered a treatment variable. Rather, we consider a treatment equivalent to the choice to follow policy $\Pi = a$ rather than $\Pi = b$.

In the following, we will focus on the simplest possible case, in which the mediating variable Z is binary, with the meaning that $Z = 1$, if the event which Z corresponds to took place during an episode (and otherwise $Z = 0$). This is analogous to how SMOKING is true if the person was smoking *at some point* in the period covered by a study. We therefore define Z as a stopping process

$$Z = \max\{Z_0, Z_1, \dots Z_T \dots\}. \tag{2}$$

where $Z_t \in \{0, 1\}$ for $n = 0, 1, \dots, T$ denotes whether Z became true at time t. Z_t is assumed to only depend on the state, and have distribution $\text{Bern}(\Phi(s_t))$. With these definitions, the causal pathway $\Pi \to Z \to Y$ denotes that the choice of policy Π influences Y by *obtaining* or *making true* Z, whereas $\Pi \to Y$ means the choice of policy alone influences Y, regardless of Z.

Example: Consider the DOORKEY environment from fig. 1. The graphs $\Pi \to Z \to Y$ or $\Pi \to Y$ would reflect either that our choice of Π affects the reward Y through Z, or that the variable Z is irrelevant, and only the choice of policy matters. The outcome depends on the choice of policy and definition of Z.

The combined policy Inspired by the traditional relationship between X and Z in mediation analysis, we assume that if $\Pi = a$ then the agent follows a policy π_a which is trained to simply maximize Y, and that otherwise, if $\Pi = b$, the agent follows a policy π_b which attempts to make Z true (i.e. it is trained with Z as the reward signal). To obtain a well-defined policy for all states, the $\Pi = b$ policy switches back to π_a once $Z = 1$, see fig. 2 (right). In other words, we assume that the agent at time step t follows the policy:

$$\pi = \begin{cases} \pi_a & \text{if } \Pi = a \\ (1 - Z_{0:t})\,\pi_b + Z_{0:t}\pi_a & \text{if } \Pi = b. \end{cases} \tag{3}$$

where $Z_{0:t} = \max\{Z_0, \dots, Z_t\}$. Since Z and Π are binary, the NIE from eq. (1) simplifies to (Pearl, 2001):

$$\begin{aligned} \text{NIE} = &\left(\mathbb{E}\left[Y|Z = 1, \pi_a\right] - \mathbb{E}\left[Y|Z = 0, \pi_a\right]\right) \\ &\times \left(P(Z = 1|\pi_b) - P(Z = 1|\pi_a)\right). \end{aligned} \tag{4}$$

Conditioning on π_a or π_b means that the actions are generated from the given policy.

The NIE has the intuitively appealing property of being separated into a product of two simpler terms, which must both be large for the NIE to be large. The first involves the return, but only conditional on policy π_a. A high value of the NIE implies an increased chance of successful completion of the environment, when $Z = 1$ relative to $Z = 0$.

The second term involves both policies, but uses Z as a reward signal, which is computed during the episode, and will therefore often be known before the episode is completed. Since this is the only term which involves π_b, it induces a modular policy, in which π_b is trained on a simpler problem.

The NIE excludes certain trivial definitions of Z. For instance, if $Z = Y$ in the DOORKEY example, the first term would be maximal. However, in this case, π_a and π_b would be trained on the same target, and so the second term should be zero. On the other hand, if Z is trained to match states

visited by π_b, which are incidental to the reward, it will not result in a high NIE, due to the first term.

Optimizing the NIE involves two challenges unfamiliar from traditional reinforcement learning: (i) The first term involves expectations conditional on Z. (ii) The NIE is optimized both with respect to Z and to π_b.

We overcome these by combining two ideas. First, we express the conditional terms using suitable generalizations of Bellman's equation. Secondly, since we optimize policies based on data collected from other policies, we use V-trace estimates of the relevant quantities (Espeholt et al., 2018).

2.3. Bellman-like equations

The value function satisfies the Bellman equation $v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right]$. On comparison with the terms in eq. (4), we see that the NIE involves conditional expectations.While we could attempt to simply divide the observations according to Z and train two value functions, this method would not provide a way to learn Z itself. To do so, we consider an alternative recursive relationship between the conditional expressions.

For times $t \notin \{0, \dots, T\}$ we define $Z_t = 0$. This allows us to introduce the variables

$$Z_t^\infty = \max\{Z_t, Z_{t+1}, \dots, \} \tag{5}$$

which are true, provided $Z_{t'} = 1$ occurs at a time step following t. Note that $Z = Z_0^\infty$.

Analogous to v_t, we define the value functions:

$$v_t^\infty(s_t) = P(Z_t^\infty = 1 | S_t = s_t, Z_{t-1} = 0), \tag{6}$$
$$v_t^z(s_t) = \mathbb{E}[G_t | S_t = s_t, Z_t^\infty = z, Z_{t-1} = 0]. \tag{7}$$

Note that the expressions are conditional on $Z_{t-1} = 0$. The first denotes that the event Z will happen in the future, starting from s_t, and the second the expected return, given that Z has not happened yet, and either will not $z = 0$, or will $z = 1$ occur in the future. Note that $v_0^z(s_0) = \mathbb{E}\left[G_0 | Z = z, s_0\right]$ and $v_0^\infty(s_0) = P(Z = 1 | s_0)$ corresponds to the terms in eq. (4). The value functions satisfy the recursions (Supplementary Material appendix A):

$$v_t^\infty(s_t) = \Phi(s_t) + \bar{\Phi}(s_t)\mathbb{E}\left[v_{t+1}^\infty(S_{t+1}) | s_t\right], \tag{8a}$$
$$v_t^1(s_t) = \frac{V(s_t)\Phi(s_t)}{V_t^\infty(s_t)} \tag{8b}$$
$$+ \frac{1-\Phi(s_t)}{V_t^\infty(s_t)}\mathbb{E}\left[v_{t+1}^\infty(S_{t+1})\left(R_{t+1} + \gamma v_{t+1}^1(S_{t+1})\right) \mid s_t\right],$$
$$v_t^0(s_t) = \frac{1 - \Phi(s_t)}{1 - v_t^\infty(s_t)} \tag{8c}$$
$$\times\mathbb{E}[\left(1 - v_{t+1}^\infty(S_{t+1})\right)(R_{t+1} + \gamma v_{t+1}^0(S_{t+1})) \mid s_t].$$

The new recursions have the same structure as Bellman's equation, but contain mutually dependent terms. If v_t^∞ and v_t were exactly estimated, the iterative policy evaluation methods corresponding to eqs. (8b) and (8c) would easily be found to be contractions with constant γ, but the updates also converge when v^z, v^∞ and v are all bootstrapped. For proof, see Supplementary Material theorem A.1.

Theorem 2.1 (Convergence, informal)**.** *Assuming $\gamma < 1$ and $0 < \Phi < 1$, all states/actions are visited infinitely often, and $v_\pi, v_\pi^\infty, v_\pi^z$ in eq. (8) are all replaced by randomly initialized bootstrap estimates. Then, (i) the operators eqs. (8b) and (8c) converge at a geometric rate to the true values v_π^z, and (ii) the corresponding online method obtained by replacing the expectations with sample estimates, converges to the true values, provided the learning rates satisfy Robbin-Munro conditions.*

2.4. Off-policy learning using V-trace estimators

The overall approach is to learn neural approximations of v^∞, v and v^z, as defined in eq. (8). This is most easily done by observing that the Bellman-like recursions in eq. (8) all have the form:

$$v_t(s_t) = \mathbb{E}_\mu\left[H_t(s_t, S_{n+1}) + G_t(s_t, S_{t+1})v_{t+1}(S_{t+1}))|s_t\right] \tag{9}$$

where actions are generated using a behavioral policy μ. Expanding the right-hand side n times, allows us to define the n-step return (Espeholt et al., 2018):

$$v_t(s_t) = \mathbb{E}\left[\sum_{i=t}^{t+n-1} H_i \prod_{\ell=t}^{i-1} G_\ell + v_{t+n}(S_{t+n}) \prod_{\ell=t}^{t+n-1} G_\ell \middle| s_k\right], \tag{10}$$

which reduces to eq. (9) if $n = 1$. Supposing the current target policy is π, experience is collected from the behavior policy μ, and then eq. (10) can be used as an estimate of the return, corresponding to π, by using importance sampling. To reduce variance, we use a V-trace type estimator, inspired by Espeholt et al. (2018):

$$V_t(s_t) = v(s_t) + \sum_{i=t}^{t+n-1}\left(\prod_{\ell=t}^{i-1} c_\ell G_\ell\right)\delta_i \tag{11a}$$
$$\delta_i = \rho_i\left[H_i(s_i, s_{i+1}) + G_i v(S_{i+1}) - v(S_i)\right] \tag{11b}$$

where c_ℓ and ρ_k are truncated importance sampling weights:

$$\rho_t = \min\left\{\bar{\rho}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right\}, \; c_t = \min\left\{\bar{c}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right\},$$

and $\bar{\rho} \geq \bar{c}$ are parameters of the method. In the on-policy case, where $\mu = \pi$, the V-trace estimate eq. (11a) reduces to

Algorithm 1 Causal learner

1: Initialize policy networks π_a and π_b (and corresponding critic networks)
2: Initialize networks v, v^0, v^1, v^∞ to estimate v_π, v_π^z, and V_π^∞
3: Initialize causal variable network Φ
4: **repeat**
5: Collect experience from π_a and add to replay buffer
6: Sample experience from replay buffer τ
7: Train π_a (and critic) using AC2
8: Calculate reward signal for π_b from τ using eq. (14) and train π_b (and critic)
9: Train v, v^z, v^∞ using n-step V-trace estimates eq. (11a), computed using eq. (13), using definitions of H_t and G_t implied by eq. (8) and experience τ
10: Train parameters in causal variable network Φ by maximizing eq. (4), where each term has been replaced by the respective V-trace estimate computed using eqs. (8) and (11a)
11: **until** forever

$\sum_{i=t}^{t+n-1} H_i \prod_{\ell=t}^{i-1} G_\ell + v_{t+n} \prod_{\ell=t}^{t+n-1} G_\ell$, and is therefore a direct estimate of eq. (9). In the general case, the method provides a biased estimate, when $\bar{\rho}, \bar{c} < \infty$, but analogous to Espeholt et al. (2018), the stationary value function can be analytically related to the true value function. The result is summarized as: (see Supplementary Material theorem A.2)

Theorem 2.2 (V-trace convergence, informal)**.** *Assume that experience is generated by a behavior policy μ, that $\gamma < 1$, $0 < \Phi < 1$, all states/actions are visited infinitely often, and that $v_\pi, v_\pi^\infty, v_\pi^z$ in eq. (8) are all replaced by randomly initialized bootstrap estimates. Then, if we apply eq. (11a) iteratively [2] on the bootstrap estimate of V^z*

$$V^z(s) \leftarrow_\alpha V^z(s) + \sum_{t=0}^{\infty} \prod_{\ell=0}^{t-1} c_\ell G_\ell^z \delta_\ell, \tag{12}$$

where H_ℓ^z and G_ℓ^z are computed using V-trace estimates of v_π and v_π^∞, it implies that V^z converges to a biased estimate of v_π^z, and if $\bar{\rho}, \bar{c} \to \infty$, then $V^z \to v_\pi^z$.

To practically compute the V-trace estimates, we start from T and proceed to t:

$$V_t = v_t + \delta_t + G_t c_t (V_{t+1} - v_{t+1}). \tag{13}$$

2.5. Combined method

The policy π_b in eq. (3) is trained in an episodic environment to maximize Z. Since the variable Z is multiplicative over individual time steps, we train π_b by decomposing the multiplicative cost using a stick-breaking construction:

$$r_{t+1}^b = \Phi(s_t) \prod_{k=0}^{t-1} (1 - \Phi(s_k)), \tag{14}$$

which satisfies $\sum_{t=0}^{\infty} r_{t+1} = P(Z = 1|\tau)$. Training on this reward signal means that the policy π_b will attempt to maximize the term $P(Z = 1|\pi_b) - P(Z = 1|\pi_a)$ in eq. (4). We can therefore train both π_a and π_b with an actor-critic method, using their respective reward signals, whereby the critics estimate of the return are trained against the V-trace estimate, as computed using eq. (13).

To maximize the NIE with respect to Φ, we introduce networks v, v^z and v^∞, to approximate v_π, v_π^∞ and v_π^z. These are trained using ordinary gradient descent against their V-trace targets, computed by eq. (13). The same V-trace estimates can be used to re-write the NIE in eq. (4), to an expression which directly depends on Φ, and can therefore be trained using stochastic gradient descent. For instance, $\mathbb{E}[Y|Z = 1, \Pi = a, s_0] = v_{\pi_a}^1(s_0)$ is equivalent to $V_t^{z=1}(s_0)$, computed using eq. (13), and the definitions of H_t and G_t implied by eq. (8b), and $\mathbb{E}[Z = 1|\Pi = a]$ can be replaced by V_0^∞, computed using eq. (8a). The pseudocode of the method can be found in algorithm 1. Note that to prevent premature convergence, and speed up convergence when both factors in the NIE are small, we train on a surrogate cost function which includes entropy terms for Φ, π_a and π_b. Full details can be found in Supplementary Material appendix B.

3. Experiments

We first test the value function recursions in eq. (8) on a simple Markov reward process, which we call TWOSTAGE. TWOSTAGE corresponds to an idealized version of the DOORKEY environment, in which the states are divided into two sets S_A and S_B. The initial state is always in S_A, and the environment can either transition within sets ($S_A \to S_A$, $S_B \to S_B$) with a fixed probability, or from set S_A to S_B, with a fixed probability. From S_B, there is a chance to terminate successfully with a reward of $+1$, and from all states there is a chance to terminate unsuccessfully with a reward of 0.

The transition from states in S_A to S_B, creates a bottleneck distinguishing successful and unsuccessful episodes, much like unlocking the door in the DOORKEY environment. The transition probabilities are chosen such that $p(R = 1|s \in S_B) = p(s \in S_B|s \in S_A) = \frac{2}{3}$ and $p(R = 1|s \in S_A) = \frac{4}{9}$, see Supplementary Material appendix B for further details.

[2] $x \leftarrow_\alpha y$ is equivalent to $x = x(1 - \alpha) + \alpha y$

3.1. Tabular learning

As a first example, we will consider simple estimation of the conditional expectations, using the Bellman recursions. We condition on whether the state enters S_B at a later time, i.e. $\mathbb{E}[Y|s_t \in S_B \text{ for some } t > 0, S_0 = s_0]$, which is equivalent to $v^1(s_0)$, since we define $\Phi(s) = 1_{S_B}(s)$. In this case, the Bellman updates from eq. (8)) for a transition $S_t = s$ to $S_{t+1} = s'$, $R_{t+1} = r$ are

$$\begin{aligned} V(s) &\underset{\alpha}{\leftarrow} r + \gamma V(s') \\ V^\infty(s) &\underset{\alpha}{\leftarrow} \Phi(s) + (1 - \Phi(s))V^\infty(s') \\ V^1(s) &\underset{\alpha}{\leftarrow} \frac{V(s)\Phi(s) + (1-\Phi(s))V^\infty(s_t)\left(r + \gamma V^1(s')\right)}{V^\infty(s)} \end{aligned} \tag{15}$$

As anticipated by theorem 2.1, iterating these updates, the value functions converge to their analytically expected values, as can be seen in figs. 3a and 3b, in which we plot v^1 and v^∞. The dashed lines represent the true (analytical) values, and the different colored lines represent the different states. In the case of v^1, the expectation estimated is the probability of successful completion, given that we begin in any state and *at some point* enter S_B; in other words, the information we condition on is something which only occurs *at a later point* in the episode, from the perspective of an observation $s \in S_A$, and therefore the correct estimation of this probability is not simply a matter of computing the return for a state starting in S_B.

3.2. Learning Φ using V-trace estimation

The second example extends the TWOSTAGE example to also learn the causal variable Φ using algorithm 1. Since the environment is a MRP, we discard terms involving π_b, and the objective therefore becomes:

$$\begin{aligned} \Delta_Y &= \mathbb{E}\left[Y|Z=1\right] - \mathbb{E}\left[Y|Z=0\right] \\ &= \mathbb{E}_{s_0}\left[V^1(s_0) - V^0(s_0)\right]. \end{aligned} \tag{16}$$

The expectation is unrolled, using the V-trace approximation, and directly optimized with respect to the parameters of Φ, using the parameterization $\Phi(s) = \frac{1}{1+\exp(-w_s)}$, as described in section 2.5. Further details can be found in Supplementary Material appendix B.

The value function approximation is quickly learned (see fig. 3c), showing convergence to the analytical values. The quantities V^∞ and V^z both depend on Φ, and will therefore only begin to converge after Φ begins to converge (see fig. 3d). Since the conditional expectations V^z depend on V^∞, they will converge relatively slower, but both will eventually converge to their expected value when the learning rate is annealed, see fig. 3e.

3.3. Causal learning and the DOORKEY environment

To apply algorithm 1 to the DOORKEY environment, we first have to parameterize the states. The environment has $|\mathcal{A}| = 5$ actions, and we consider a fully-observed variant of the environment. We choose the simplest possible encoding, in which each tile, depending on its state, is one-hot encoded as an 11-dimensional vector. This means that an $n \times n$ environment is encoded as an $n \times n \times 11$-dimensional sparse tensor, and we include a single one-hot encoded feature to account for the player orientation. Further details can be found in Supplementary Material appendix B. Episode length is capped at 60 steps.

Since the environment encodes orientation, player position and goal position separately, and since specific actions must be used when picking up the key and opening the door, the environment is surprisingly difficult to explore and generalize in. We train an agent using A2C (Mnih et al., 2016) with 1-hidden-layer fully connected neural networks, which results in a completion rate of about 0.25 within the episode limit. We also attempted to train an agent using the Option-Critic framework (Bacon et al., 2017), since it is a quite relevant comparison to our method, but failed to find parameters for which the learned options could solve the environment better than chance.

After an initial training of π_a, we train Φ and π_b by maximizing the NIE, using algorithm 1. We use a batch size of 100, buffer size of 1000, gradient normalization and a learning rate of 0.01, see Supplementary Material appendix B for additional details.

To obtain a fair evaluation on separate test data, we simulate the method on 200 random instances of the DOORKEY environment, and use Monte-Carlo roll-outs of the policies π_a and π_b to estimate the quantities $\mathbb{E}\left[Z = 1 \mid \Pi = a\right]$, $\mathbb{E}\left[Z = 1 \mid \Pi = b\right]$. This then allows us to estimate the NIE on a separate test set.

To examine whether the obtained definition of Z is non-trivial, we compare it against a natural alternative that learns Z by maximizing the cross-entropy of Z and Y,

$$-\mathbb{E}_\tau\left[Y(\tau) \log P(Z = z|\tau)\right]. \tag{17}$$

Since Y is binary, this corresponds to determining Φ as the binary classifier which separates successful ($Y = 1$) episodes from unsuccessful episodes ($Y = 0$), i.e. ensures that the first factor of the NIE eq. (16) is large.

The results of both methods can be found in table 1 (results averaged over 10 runs). The causal learner obtains a value of the NIE that is significantly different from zero in all runs. While the absolute value is small, this can be attributed to the NIE being a product of two factors which are both small.

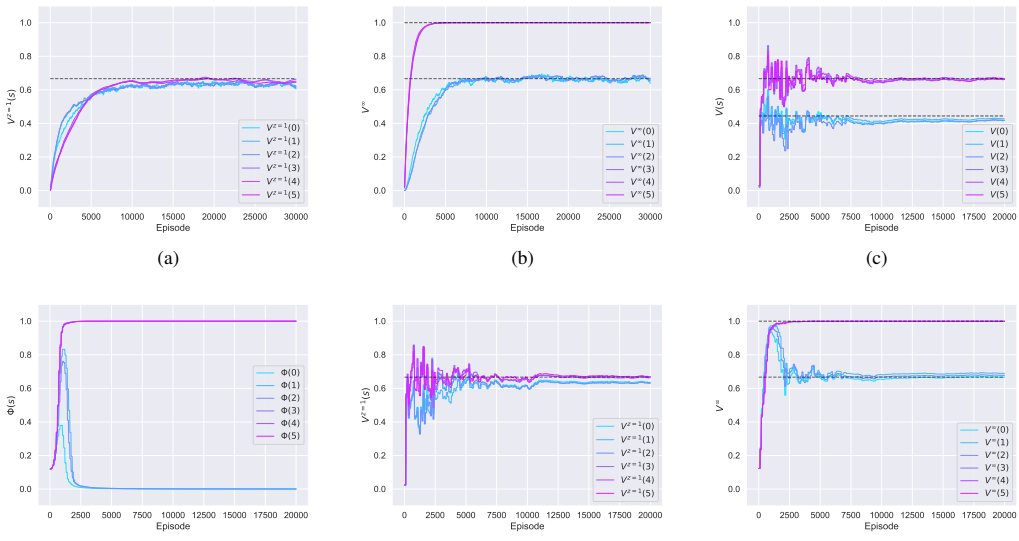


Figure 3. (a-b) Trace plots of v^1 and v^∞ for the tabular TWOSTAGE environment obtained using eq. (10), with a given Φ. (c-f) Estimates with neural function approximators for the value functions in the TWOSTAGE environment, while Φ is being learned.

Table 1. Performance of causal agent on the DOORKEY environment and standard deviation of the mean.

Method	$\mathbb{E}[Y \mid Z=1]$	$\mathbb{E}[Y \mid Z=0]$	Δ_Y	$\mathbb{E}_{\pi_a}[Z]$	$\mathbb{E}_{\pi_b}[Z]$	NIE
Causal Learner	0.410(30)	0.000()	0.410(30)	0.550(20)	0.790(20)	0.098(10)
Cross-entropy	0.560(80)	0.130(30)	0.430(100)	0.270(40)	0.270(30)	0.011(8)

Considering the first two terms, we observe that the causal variable $Z = 1$ is a necessary condition for completing the environment, while the corresponding variable for the cross-entropy target can be false, yet the agent is still able to successfully complete the environment.

We also notice that the cross-entropy based learner outperforms the causal target, in terms of obtaining a proper separation between good versus bad trajectories, i.e. a higher value of Δ_Y. This is expected, since cross-entropy is an efficient cost-function for a binary classification problem.

However, the causal variable Z, which is learned by the cross-entropy learner, does not present a suitable target for the policy π_b. Indeed, the variable Z becomes true at the same rate under π_a and π_b (all policies are trained using the same settings). This can be accounted for by recalling that the environment is random, and that the variable Z learned by the causal learner represents a relatively stable feature of the environment (such as picking up the key, opening the door, etc.), whereas the cross-entropy trained variable Z corresponds to a combination of features in the environment which presents a less suitable optimization target.

To obtain insight in the causal variable we learn, we plot $P(Z = 1)$ both against the reward, and whether the door was opened in this particular run (jitter added for easier visualization). The results can be found in fig. 4. As indicated, the learned causal variable correlates well with whether the door is opened or not, and not as well with the total reward. In other words, the method is able to learn that the feature of whether the agent has opened the door acts as a mediating cause in terms of completing the environment. This is a natural result, considering this task is necessary in order for the agent to complete the environment.

The fact that the causal variable corresponds to a meaningful objective, is reinforced by examining the total reward obtained from either following policy $\Pi = a$, or the joint policy $\Pi = b$ (see table 2). Although the difference is slight, we observe a small increase in accumulated reward for the joint policy.

Table 2. Total reward obtained in the DOORKEY environment.

Method	$\mathbb{E}[Y\|\Pi = a]$	$\mathbb{E}[Y\|\Pi = b]$
Causal Learner	0.240(20)	0.300(30)
Cross-entropy	0.230(20)	0.240(10)

4. Conclusion

Every causal conclusion depends on assumptions that are not testable in observational studies (Pearl et al., 2009). In all methods of causal discovery, therefore, the question arises of why we are justified in believing that a particular method finds a *causal* representation of the environment.

In work involving pre-defined variables, such justification can be found either through external distributional assumptions about the relationship between the structure of the model and the data it generates (Spirtes et al., 2000), or in the model belonging to a class of models of which so many examples have been observed, that meta-learning allows the structure to be identified (Dasgupta et al., 2019).

In our work, we assume a specific causal diagram, and this, along with the definition of Z and Y, ensures that Z is automatically imbued with a natural interpretation as a mediating factor in the reason Y became true: Our approach identifies causal variables exactly by searching for definitions of variables which meet the definition of mediating cause.

A far more difficult and fundamental question is *why* a parsimonious model of causal knowledge, such as the SMOKING/CANCER example, is preferable to a detailed causal model of patient history. In our view, human interpretability is not a sufficient case for parsimonous causal models. Additionally, if we adopt the view that a model should best fit the MDP (i.e. a generative view), it is difficult to see why parsimony would be preferred.

Although we cannot claim a definite answer, our approach differentiates situations in which it can obtain causal knowledge, from those in which it cannot, *without* referencing a generative/best fit criteria. For a variable Z to be identified, it must always occur relative to a policy π_a, as *something the agent could potentially do*, and is associated with a high reward ($\mathbb{E}[Y|Z = 1] > \mathbb{E}[Y|Z = 0]$), but it *might not do it under its baseline behavior* π_a. In light of that, the policy must be sub-optimal in order for the agent to determine a causal model. Though at a first glance this may seem like a flaw in our method, the idea that causation is ill defined when one has a precise description of the physical world is not new (Russell, 1913), and it closely matches every day reasoning about causal mechanism. When a child learns to ride a bicycle, for example, he tries to do so as best as possible (but not optimally). A potential causal explanation

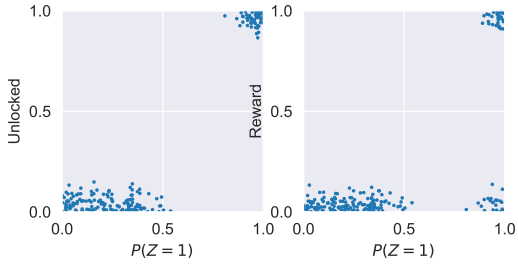


Figure 4. Scatter plot of $P(Z = 1)$ and (left) chance of unlocking the door, and (right) chance of successfully reaching the goal state. The causal variable Z appears to correspond to opening the door.

for a subsequent fall, such as *steering too far from the center of the lane*, $Z = 0$, and for hitting the curb $Y = 0$, is only relevant in case the child can actively take actions to keep near the center of the lane $Z = 1$. To put this in a different way, if the agent knows so much about the environment it has an optimal policy, a coarse-grained causal model cannot offer the agent any benefits because there are no policy decisions to optimize.

We have argued that the NIE offers an interesting alternative way to define coarse-grained causal knowledge. In order to identify a causal variable, our method must have also learn a policy to manipulate it, and the variables are learned directly from experience without first requiring specification of a generative process. Although the method depends on estimating conditional expectations, we have shown these can be estimated using an n-step temporal different methods, and we have shown the method has provable convergence guarantees comparable to TD learning (but with worse constants). We found that the method was able to learn a causal variable which was both sensible and relevant for solving a simple task, and did so better than a natural (non-causal) alternative method. An independent experiment on a test-set indicated the causal representation is associated with an increased NIE, and that the causal representation is relevant for the task in that it resulted in a performance increase. An alternative causal variable trained on a discriminative criteria did not offer similar benefits.

Future work should focus on identifying a larger network of causes, for instance by letting Z play the role of Y and repeating the analysis, or to use ideas from mediation analysis with multiple mediators (VanderWeele & Vansteelandt, 2014). It would likewise be of interest to extend the convergence analysis to the case where Φ is learned.

References

Alwin, D. F. and Hauser, R. M. The decomposition of effects in path analysis. *American sociological review*,

pp. 37–47, 1975.

Amirinezhad, A., Salehkaleybar, S., and Hashemi, M. Active learning of causal structures with deep reinforcement learning. *arXiv preprint arXiv:2009.03009*, 2020.

Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Besserve, M., Mehrjou, A., Sun, R., and Schölkopf, B. Counterfactuals uncover the modular structure of deep generative models. In *Eighth International Conference on Learning Representations (ICLR 2020)*, 2020.

Camacho, A., Icarte, R. T., Klassen, T. Q., Valenzano, R. A., and McIlraith, S. A. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, volume 19, pp. 6065–6073, 2019.

Dasgupta, I., Wang, J., Chiappa, S., Mitrovic, J., Ortega, P., Raposo, D., Hughes, E., Battaglia, P., Botvinick, M., and Kurth-Nelson, Z. Causal reasoning from meta-reinforcement learning. *arXiv preprint arXiv:1901.08162*, 2019.

Davis, R., Shrobe, H., and Szolovits, P. What is a knowledge representation? *AI magazine*, 14(1):17–17, 1993.

Deisenroth, M. and Rasmussen, C. E. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472. Citeseer, 2011.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pp. 1407–1416. PMLR, 2018.

Icarte, R. T., Klassen, T., Valenzano, R., and McIlraith, S. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pp. 2107–2116. PMLR, 2018.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.*, 17(1):1334–1373, January 2016. ISSN 1532-4435.

Lopez-Paz, D., Nishihara, R., Chintala, S., Scholkopf, B., and Bottou, L. Discovering causal signals in images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 6979–6987, 2017.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Pearl, J. Direct and indirect effects. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pp. 411–420, 2001.

Pearl, J. *Causality: Models, Reasoning and Inference*. Cambridge University Press, USA, 2nd edition, 2009. ISBN 052189560X.

Pearl, J. *The mediation formula: A guide to the assessment of causal pathways in nonlinear models*. Wiley Online Library, 2012.

Pearl, J. and Mackenzie, D. *The book of why: the new science of cause and effect*. Basic Books, 2018.

Pearl, J. et al. Causal inference in statistics: An overview. *Statistics surveys*, 3:96–146, 2009.

Peters, J., Janzing, D., and Schölkopf, B. *Elements of Causal Inference - Foundations and Learning Algorithms*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, MA, USA, 2017.

Russell, B. On the notion of cause', reprinted in mysticism and logic and other essays [1917], 1913.

Schölkopf, B. Causality for machine learning. *arXiv preprint arXiv:1911.10500*, 2019.

Shrestha, A. and Mahmood, A. Review of deep learning algorithms and architectures. *IEEE Access*, 7:53040–53065, 2019.

Spirtes, P., Glymour, C. N., Scheines, R., and Heckerman, D. *Causation, prediction, and search*. MIT press, 2000.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., and Goodman, N. D. How to grow a mind: Statistics, structure, and abstraction. *Science*, 331(6022):1279–1285, 2011. ISSN 0036-8075. doi: 10.1126/science.1192788.

VanderWeele, T. and Vansteelandt, S. Mediation analysis with multiple mediators. *Epidemiologic methods*, 2(1): 95–115, 2014.

Wang, L., Yang, Z., and Wang, Z. Provably efficient causal reinforcement learning with confounded observational data. *ArXiv*, abs/2006.12311, 2020.

Zhang, A., Lipton, Z. C., Pineda, L., Azizzadenesheli, K., Anandkumar, A., Itti, L., Pineau, J., and Furlanello, T. Learning causal state representations of partially observable environments. *arXiv preprint arXiv:1906.10437*, 2019.

Zhang, J. and Bareinboim, E. Fairness in decision-making—the causal explanation formula. In *Proceedings of the... AAAI Conference on Artificial Intelligence*, 2018.

Zhang, K., Gong, M., and Schölkopf, B. Multi-source domain adaptation: A causal view. In *AAAI*, volume 1, pp. 3150–3157, 2015.

Supplementary Material

A. Properties of the method

To simplify the analysis, we will assume there is a ϕ such that $0 < \phi \leq \Phi(s) < 1 - \phi < 1$ and the reward is bounded $|R_t| \leq M$. The Bellman operators corresponding to eqs. (8a) to (8c) are:

$$\mathcal{T}^{\infty}V^{\infty}(s) = \Phi(s) + (1 - \Phi(s))\mathbb{E}\left[V^{\infty}(S')|s\right] \tag{1a}$$

$$\mathcal{T}^{1}V^{1}(s) = \frac{V(s)\Phi(s)}{V^{\infty}(s)} + \frac{1 - \Phi(s)}{V^{\infty}(s)}\mathbb{E}\left[V^{\infty}(S')\left(R + \gamma V^{1}(S')\right) \mid s\right] \tag{1b}$$

$$\mathcal{T}^{0}V(s) = \frac{1 - \Phi(s)}{1 - V^{\infty}(s)}\mathbb{E}[\,(1 - V^{\infty}(S'))(R + \gamma V^{0}(S')) \mid s] \tag{1c}$$

where s is the current state and S' is the state we immediately transition to under policy π. Note $\mathcal{T}^{\infty}$ is a contraction with constant ϕ and we let $\alpha = \max\{\gamma, 1 - \phi\}$ for simplicity. To avoid repetitive algebra, we will first focus on the V-trace operator from eq. (11)

$$\mathcal{T}V(s) = V(s) + \mathbb{E}_{\mu}\left[\sum_{t=0}\left(\prod_{\ell=0}^{t-1} c_{\ell}G_{\ell}\right)\rho_t\left(H_t + G_tV(s_{t+1}) - V(s_t)\right)\middle| S_0 = s\right] \tag{2}$$

Where $\rho_t = \min\left(\bar{\rho}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right)$ and $c_t = \min\left(\bar{c}, \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}\right)$ are the truncated IS weights.

Note both of the operators $\mathcal{T}^0, \mathcal{T}^1$ are instances of the $\mathcal{T}$ for appropriate choices of $H_t = H(R_{t+1}, S_t, S_{t+1})$ and $G_t = G(S_t, S_{t+1})$ and assuming the specific case where $\rho_0 = 1, 0 = \rho_1, \rho_2, \ldots$, and $c_t = 1$ for all t.

We first show $\mathcal{T}$ is a contraction when H and G are fixed:

Lemma A.1. *Assuming for all s that $0 \leq \mathbb{E}[G_t \mid s] \leq \kappa < 1$, the policy has full support in the first step $\pi_0(a_0|s_0) > 0$, and $\bar{c} \leq \bar{\rho}$ then $\mathcal{T}$ defined in eq. (2) is a contraction and the fixed point $V_{\bar{\phi}}$ is the value function*

$$V_{\bar{\pi}}(s) = \mathbb{E}_{\bar{\pi}}\left[\sum_{t=0}^{\infty} H_t\prod_{\ell=0}^{t-1} G_t\,\middle|\, S_0 = s\right] \tag{3}$$

of the policy

$$\bar{\pi}(a|s) = \frac{\min\{\bar{\rho}\mu(a \mid s), \pi(a \mid s)\}}{\sum_{a' \in \mathcal{A}}\min\{\bar{\rho}\mu(a' \mid s), \pi(a' \mid s)\}} \tag{4}$$

Proof. Using the shorthand $c_{k:m} = \prod_{i=k}^{m} c_i$ we can re-write eq. (2) as

$$\mathcal{T}V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-2}G_{0:t-1}\left[(\rho_{t-1} - c_{t-1}\rho_t)\,V(s_t) + c_{t-1}\rho_t H_t\right]\middle| s\right] \tag{5}$$

where we use the convention $c_{0:-2} = c_{0:-1} = G_{0:-1} = 1$. Applying the operator on two value functions V_1 and V_2 gives us

$$\mathcal{T}V_1(s) - \mathcal{T}V_2(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-2}G_{0:t-1}(\rho_{t-1} - c_{t-1}\rho_t)(V_1(s_t) - V_2(s_t))\,\middle|\, s\right] \tag{6}$$

$$= \sum_{t=0}^{\infty}\mathbb{E}\left[c_{0:t-2}G_{0:t-1}(\rho_{t-1} - c_{t-1}\mathbb{E}_{a_t|s_t}\rho_t)(V_1(s_t) - V_2(s_t))\mid s\right] \tag{7}$$

In the last expression, we could peel off the expectation wrt. the last action and move it inside the parenthesis. Note that $\rho_{t-1} - c_{t-1}\mathbb{E}_{a_t}\rho_t \geq \rho_{t-1} - c_{t-1}\mathbb{E}_{a_t}\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)} = \rho_{t-1} - c_{t-1} > 0$. The expression is therefore a sum of weighted terms $V_1 - V_2$ where all the weights are non-negative, and therefore we obtain the bound:

$$|\mathcal{T}V_1(s) - \mathcal{T}V_2(s)| \leq \sum_{t=0}^{\infty} \mathbb{E}\left[c_{0:t-2}G_{0:t-1}(\rho_{t-1} - c_{t-1}\rho_t) \mid s\right] \|V_1 - V_2\|_\infty \tag{8}$$

The constant can in turn be easily bounded by noting that the $0 \leq \mathbb{E}[G_t|s] \leq \kappa \leq 1$ and only subtracting the first term of the sum:

$$\begin{aligned}\sum_{t=0}^{\infty} \mathbb{E}\left[c_{0:t-2}G_{0:t-1}(\rho_{t-1} - c_{t-1}\rho_t) \mid s\right] &= 1 - \sum_{t=0}^{\infty} \mathbb{E}\left[\rho_t c_{0:t-1}G_{0:t-1}\left(1 - G_t\right) \mid s\right] \\ &\leq 1 - \mathbb{E}[\rho_0|s](1-\kappa) < 1\end{aligned} \tag{9}$$

□

Remark: Both eq. (1b) and eq. (1c) are instances of eq. (2):

$$G_t^1 = \gamma\frac{(1-\Phi(s_t))V^\infty(s_{t+1})}{V^\infty(s_t)}, \quad G_t^0 = \gamma\frac{(1-\Phi(s_t))(1-V^\infty(s_{t+1})}{1-V^\infty(s_t)} \tag{10a}$$

$$H_t^1 = \frac{V(s_t)\Phi(s_t) + (1-\Phi(s_t))V^\infty(s_{t+1})R_{t+1}}{V^\infty(s_t)}, \quad H_t^1 = \frac{(1-\Phi(s_t))(1-V^\infty(s_{t+1})R_{t+1}}{1-V^\infty(s_t)} \tag{10b}$$

And they are easily seen to satisfy the conditions of lemma A.1:

$$\mathbb{E}[G_t^1] = \gamma\frac{(1-\Phi(s_t))\mathbb{E}V^\infty(s_{t+1})}{\Phi(s_t) + (1-\Phi(s_t))\mathbb{E}V^\infty(s_{t+1})} < \gamma(1-\phi) \tag{11}$$

$$\mathbb{E}[G_t^0] = \gamma\frac{(1-\Phi(s_t))(1-\mathbb{E}[V^\infty(s_{t+1})]}{1-\Phi(s_t) - (1-\Phi(s_t))\mathbb{E}V^\infty(s_{t+1})} \leq \gamma(1-\phi) \tag{12}$$

A.1. Convergence of V-trace

The next lemma is cumbersome to state, but simply says that if we plug the V-trace estimates of V^∞ and V into the expression for V^z, eq. (10), and then use the V-trace to learn all terms online, then the method still converges.

Lemma A.2 (Convergence of joint updates under V-trace)**.** *Suppose we initialize V, V^∞ and V^z arbitrarily. Let $(H_t^z, G_t^z)_t$ be defined as in eq. (10) and correspond to the Bellman operator eqs. (1b) and (1c) for V^z and assume the conditions of lemma A.1 are met.*

Suppose we apply the V-trace operator eq. (2), adapted to the particular expressions for G_t and H_t implied in eqs. (1a) to (1c), a well as the regular V-trace updates for V. Then, provided the conditions of lemma A.1 are met, V^z will converge to the unique fixed point given in lemma A.1 for the problem $\bar{H}_t^z$ and $\bar{G}_t^z$ corresponding to H_t^z, G_t^z but with the value functions of V^∞ and V replaced by their V-trace fixed points given in lemma A.1 at a rate:

$$\|\mathcal{T}^{(T)}V_1^z - \bar{V}^z\|_\infty < \xi^T\|V_1^z - \bar{V}^z\|_\infty + \frac{C}{1-\xi}(1-\phi)^T \tag{13}$$

for constants $0 \leq \xi < 1$ and $0 < C$.

Proof. Let $\bar{V}$ and $\bar{V}^\infty$ and $\bar{V}^z$ be the fixed points arising from using lemma A.1 on eqs. (1a) to (1c) using the definitions of $(H_t, G_t)_t$ in eq. (10).

Suppose $\hat{H}_t$ and $\hat{G}_t$ be the current estimates of H_t and G_t arising by plugging in the current expression of V^∞ and V into eq. (10).

Since V^∞ is the result of evaluating $\bar{\pi}$ it must satisfy $\phi \leq \bar{V}^\infty \leq 1-\phi$. Similarly we can assume V^z is initialized to satisfy $\|V^z\|_\infty < R_M$. Then eq. (5) allows us to bound the difference between the current value function V_1^z and the fixed point $\bar{V}^z$ as:

$$
\begin{aligned}
&|\mathcal{T}V_1^z(s) - \mathcal{T}\bar{V}^z(s)| && (14)\\
&= \left|\mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-2}\left[(\rho_{t-1}-c_{t-1}\rho_t)\left(V_1^z(s_t)\hat{G}_{0:t-1} - \bar{V}^z(s_t)\bar{G}_{0:t-1}\right) + c_{t-1}\rho_t(\hat{G}_{0:t-1}\hat{H}_t - \bar{G}_{0:t-1}\bar{H}_t)\right]\middle| s\right]\right| \\
&\leq \mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left[\bar{G}_{0:t-1}|V^z(s_t) - \bar{V}^z(s_t)|\right]\middle| s\right] && (15)\\
&+ \left|\mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left[\hat{G}_{0:t-1} - \bar{G}_{0:t-1}\right]V_1^z(s_t)\middle| s\right]\right| && (16)\\
&+ \left|\mathbb{E}\left[\sum_{t=0}^{\infty} c_{0:t-1}\rho_t\left[\hat{G}_{0:t-1}\hat{H}_t - \bar{G}_{0:t-1}\bar{H}_t\right]|\middle| s\right]\right| && (17)
\end{aligned}
$$

The first term is recognized as an instance of eq. (7) and the lower bound in eq. (9) applies. For the second term, when $\delta_t = \bar{V}_t^\infty(s_t) - \hat{V}_t^\infty(s_t)$, and focusing on V^1, then

$$
\bar{G}_t - \hat{G}_t = (1-\Phi_t)\left[\frac{\delta_{t+1}}{\bar{V}_t^\infty + \delta_t} - \frac{\delta_t \bar{V}_{t+1}^\infty}{\bar{V}_t^\infty(\bar{V}_t^\infty + \delta_t)}\right] = \Delta_t. \tag{18}
$$

The next step is to expand the difference in the product and obtain a linear combination of non-negative terms. We expand the difference in the product $\hat{G}_{0:t-1} - \bar{G}_{0:t-1}$ in orders of Δ_ℓ:

$$
\begin{aligned}
&\left|\mathbb{E}\left[c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left[\hat{G}_{0:t-1} - \bar{G}_{0:t-1}\right]V_1^z(s_t)\middle| s\right]\right| && (19)\\
&\leq \sum_{\ell=0}^{t} |\mathbb{E}\left[c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left(\prod_{i\neq\ell} G_i\right)(\Delta_\ell V_1^z(s_t))\right]| + \sum_{\substack{\ell,s=0\\ \ell\neq s}}^{t} |\mathbb{E}\left[c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left(\prod_{i\neq\ell,s} G_i\right)(\Delta_\ell\Delta_s V_1^z(s_t))\right]| + \cdots && (20)
\end{aligned}
$$

By an argument similar to lemma A.1, eq. (9) each term in the sum is a linear combinations with positive weights of the terms involving products of Δ and V^z. We can therefore upper-bound it as

$$
\leq \sum_{\ell=0}^{t} \mathbb{E}\left[c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left(\prod_{i\neq\ell} G_i\right)\right]\|\Delta_\ell V_1^z\|_\infty + \sum_{\substack{\ell,s=0\\ \ell\neq s}}^{t} \mathbb{E}\left[c_{0:t-2}\left(\rho_{t-1}-c_{t-1}\rho_t\right)\left(\prod_{i\neq\ell,s} G_i\right)\right]\|\Delta_\ell\Delta_s V_1^z\|_\infty + \ldots, \tag{21}
$$

Since the expectation of G_i is bounded by $\gamma(1-\phi)$, all terms are positive. Each term $\|\Delta_\ell\|$ can be bounded by Δ, and we obtain the bound

$$
\leq \binom{t+1}{1}(\gamma(1-\phi))^t\Delta R_M + \binom{t+1}{2}(\gamma(1-\phi))^{t-1}\Delta^2 R_M + \ldots \quad = R_M\left(\gamma(1-\phi)+\Delta\right)^t - R_M(\gamma(1-\phi))^t. \tag{22}
$$

Since the Bellman update for V^∞ is a contraction with constant at most $1-\phi$, and using that the bounds on V^∞ imposes restrictions on the possible values of δ_t and δ_{t+1}, we can in the case of V^1 upper-bound Δ_t in eq. (19) by $|\Delta_t| \leq \frac{|\delta_{t+1}|}{\phi} \leq \frac{(1-\phi)^T}{\phi}$ where T are the number of Bellman updates of V^∞.

Therefore, eq. (19) can be bounded by

$$
\begin{aligned}
\sum_{t=0}^{\infty} R_M\left(\gamma(1-\phi)+\Delta\right)^t &< R_M\left[\frac{1}{1-\gamma(1-\phi)-\Delta} - \frac{1}{1-\gamma(1-\phi)}\right] && (23)\\
&= \frac{R_M}{\phi}\frac{(1-\phi)^T}{(1-\gamma(1-\phi))((1-\gamma(1-\phi)) - \phi^{-1}(1-\phi)^T} && (24)
\end{aligned}
$$

The last term eq. (17) can be bounded using similar techniques. All in all we obtain:

$$\|\mathcal{T}V_1^z - \bar{V}^z\|_\infty \leq C(1-\phi)^T + \underbrace{(1 - \min_s \mathbb{E}[\rho_0|s](1-\gamma(1-\phi)))}_{=\xi<1} \|V_1^z - \bar{V}^z\|_\infty. \tag{25}$$

Finally, assuming π has a minimal support wherever the behavioral policy μ has support, it holds that $\xi < 1$. Applying the operator T times we obtain:

$$\|\mathcal{T}^{(T)}V_1^z - \bar{V}^z\|_\infty < \xi^T \|V_1^z - \bar{V}^z\|_\infty + \frac{C}{1-\xi}(1-\phi)^T. \tag{26}$$

□

Theorem A.1 (Convergence of online updates of V^z)**.** *Assume a finite state and action space and suppose a list of trajectories $s_0, a_0, r_1, \ldots$ are generated by following a policy π. Assume each state is visited infinitely often and $(\alpha_k)_k$ is a sequence of learning rates satisfying the Robbins-Munro conditions and $\phi < \Phi < 1 - \phi$ is bounded. If we iteratively apply the updates corresponding to eq. (8), for all states/rewards in the trajectory k:*

$$V_{k+1}(s_t) \leftarrow_{\alpha_k} r_{t+1} + \gamma V_k(s_{t+1}) \tag{27a}$$
$$V^\infty_{k+1}(s_t) \leftarrow_{\alpha_k} \Phi(s_t) + \bar{\Phi}(s_t) V^\infty_k(s_{t+1}) \tag{27b}$$
$$V^1_{k+1}(s_t) \leftarrow_{\alpha_k} \frac{V_k(s_t)\Phi(s_t)}{V^\infty_k(s_t)} + \frac{1-\Phi(s_t)}{V^\infty_k(s_t)} V^\infty_k(s_{t+1}) \left(r_{t+1} + \gamma V^1_k(s_{t+1})\right) \tag{27c}$$
$$V^0_{k+1}(s_t) \leftarrow_{\alpha_k} \frac{1-\Phi(s_t)}{1-V^\infty_k(s_t)}(1 - V^\infty_k(s_{t+1}))(r_{t+1} + \gamma V^0_k(s_{t+1})) \tag{27d}$$

using learning rate α_k then $V^z \underset{\infty}{\to} v_\pi$ a.s.

Proof. As already remarked, the updates are an online approximation of the operators eq. (1), where we only include the $t = 0$ term of the V-trace operator eq. (2). Assuming H_t and G_t were available for the converged value functions $\bar{V}_\pi$ and $\bar{v}^\infty_\pi$ this operator is by lemma A.1 a contraction with constant (see eq. (9))

$$|\mathcal{T}V_1^z(s) - \mathcal{T}V_2^z(s)| \leq \gamma(1-\phi)\|V_1^z - V_2^z\|_\infty \tag{28}$$

and therefore converge to fixed points $\bar{V}^0$ and $\bar{V}^1$. When the quantities V^∞ and V are estimated online, and we only have approximations $\hat{V}^\infty$ and $\hat{V}$ available, we re-use the argument in lemma A.2 once more truncating the sum at $t = 0$. We obtain:

$$|\mathcal{T}V^z(s) - \bar{V}^z(s)| \leq \mathbb{E}\bar{G}_0|V^z(s') - \bar{V}^z(s')| + \mathbb{E}|\bar{G}_0 - \hat{G}_0)||V^z(s')| + \mathbb{E}|\bar{H}_0 - \hat{H}_0)||V^z(s')| \tag{29}$$

The terms can be bounded to obtain a similar result but with simpler constants:

$$|\mathcal{T}V^z(s) - \bar{V}^z(s)| \leq \gamma(1-\phi)|V^z(s') - \bar{V}^z(s')| + 2\Delta_0 R_M \tag{30}$$
$$\leq \gamma(1-\phi)|V^z(s') - \bar{V}^z(s')| + \frac{2R_M}{\phi}(1-\phi)^T \tag{31}$$

where T is the number of times the contraction operator has been applied to V^∞ and we therefore once more obtain:

$$|\mathcal{T}V^z(s) - \bar{V}^z(s)| \leq \gamma(1-\phi)|V^z(s') - \bar{V}^z(s')| + 2\Delta_0 R_M \tag{32}$$
$$\leq \gamma(1-\phi)|V^z(s') - \bar{V}^z(s')| + \frac{2R_M}{\phi}(1-\phi)^T \tag{33}$$

and therefore

$$\|\mathcal{T}^{(T)}V_1^z - \bar{V}^z\|_\infty < (\gamma(1-\phi))^T \|V_1^z - \bar{V}^z\|_\infty + \frac{2R_M}{\phi(1-\gamma(1-\phi))}(1-\phi)^T. \tag{34}$$

The updates in eq. (27) are simply an online stochastic relaxation of these expressions and therefore converge (Kushner & Yin, 2003) □

Theorem A.2 (Convergence V-trace updates of V^z)**.** *Under the same assumptions as in theorem A.1, and assuming the weights satisfy* $c_i = \min\left\{\bar{c}, \frac{\pi(a_i|s_i)}{\mu(a_i|s_i)}\right\}, \rho_i = \min\left\{\bar{\rho}, \frac{\pi(a_i|s_i)}{\mu(a_i|s_i)}\right\}$ *with* $\bar{\rho} \geq \bar{c}$ *and* π *has a minimal support where the behavior policy* μ *has support so that* $\mathbb{E}[\rho_t]$ *can be lower bounded. Suppoose we iteratively apply the* V*-trace updates*

$$V_k(s) \leftarrow_{\alpha_k} \sum_{t=0}^{t-1} \left(\prod_{\ell=0}^{t-1} c_\ell G_\ell\right) \rho_t \left(H_{t,k} + G_{t,k} V_k(s_{t+1}) - V_k(s_t)\right) \tag{35}$$

with $H_{t,k}, G_{t,k}$ *being those choices of* H_t, G_t *appropriate for estimating* V*,* V^∞ *and* V^z *(but with values at iteration* k *inserted). Then, provided the learning rates* α_k *satisfying Robbin-Munro conditions then* V^z *will converge to a unique fixed point which becomes identical to the true values* $\bar{V}^z$ *when* $\bar{c}, \bar{\rho} \to \infty$*.*

Proof. Under the assumptions the derivation in lemma A.2 applies and we obtain the exponential contraction to the unique fixed point $\bar{V}^z$ given in eq. (34). The update eq. (35) is a stochastic relaxation of these expressions and therefore converge (Kushner & Yin, 2003). □

B. Simulation Details

Since the NIE is a product of two terms which are each close to zero when initialized we found it beneficial to train using an augmented objective of the form:

$$\lambda_1 \text{NIE} + \lambda_2 \left(\mathbb{E}\left[Z = 1 | \Pi = a\right] - \mathbb{E}\left[Z = 0 | \Pi = a\right]\right) + \lambda_3 \left(\mathbb{E}\left[Z = 1 | \Pi = b\right] - \mathbb{E}\left[Z = 1 | \Pi = a\right]\right) + \lambda_4 \mathcal{H}[Z] \tag{36}$$

where in typical fashion we include an entropy term $\mathcal{H}[Z]$ to prevent trivial definitions of Z.

B.1. Twostage details

We use a learning rate schedule for the neural parameters while learning Φ in section 3.2. The schedule was chosen such that reasonable convergence could be shown in the plots in figure 3. We use an RMSprop optimizer, with an initial learning rate of 0.003 which is reduced to 0.001 after 800 parameter updates, and again to 0.0003 after 2400 parameter updates.

B.2. Doorkey details

The doorkey environment is distributed as part of gym minigrid (Chevalier-Boisvert et al., 2018). We used the `Minigrid-Doorkey5x5-v0` environment with the following modifications:

- The environment was fully observed and the outer wall was clipped to reduce state size.
- The number of actions was reduced to the minimal necessary to complete the environment (pick up, use, up, right, down, left).
- The reward is sparse and given upon completion.

To encode the board, we start with a game board given as a $n \times n \times 3$ tensor. The first slice indicates object type (wall, free square, key, agent, etc.), the second indicates the color (and is discarded as irrelevant), and the last indicates state (i.e. player orientation and whether the door is opened or not). We selected a simple one-hot encoding scheme which consists of:

- Removing the player orientation and encoding it as a one-hot vector.
- Forming a $n \times n \times m$ representation corresponding to a one-hot encoding the first slice concatenate with a one-hot encoding of the last slice.
- Concatenating these two.

For the experiments, we divided the training into the following steps:

- Training the policy network π_a.

Table 1. Overview of parameters for twostage

Name	Value
γ	0.999
Learning rate	0.001
Gradient normalization	No
Optimizer	Adam
Network layers	0
Buffer size	500
Batch size	50
On policy steps	1
Off policy steps	5
Episodes	10'000

- Training the causal variable Φ.
- Training the policy network π_b.

To train the policy π_a, we used the parameters in table 1

For the causal variable Φ we used the modified cost function eq. (36) with $\lambda_2 = 1$, $\lambda_4 = 0.1$ and otherwise 0. The batch size was increased to 100 and buffer size to 1000 and the method was trained for 5000 episodes. We found gradient normalization to be important and in that case a learning rate of 0.01 was suitable. For μ_b we decreased the learning rate to 0.00025 and trained for an additional 5000 episodes starting from the same weights as π_a.

References

Chevalier-Boisvert, M., Willems, L., and Pal, S. Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid, 2018.

Kushner, H. and Yin, G. G. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

Bibliography

Agler, Robert and Paul De Boeck (November 2017). "On the Interpretation and Use of Mediation: Multiple Perspectives on Mediation Analysis." In: *Frontiers in psychology* 8, page 1984. ISSN: 1664-1078.

Allamanis, Miltiadis, Earl T. Barr, et al. (July 2018). "A Survey of Machine Learning for Big Code and Naturalness." In: *ACM Comput. Surv.* 51.4, pages 1–37. ISSN: 0360-0300.

Allamanis, Miltiadis, Pankajan Chanthirasegaran, et al. (June 2017). "Learning Continuous Semantic Representations of Symbolic Expressions." In: *Proceedings of the 34th International Conference on Machine Learning*. Edited by Doina Precup and Yee Whye Teh. Volume 70. Proceedings of Machine Learning Research. PMLR, pages 80–88. URL: http://proceedings.mlr.press/v70/allamanis17a.html.

Alon, Uri et al. (2020). "Structural Language Models of Code." In: *Proceedings of the 37th International Conference on Machine Learning*. Edited by Hal Daumé Iii and Aarti Singh. Volume 119. Proceedings of Machine Learning Research. PMLR, pages 245–256.

Anderson, Greg et al. (September 2020). "Neurosymbolic Reinforcement Learning with Formally Verified Exploration." In: *NeurIPS*. URL: http://arxiv.org/abs/2009.12612.

Andre, David and Stuart J. Russell (2001). "Programmable Reinforcement Learning Agents." In: *Advances in neural information processing systems*. Edited by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, pages 1019–1025.

— (2002). "State abstraction for programmable reinforcement learning agents." In: *Aaai/iaai*. aaai.org, pages 119–125.

Andreas, Jacob, Dan Klein, and Sergey Levine (November 2016). "Modular Multitask Reinforcement Learning with Policy Sketches." URL: http://arxiv.org/abs/1611.01796.

Bagaria, Akhil, Jason Crowley, et al. (June 2020). "Skill Discovery for Exploration and Planning using Deep Skill Graphs." URL: https://openreview.net/pdf?id=-mvAo5hWNp.

Bagaria, Akhil and George Konidaris (2020). "Option Discovery using Deep Skill Chaining." In: *ICLR*. URL: https://openreview.net/pdf?id=B1gqipNYwH.

Bahdanau, Dzmitry et al. (November 2018). "Systematic Generalization: What Is Required and Can It Be Learned?" URL: http://arxiv.org/abs/1811.12889.

Balog, Matej et al. (2016). "Deepcoder: Learning to write programs." In: *arXiv preprint arXiv:1611.01989*.

Barendregt, Henk, Wil Dekkers, and Richard Statman (June 2013). *Lambda Calculus with Types*. Cambridge University Press. ISBN: 9780521766142.

Barreto, André et al. (December 2020). "Fast reinforcement learning with generalized policy updates." In: *Proceedings of the National Academy of Sciences of the United States of America* 117.48, pages 30079–30087. ISSN: 0027-8424.

Barrett, David G. T. et al. (July 2018). "Measuring abstract reasoning in neural networks." URL: http://proceedings.mlr.press/v80/santoro18a/santoro18a.pdf.

Barto, Andrew G. and Sridhar Mahadevan (January 2003). "Recent Advances in Hierarchical Reinforcement Learning." In: *Discrete Event Dynamic Systems: Theory and Applications* 13.1, pages 41–77. ISSN: 0924-6703.

Bastani, Osbert, Yewen Pu, and Armando Solar-Lezama (May 2018). "Verifiable Reinforcement Learning via Policy Extraction." URL: http://arxiv.org/abs/1805.08328.

Bengio, Yoshua et al. (January 2019). "A Meta-Transfer Objective for Learning to Disentangle Causal Mechanisms." URL: http://arxiv.org/abs/1901.10912.

Berman, Gordon J., William Bialek, and Joshua W. Shaevitz (October 2016). "Predictability and hierarchy in Drosophila behavior." In: *Proceedings of the National Academy of Sciences of the United States of America* 113.42, pages 11943–11948. ISSN: 0027-8424.

Bertsekas, Dimitri P. (2011). "Dynamic programming and optimal control." In: *Belmont, MA: Athena Scientific*. URL: https://www.academia.edu/download/52019115/Dynamic_programming_optimal_control.pdf.

Besold, Tarek R. et al. (November 2017). "Neural-Symbolic Learning and Reasoning: A Survey and Interpretation." URL: http://arxiv.org/abs/1711.03902.

Bošnjak, Matko et al. (May 2016). "Programming with a Differentiable Forth Interpreter." URL: http://arxiv.org/abs/1605.06640.

Breiman, Leo et al. (2017). *Classification and regression trees*. Routledge.

Burke, Michael, Svetlin Penkov, and Subramanian Ramamoorthy (February 2019). "From explanation to synthesis: Compositional program induction for learning from demonstration." URL: http://arxiv.org/abs/1902.10657.

Chang, Michael B. et al. (December 2016). "A Compositional Object-Based Approach to Learning Physical Dynamics." URL: http://arxiv.org/abs/1612.00341.

Chen, Mark et al. (July 2021). "Evaluating Large Language Models Trained on Code." URL: http://arxiv.org/abs/2107.03374.

Chiappa, Silvia et al. (April 2017). "Recurrent Environment Simulators." URL: http://arxiv.org/abs/1704.02254.

Christakopoulou, Konstantina and Adam Tauman Kalai (September 2017). "Glass-Box Program Synthesis: A Machine Learning Approach." URL: http://arxiv.org/abs/1709.08669.

Das, Sreerupa, C. Lee Giles, and Guo-Zheng Sun (1992). "Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external

stack memory." In: *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*. Volume 14. Citeseer. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.8599&rep=rep1&type=pdf.

Dasgupta, Ishita et al. (January 2019). "Causal Reasoning from Meta-reinforcement Learning." URL: http://arxiv.org/abs/1901.08162.

Davis, Randall, Howard Shrobe, and Peter Szolovits (March 1993). "What is a knowledge representation?" In: *AI magazine* 14.1, pages 17–17. ISSN: 0738-4602.

Dechter, Eyal et al. (2013). "Bootstrap Learning via Modular Concept Discovery." In: *IJCAI*, pages 1302–1309.

Devin, Coline et al. (2019). "Compositional Plan Vectors." In: *Advances in Neural Information Processing Systems*. Edited by H. Wallach et al. Volume 32. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2019/file/00989c20ff1386dc386d8124ebcba1a5-Paper.pdf.

Dietterich, Thomas G. (May 1999). "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition." URL: http://arxiv.org/abs/cs/9905014.

Diuk, Carlos, Andre Cohen, and Michael L. Littman (July 2008). "An object-oriented representation for efficient reinforcement learning." In: *Proceedings of the 25th international conference on Machine learning*. ICML '08. Association for Computing Machinery, pages 240–247. ISBN: 9781605582054.

Ellis, Kevin, Lucas Morales, et al. (2018). "Dreamcoder: Bootstrapping domain-specific languages for neurally-guided bayesian program learning." In: *Proceedings of the 2nd Workshop on Neural Abstract Machines and Program Induction*. URL: https://uclmr.github.io/nampi/extended_abstracts/ellis.pdf.

Ellis, Kevin, Daniel Ritchie, et al. (July 2017). "Learning to Infer Graphics Programs from Hand-Drawn Images." URL: http://arxiv.org/abs/1707.09627.

Eppe, Manfred, Phuong D. H. Nguyen, and Stefan Wermter (November 2019). "From Semantics to Execution: Integrating Action Planning With Reinforcement Learning for Robotic Causal Problem-Solving." In: *Frontiers in robotics and AI* 6, page 123. ISSN: 2296-9144.

Espeholt, Lasse et al. (February 2018). "IMPALA: Scalable distributed deep-RL with Importance Weighted Actor-learner architectures." URL: http://proceedings.mlr.press/v80/espeholt18a/espeholt18a.pdf.

Ferguson, Chris and Richard E. Korf (1988). "Distributed Tree Search and Its Application to Alpha-Beta Pruning." In: *AAAI*. Volume 88. aaai.org, pages 128–132.

Ferreira, Candida (February 2001). "Gene Expression Programming: a New Adaptive Algorithm for Solving Problems." URL: http://arxiv.org/abs/cs/0102027.

Feser, John K., Swarat Chaudhuri, and Isil Dillig (June 2015). "Synthesizing data structure transformations from input-output examples." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Volume 50. ACM, pages 229–239. ISBN: 9781450334686.

Fu, Justin et al. (February 2019). "From Language to Goals: Inverse Reinforcement Learning for Vision-Based Instruction Following." URL: http://arxiv.org/abs/1902.07742.

Gaunt, Alexander L. et al. (August 2016). "TerpreT: A Probabilistic Programming Language for Program Induction." URL: http://arxiv.org/abs/1608.04428.

Gerstenberg, Tobias and Joshua B. Tenenbaum (2017). "Intuitive theories." In: *Oxford handbook of causal reasoning*, pages 515–548.

Graves, Alex, Greg Wayne, and Ivo Danihelka (October 2014). "Neural Turing Machines." URL: http://arxiv.org/abs/1410.5401.

Greff, Klaus et al. (March 2019). "Multi-Object Representation Learning with Iterative Variational Inference." URL: http://arxiv.org/abs/1903.00450.

Gulwani, Sumit (July 2010). "Dimensions in program synthesis." In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. PPDP '10. Association for Computing Machinery, pages 13–24. ISBN: 9781450301329.

Gulwani, Sumit, Oleksandr Polozov, and Rishabh Singh (July 2017). *Program Synthesis*. Foundations and Trends in Programming Languages. now. ISBN: 9781680832921.

Guo, Xiaoxiao et al. (2014). "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning." In: *Advances in Neural Information Processing Systems*. Edited by Z. Ghahramani et al. Volume 27. Curran Associates, Inc., pages 3338–3346.

Haarnoja, Tuomas et al. (January 2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." URL: http://arxiv.org/abs/1801.01290.

Hangl, Simon et al. (April 2020). "Skill Learning by Autonomous Robotic Playing Using Active Learning and Exploratory Behavior Composition." In: *Frontiers in robotics and AI* 7, page 42. ISSN: 2296-9144.

Harding, Simon et al. (July 2012). "MT-CGP: mixed type cartesian genetic programming." In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. GECCO '12. Association for Computing Machinery, pages 751–758. ISBN: 9781450311779.

Hasselt, Hado van, Matteo Hessel, and John Aslanides (June 2019). "When to use parametric models in reinforcement learning?" URL: http://arxiv.org/abs/1906.05243.

Hein, Daniel, Steffen Udluft, and Thomas A. Runkler (December 2017). "Interpretable Policies for Reinforcement Learning by Genetic Programming." URL: http://arxiv.org/abs/1712.04170.

Hill, Felix et al. (January 2019). "Learning to Make Analogies by Contrasting Abstract Relational Structure." URL: http://arxiv.org/abs/1902.00120.

Hindle, Abram et al. (April 2016). "On the naturalness of software." In: *Communications of the ACM* 59.5, pages 122–131. ISSN: 0001-0782.

Holland, John H. (July 1992). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control and artificial intelligence*. Complex Adaptive Systems. MIT Press. ISBN: 9780262082136.

Holtz, Jarrett, Arjun Guha, and Joydeep Biswas (August 2020). "Robot Action Selection Learning via Layered Dimension Informed Program Synthesis." URL: http://arxiv.org/abs/2008.04133.

Inala, Jeevana Priya et al. (2020). "Synthesizing Programmatic Policies that Inductively Generalize." In: *ICLR*. URL: https://openreview.net/pdf?id=S1l8oANFDH.

Izzo, Dario, Francesco Biscani, and Alessio Mereta (November 2016). "Differentiable Genetic Programming." URL: http://arxiv.org/abs/1611.04766.

Jaques, Miguel, Michael Burke, and Timothy Hospedales (May 2019). "Physics-as-Inverse-Graphics: Unsupervised Physical Parameter Estimation from Video." URL: http://arxiv.org/abs/1905.11169.

Jetchev, Nikolay, Tobias Lang, and Marc Toussaint (2013). "Learning Grounded Relational Symbols from Continuous Data for Abstract Reasoning." In: URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.360.6567.

Jong, Edwin D. de and Jordan B. Pollack (September 2003). "Multi-Objective Methods for Tree Size Control." In: *Genetic Programming and Evolvable Machines* 4.3, pages 211–233. ISSN: 1389-2576.

Joshi, Rajeev, Greg Nelson, and Keith Randall (May 2002). "Denali: a goal-directed superoptimizer." In: *SIGPLAN Not.* 37.5, pages 304–314. ISSN: 0362-1340.

Kamio, Shotaro, Hideyuki Mitsuhashi, and Hitoshi Iba (2003). "Integration of Genetic Programming and Reinforcement Learning for Real Robots." In: *Genetic and Evolutionary Computation — GECCO 2003*. Springer Berlin Heidelberg, pages 470–482.

Karlsson, Jonas (1997). "Learning to Solve Multiple Goals." PhD thesis. University of Rochester.

Kelly, Stephen and Malcolm I. Heywood (July 2017). "Multi-task learning in Atari video games with emergent tangled program graphs." In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17. Association for Computing Machinery, pages 195–202. ISBN: 9781450349208.

— (June 2018). "Emergent Solutions to High-Dimensional Multitask Reinforcement Learning." In: *Evolutionary computation* 26.3, pages 347–380. ISSN: 1063-6560.

Keuning, Hieke, Johan Jeuring, and Bastiaan Heeren (September 2018). "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises." In: *ACM Trans. Comput. Educ.* 19.1, pages 1–43.

Kipf, Thomas et al. (February 2018). "Neural Relational Inference for Interacting Systems." URL: http://arxiv.org/abs/1802.04687.

Krawiec, K. and P. Lichocki (2010). "Using Co-solvability to Model and Exploit Synergetic Effects in Evolution." In: *Parallel Problem Solving from Nature, PPSN XI*. Springer Berlin Heidelberg, pages 492–501.

Kurach, Karol, Marcin Andrychowicz, and Ilya Sutskever (November 2015). "Neural Random-Access Machines." URL: http://arxiv.org/abs/1511.06392.

Lázaro-Gredilla, Miguel et al. (January 2019). "Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs." In: *Science robotics*

4.26. ISSN: 2470-9476. DOI: 10.1126/scirobotics.aav3150. URL: http://dx.doi.org/10.1126/scirobotics.aav3150.

Levy, Andrew et al. (December 2017). "Learning Multi-Level Hierarchies with Hindsight." URL: http://arxiv.org/abs/1712.00948.

Lillicrap, Timothy P. et al. (September 2015). "Continuous control with deep reinforcement learning." URL: http://arxiv.org/abs/1509.02971.

Lin, Christopher H., Mausam, and Daniel S. Weld (August 2016). "A Programming Language With a POMDP Inside." URL: http://arxiv.org/abs/1608.08724.

Ling, Wang et al. (May 2017). "Program Induction by Rationale Generation : Learning to Solve and Explain Algebraic Word Problems." URL: http://arxiv.org/abs/1705.04146.

Liu, Xuan and Jie Fu (July 2019). "Compositional planning in Markov decision processes: Temporal abstraction meets generalized logic composition." In: *2019 American Control Conference (ACC)*. IEEE, pages 559–566. ISBN: 9781538679265.

Lyu, Daoming et al. (October 2018). "SDRL: Interpretable and Data-efficient Deep Reinforcement Learning Leveraging Symbolic Planning." URL: http://arxiv.org/abs/1811.00090.

Ma, Zhihao et al. (March 2021). "Learning Symbolic Rules for Interpretable Deep Reinforcement Learning." URL: http://arxiv.org/abs/2103.08228.

Mania, Horia, Aurelia Guy, and Benjamin Recht (March 2018). "Simple random search provides a competitive approach to reinforcement learning." URL: http://arxiv.org/abs/1803.07055.

McKay, Robert I. (2000). "Fitness Sharing in Genetic Programming." In: *GECCO*, pages 435–442.

Miller, Julian F. (2011). "Cartesian Genetic Programming." In: *Cartesian Genetic Programming*. Edited by Julian F. Miller. Springer Berlin Heidelberg, pages 17–34. ISBN: 9783642173103.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, et al. (December 2013). "Playing Atari with Deep Reinforcement Learning." URL: http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, et al. (February 2015). "Human-level control through deep reinforcement learning." In: *Nature* 518.7540, pages 529–533. ISSN: 0028-0836.

Moerman, W. (2009). "Hierarchical reinforcement learning: Assignment of behaviours to subpolicies by self-organization." PhD thesis. Utrecht University.

Montana, David J. (June 1995). "Strongly typed genetic programming." In: *Evolutionary computation* 3.2, pages 199–230. ISSN: 1063-6560.

Mwaura, Jonathan (July 2010). "Evolution of robotic behaviours using Gene Expression Programming." PhD thesis. University of Exeter. ISBN: 9781424469093. DOI: 10.1109/cec.2010.5586083. URL: https://ore.exeter.ac.uk/repository/bitstream/handle/10036/3493/MwauraJ.pdf?isAllowed=y&sequence=2.

Nabi, Razieh, Phyllis Kanki, and Ilya Shpitser (August 2018). "Estimation of Personalized Effects Associated With Causal Pathways." In: *Uncertainty in artificial intelligence: proceedings of the... conference. Conference on Uncertainty in Arti-*

ficial Intelligence 2018. ISSN: 1525-3384. URL: https://www.ncbi.nlm.nih.gov/pubmed/30643490.

Natarajan, Nagarajan et al. (July 2020). "Programming by Rewards." URL: http://arxiv.org/abs/2007.06835.

Odena, Augustus, Kensen Shi, et al. (2021). "Bustle: Bottom-up program synthesis through learning-guided exploration." In: *ICLR*. URL: https://openreview.net/pdf?id=yHeg4PbFHh.

Odena, Augustus and Charles Sutton (February 2020). "Learning to Represent Programs with Property Signatures." URL: http://arxiv.org/abs/2002.09030.

Osera, Peter-Michael and Steve Zdancewic (June 2015). "Type-and-example-directed program synthesis." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Volume 50. ACM, pages 619–630. ISBN: 9781450334686.

Parr, Ronald and Stuart J. Russell (1998). "Reinforcement learning with hierarchies of machines." In: *Advances in neural information processing systems*. URL: https://axon.cs.byu.edu/Dan/778/papers/Hierarchical%20Reinforcement%20Learning/Parr*.pdf.

Pawlak, T. P., B. Wieloch, and K. Krawiec (June 2015). "Semantic Backpropagation for Designing Search Operators in Genetic Programming." In: *IEEE Transactions on Evolutionary Computation* 19.3, pages 326–340. ISSN: 1941-0026.

Pearl, Judea (2001). "Direct and Indirect Effects." In: *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*. URL: https://ftp.cs.ucla.edu/pub/stat_ser/R273-U.pdf.

Penkov, Svetlin and Subramanian Ramamoorthy (May 2017). "Explaining Transition Systems through Program Induction." URL: http://arxiv.org/abs/1705.08320.

Piech, Chris et al. (2015). "Learning Program Embeddings to Propagate Feedback on Student Code." In: *Proceedings of the 32nd International Conference on Machine Learning*. Edited by Francis Bach and David Blei. Volume 37. Proceedings of Machine Learning Research. PMLR, pages 1093–1102.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. ISBN: 9780262162098.

Pierrot, Thomas et al. (July 2020). "Learning Compositional Neural Programs for Continuous Control." URL: http://arxiv.org/abs/2007.13363.

Polozov, Oleksandr and Sumit Gulwani (October 2015). "FlashMeta: a framework for inductive program synthesis." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Volume 50. OOPSLA 2015. Association for Computing Machinery, pages 107–126. ISBN: 9781450336895.

Ross, Don and David Spurrett (2007). "Notions of Cause: Russell's thesis revisited." In: *The British journal for the philosophy of science* 58.1, pages 45–76. ISSN: 0007-0882.

Rothkopf, Constantin A. and Dana H. Ballard (November 2013). "Learning and Coordinating Repertoires of Behaviors with Common Reward: Credit Assignment and Module Activation." In: *Computational and robotic models of the hierarchi-*

cal organization of behavior. Edited by Gianluca Baldassarre and Marco Mirolli. 2013th edition. Springer. ISBN: 9783642398742.

Russell, Bertrand (1912). "On the Notion of Cause." In: *Proceedings of the Aristotelian Society* 13, pages 1–26. ISSN: 0066-7374.

Salimans, Tim et al. (March 2017). "Evolution Strategies as a Scalable Alternative to Reinforcement Learning." URL: http://arxiv.org/abs/1703.03864.

Salomon, R. (2003). "The deterministic genetic algorithm: implementation details and some results." In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. IEEE. ISBN: 9780780355361. DOI: 10.1109/cec.1999.782001. URL: http://ieeexplore.ieee.org/document/782001/.

Schkufza, Eric, Rahul Sharma, and Alex Aiken (March 2013). "Stochastic superoptimization." In: *SIGARCH Comput. Archit. News* 41.1, pages 305–316. ISSN: 0163-5964.

Schmid, Ute (1999). *Iterative macro-operators revisited: Applying program synthesis to learning in planning*. URL: https://pdfs.semanticscholar.org/174e/692a8164d71c79776ddab573be8798f90579.pdf.

Schmidt, Michael and Hod Lipson (April 2009). "Distilling free-form natural laws from experimental data." In: *Science* 324.5923, pages 81–85. ISSN: 0036-8075.

Schulman, John, Sergey Levine, et al. (February 2015). "Trust Region Policy Optimization." URL: http://arxiv.org/abs/1502.05477.

Schulman, John, Filip Wolski, et al. (July 2017). "Proximal Policy Optimization Algorithms." URL: http://arxiv.org/abs/1707.06347.

Schulte, Phillip J. et al. (November 2014). "Q- and A-learning Methods for Estimating Optimal Dynamic Treatment Regimes." In: *Statistical science: a review journal of the Institute of Mathematical Statistics* 29.4, pages 640–661. ISSN: 0883-4237.

Shah, Ameesh et al. (July 2020). "Learning Differentiable Programs with Admissible Neural Heuristics." URL: http://arxiv.org/abs/2007.12101.

Shalizi, Cosma Rohilla and James P. Crutchfield (July 1999). "Computational Mechanics: Pattern and Prediction, Structure and Simplicity." URL: http://arxiv.org/abs/cond-mat/9907176.

Silver, David and Kamil Ciosek (June 2012). "Compositional Planning Using Optimal Option Models." URL: http://arxiv.org/abs/1206.6473.

Simmons-Edler, Riley, Anders Miltner, and Sebastian Seung (June 2018). "Program Synthesis Through Reinforcement Learning Guided Tree Search." URL: http://arxiv.org/abs/1806.02932.

Simpkins, Christopher and Charles Isbell (July 2019). "Composable Modular Reinforcement Learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01, pages 4975–4982. ISSN: 2374-3468.

Singh, Rishabh and Pushmeet Kohli (2017). "AP: artificial programming." In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Singh, Satinder Pal (May 1992). "Transfer of learning by composing solutions of elemental sequential tasks." In: *Machine learning* 8.3, pages 323–339. ISSN: 0885-6125.

Sipper, Moshe (2011). *Evolved to Win*. Lulu.com. ISBN: 9781470972837.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms." In: *Advances in Neural Information Processing Systems*. Edited by F. Pereira et al. Volume 25. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf.

Solar-Lezama, Armando (2008). "Program synthesis by sketching." PhD thesis. University of California, Berkeley. ISBN: 9781243994462. URL: https://people.csail.mit.edu/asolar/papers/thesis.pdf.

Steenkiste, Sjoerd van, Michael Chang, et al. (February 2018). "Relational Neural Expectation Maximization: Unsupervised Discovery of Objects and their Interactions." URL: http://arxiv.org/abs/1802.10353.

Steenkiste, Sjoerd van, Klaus Greff, and Jürgen Schmidhuber (June 2019). "A Perspective on Objects and Systematic Generalization in Model-Based RL." URL: http://arxiv.org/abs/1906.01035.

Sun, Shao-Hua, Te-Lin Wu, and Joseph J. Lim (September 2019). "Program Guided Agent." URL: https://openreview.net/pdf?id=BkxUvnEYDH.

Sutton, Richard S. and Andrew G. Barto (November 2018). *Reinforcement Learning, second edition: An Introduction*. MIT Press. ISBN: 9780262352703.

Sutton, Richard S., Doina Precup, and Satinder Singh (August 1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning." In: *Artificial intelligence* 112.1, pages 181–211. ISSN: 0004-3702.

Tenenbaum, Joshua B. et al. (March 2011). "How to grow a mind: statistics, structure, and abstraction." In: *Science* 331.6022, pages 1279–1285. ISSN: 0036-8075.

Tian, Yonglong et al. (January 2019). "Learning to Infer and Execute 3D Shape Programs." URL: http://arxiv.org/abs/1901.02875.

Topin, Nicholay and Manuela Veloso (May 2019). "Generation of Policy-Level Explanations for Reinforcement Learning." URL: http://arxiv.org/abs/1905.12044.

Uesato, Jonathan et al. (December 2018). "Rigorous Agent Evaluation: An Adversarial Approach to Uncover Catastrophic Failures." URL: http://arxiv.org/abs/1812.01647.

Verma, Abhinav, Hoang Le, et al. (2019). "Imitation-Projected Programmatic Reinforcement Learning." In: *Advances in Neural Information Processing Systems*. Edited by H. Wallach et al. Curran Associates, Inc. URL: http://papers.nips.cc/paper/9705-imitation-projected-programmatic-reinforcement-learning.pdf.

Verma, Abhinav, Vijayaraghavan Murali, et al. (April 2018). "Programmatically Interpretable Reinforcement Learning." URL: http://arxiv.org/abs/1804.02477.

Wang, Rui et al. (January 2019). "Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions." URL: http://arxiv.org/abs/1901.01753.

Watkins, Christopher J. C. H. and Peter Dayan (May 1992). "Q-learning." In: *Machine learning* 8.3, pages 279–292. ISSN: 0885-6125.

Whittle, R. et al. (April 2017). "Applying causal mediation methods to clinical trial data: What can we learn about why our interventions (don't) work?" In: *European journal of pain* 21.4, pages 614–622. ISSN: 1090-3801.

Wilson, Dennis G. et al. (June 2018). "Evolving simple programs for playing Atari games." URL: http://arxiv.org/abs/1806.05695.

Wiltschko, Alexander B. et al. (December 2015). "Mapping Sub-Second Structure in Mouse Behavior." In: *Neuron* 88.6, pages 1121–1135. ISSN: 0896-6273.

Xu, Danfei et al. (October 2017). "Neural Task Programming: Learning to Generalize Across Hierarchical Tasks." URL: http://arxiv.org/abs/1710.01813.

Xu, Tian, Ziniu Li, and Yang Yu (2020). "Error bounds of imitating policies and environments∗." In: *NeurIPS*. URL: https://proceedings.neurips.cc/paper/2020/file/b5c01503041b70d41d80e3dbe31bbd8c-Paper.pdf.

Yin, Pengcheng and Graham Neubig (April 2017). "A Syntactic Neural Model for General-Purpose Code Generation." URL: http://arxiv.org/abs/1704.01696.

Young, Halley, Osbert Bastani, and Mayur Naik (January 2019). "Learning Neurosymbolic Generative Models via Program Synthesis." URL: http://arxiv.org/abs/1901.08565.

Zaremba, Wojciech and Ilya Sutskever (May 2015). "Reinforcement Learning Neural Turing Machines - Revised." URL: http://arxiv.org/abs/1505.00521.

Zhang, Byoung-Tak and Heinz Mühlenbein (1995). "Balancing accuracy and parsimony in genetic programming." In: *Evolutionary computation* 3.1, pages 17–38. ISSN: 1063-6560.

Zhang, Shiqi and Mohan Sridharan (August 2020). "A Survey of Knowledge-based Sequential Decision Making under Uncertainty." URL: http://arxiv.org/abs/2008.08548.

Zuidberg Dos Martires, Pedro et al. (July 2020). "Symbolic Learning and Reasoning With Noisy Data for Probabilistic Anchoring." In: *Frontiers in robotics and AI* 7, page 100. ISSN: 2296-9144.