Hierarchical Variational Imitation Learning of Control Programs

Roy Fox [1] **Richard Shin** [2] **William Paul** [2] **Yitian Zou** [2]
Dawn Song [2] **Ken Goldberg** [2 3] **Pieter Abbeel** [2] **Ion Stoica** [2]

Abstract

Autonomous agents can learn by imitating teacher demonstrations of the intended behavior. Hierarchical control policies are ubiquitously useful for such learning, having the potential to break down structured tasks into simpler sub-tasks, thereby improving data efficiency and generalization. In this paper, we propose a variational inference method for imitation learning of a control policy represented by parametrized hierarchical procedures (PHP), a program-like structure in which procedures can invoke sub-procedures to perform sub-tasks. Our method discovers the hierarchical structure in a dataset of observation–action traces of teacher demonstrations, by learning an approximate posterior distribution over the latent sequence of procedure calls and terminations. Samples from this learned distribution then guide the training of the hierarchical control policy. We identify and demonstrate a novel benefit of variational inference in the context of hierarchical imitation learning: in decomposing the policy into simpler procedures, inference can leverage acausal information that is unused by other methods. Training PHP with variational inference outperforms LSTM baselines in terms of data efficiency and generalization, requiring less than half as much data to achieve a 24% error rate in executing the bubble sort algorithm, and to achieve no error in executing Karel programs.

1. Introduction

Autonomous agents that interact with their environment can learn to perform desired tasks when provided with informative learning signals. In imitation learning (IL), a teacher provides demonstrations of successful performance of the task, which consist of traces of sensory observations and the desired control actions during execution of the control policy of the teacher (Schaal, 1999; Argall et al., 2009; Ho & Ermon, 2016; Hussein et al., 2017) or of the learner (Ross et al., 2011; Ross & Bagnell, 2014; Sun et al., 2017). This places some burden on the teacher to correctly control the system, requiring at least an implicit knowledge of *how* to perform the task, rather than just *what* the task is. However, such implicit knowledge is in many cases more available to humans than a formal description of success criteria or a reward function — some things are "easier done than said", e.g. household chores or grammatical speech. Moreover, the rich supervision signal in IL is generally more informative than the reward signal provided in reinforcement learning (RL), potentially reducing the required amount of interaction with the system and the teacher.

Structured control, and in particular hierarchical control, has the potential to further improve data efficiency. Modularity and hierarchy facilitate specialization and abstraction, so that each distinct module of the controller can focus on simpler behavior that is useful in a subset of system states. This specialization reduces the complexity of the relevant features of the state and of the environment dynamics, allowing each module to have a simpler model that can be learned from less data.

This paper takes a hierarchical imitation learning (HIL) approach, by modeling the control policy as parametrized hierarchical procedures (PHP) (Fox et al., 2018), a program-like structure in which each procedure, in each step it takes, can either invoke a sub-procedure, take a control action, or terminate and return to its caller. Given a dataset of observation–action traces of the execution of a teacher control policy, we seek to train the parameters of all procedures in a learner control policy. Taken together, these trained procedures form a control program, with a conditional branching structure induced by each procedure's choice of sub-procedure calls. This structure is latent in the data and must be discovered, which is particularly challenging with deep hierarchies where multiple levels of nested procedures can call each other.

We propose hierarchical variational imitation learning (HVIL): imitation learning of hierarchical control policies

[1]Department of Computer Science, University of California, Irvine [2]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley [3]Department of Industrial Engineering and Operations Research, University of California, Berkeley. Correspondence to: Roy Fox <royf@uci.edu>.

via variational inference. We train an inference model to approximate the posterior distribution, given a demonstration, of the latent sequence of procedure calls and terminations that could generate that trace. We sample from this distribution to impute the latent variables and guide the training of the generative PHP.

We experiment with our method to train control programs in two domains. In the Bubble Sort domain, we learn a policy that performs bubble sort of a memory array. Using HVIL to train a PHP with a 4-level hierarchy of 6 procedures, we outperform a 4-layer LSTM baseline in data efficiency, requiring less than half as much data to achieve a 24% error rate. In the Karel domain, we learn to imitate the execution of programs in the Karel language. On some programs, PHP trained with HVIL achieves zero test errors with less than half as much data as an LSTM baseline. PHP also generalize better than LSTMs to longer executions than seen in training.

Our work extends SRNNs (Fraccaro et al., 2016) to hierarchically-structured discrete latent variables, making three contributions. First, we present a network architecture for an inference model of a hierarchical control policy. Second, we provide analysis of technical considerations that we found necessary for training discrete stochastic RNNs, namely analytic KL and Rao–Blackwellization of non-reparametrizable latent variables. Third, we identify and demonstrate a novel benefit of variational inference in the context of hierarchical imitation learning, leveraging acausal information unused by other methods to facilitate the decomposition of the control policy into simpler procedures. The code used in our experiments is available at `https://github.com/royf/hvil`.

2. Related Work

Frameworks of hierarchical control often include explicit or implicit call-stacks. In the popular options framework (Sutton et al., 1999), options can use "intra-option" control to call other options. StackRNNs (Joulin & Mikolov, 2015) can explicitly perform stack operations as part of their control. Neural Programmers–Interpreters (NPI) (Reed & De Freitas, 2015) control program execution by maintaining a call-stack of procedures and their arguments. Neural Program Lattices (Li et al., 2016) add the ability to train NPI when the hierarchical structure is latent in some of the traces, using lattices to group the exponentially many latent stack trajectories into a manageable set. Parametrized hierarchical procedures (PHP) (Fox et al., 2018) maintains a similar call-stack, and trains it with a level-by-level application of an exact-inference method. While StackRNN and NPI keep real vectors on the stack, call-stacks of options and PHP maintain discrete values, namely the *identifiers* of the called options or procedures. PHP, being inspired by procedural programming, differs from options by additionally maintaining a program counter for each procedure.

In this work, we propose to train PHP via a variational inference (VI) method. VI has been used for training autoencoders in unsupervised learning (Kingma & Welling, 2013; Rezende et al., 2014; Zhang et al., 2017), for time-series modelling (Ghahramani & Hinton, 2000; Kulkarni et al., 2014; Bayer & Osendorfer, 2014; Chung et al., 2015; Fraccaro et al., 2016), and recently for RL (Levine, 2018; Fellows et al., 2018; Achiam et al., 2018).

Our method builds on SRNN (Fraccaro et al., 2016), which was originally presented for unstructured continuous latent variables, extending it to hierarchical control policies. Our method also extends PHP (Fox et al., 2018), by training multi-level hierarchies jointly, rather than level-by-level.

3. Preliminaries

3.1. Imitation Learning as Stochastic Inference

We model an agent's interaction with its environment as a Partially Observable Markov Decision Process (POMDP). At time t, the environment is in state $s_t \in \mathcal{S}$, and emits an observation $o_t \in \mathcal{O}$. Upon seeing the observation, the agent makes a stochastic choice of an update for its internal memory state to $m_t \in \mathcal{M}$, and of an action $a_t \in \mathcal{A}$ (discrete, in this work), according to a policy $p_\theta(m_t, a_t|m_{t-1}, o_t)$. The environment then makes a stochastic transition to the next state and observation $p(s_{t+1}, o_{t+1}|s_t, a_t)$.

We consider the imitation learning (IL) setting in which a teacher provides a set of demonstrations $\mathcal{D} = \{x_j\}$. Each demonstration $x = o_0, a_0, o_1, a_1, \ldots, o_T, a_T$ is an observation–action trace induced by the execution of a teacher policy in the environment. Here a_T is the first occurrence in the sequence of a *termination* action $\varnothing$. The learning problem is to find the parameter θ of a policy p_θ that gives high likelihood to the observed data, i.e. to maximize $\log p_\theta(\mathcal{D}) = \sum_j \log p_\theta(x_j)$. Denoting by $z = m_0, \ldots, m_T$ the latent sequence of memory states, we have

$$\log p_\theta(x) = \log \sum_z p_\theta(z, x) \qquad (1)$$
$$= \log \sum_z \prod_{t=0}^{T} p_\theta(m_t, a_t|m_{t-1}, o_t) + \text{const},$$

with the constant in θ incorporating the log-probability of the environment steps $p(s_{t+1}, o_{t+1}|s_t, a_t)$.

When the memory update is non-deterministic, the support of z can have size exponential in the horizon T, which prevents direct optimization of (1). Many existing approaches either have no memory state, allowing only reactive policies $p_\theta(a_t|o_t)$, or more generally have deterministic memory updates $m_t = g_\theta(m_{t-1}, o_t)$, and by extension $z = g_\theta(x)$,

Algorithm 1 PHP Step

Require: top stack frame (h_i, τ_i), PHP step selector u_i, current observation o_t
  if $u_i = \varnothing$ $\{h_i$ terminates$\}$ **then**
    Pop the top stack frame
    If stack is empty, terminate episode with action $a_t = \varnothing$
  else
    Increment the top step counter τ_i
    if $u_i \in \mathcal{A}$ $\{h_i$ takes elementary action $u_i\}$ **then**
      Take action $a_t = u_i$ in the environment
    else if $u_i \in \mathcal{H}$ $\{h_i$ calls sub-procedure $u_i\}$ **then**
      Push $(u_i, 0)$ onto the stack
    end if
  end if

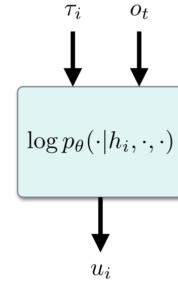

Figure 1. Illustration of a parametrized hierarchical procedure. A control program is a hierarchical policy assembled from a set of procedures, one for each $h \in \mathcal{H}$. A procedure can call another as sub-procedure by outputting the callee's identifier, or otherwise output the identifier of a control action to take it in the environment, or the indicator $\varnothing$ to terminate and return to its caller.

e.g. using recurrent neural networks (RNNs). This simplifies (1) to $\sum_t \log p_\theta(a_t|m_{t-1}, o_t)$, which permits direct optimization, e.g. using back-propagation through time.

Unfortunately, when the internal memory process z consists of discrete variables, such as the choice of procedures to invoke in a hierarchical structure, it is no longer representable as a sequence of differentiable memory-update functions. To allow gradient-based optimization methods, we relax the RNN to be stochastic, and optimize in the space of stochastic policies. As mentioned above, this in turn makes it infeasible to enumerate the space of latent memory trajectories.

Instead, we optimize the stochastic RNN with variational inference (Fraccaro et al., 2016). As detailed in Section 4, we train an *inference model* $q_\phi(z|x)$ from which we can sample z rather than exhaust its potentially large support. The sampled z then guides the training of the *generative model* $p_\theta(z, x)$, which in this work has the PHP structure described in the next section.

3.2. Parametrized Hierarchical Procedures

Structure in the policy p_θ is an inductive bias that can facilitate sample-efficient generalization from the demonstrated behavior to execution on unseen states. In this work, we use the hierarchical control framework of parametrized hierarchical procedures (PHP) (Fox et al., 2018), in which a control policy is modelled as a set of procedures $\mathcal{H}$. Each procedure is tasked with performing a specific behavior, which it does by breaking the task down into a sequence of simpler sub-tasks, and calling a sequence of sub-procedures and control actions to perform these sub-tasks.

Similarly to computer programs, the hierarchical controller's execution is managed by a call-stack, onto which sub-procedures are pushed when called, and from which they are popped when they terminate (Algorithm 1). Each *frame* (h, τ) in the stack consists of the one-hot encoded procedure identifier $h \in \mathcal{H}$ and the counter $\tau \in \mathbb{N}$ of steps that the procedure executed so far. Initially, the stack consists of a single frame, $(h_0, 0)$, where h_0 is a fixed *root* procedure. Let step i of the execution occur in time t, and let the top stack frame be (h_i, τ_i). Then procedure h_i takes a stochastic step with distribution $p_\theta(u_i|h_i, \tau_i, o_t)$, where $u_i \in \mathcal{H} \cup \mathcal{A} \cup \{\varnothing\}$ is either a sub-procedure, a control action, or the termination indicator $\varnothing$. Respectively in these three cases: (1) if $u_i \in \mathcal{H}$, the frame $(u_i, 0)$ is pushed into the stack, calling the sub-procedure $h_{i+1} = u_i$; (2) if $u_i \in \mathcal{A}$, the action $a_t = u_i$ is taken in the environment, and h_i remains the active procedure; or (3) if $u_i = \varnothing$, then the frame (h_i, τ_i) is popped from the stack, and h_i's caller (now at the top) resumes control. In either of the cases (1) or (2), h_i's step counter τ_i is incremented by 1. The episode ends with $a_T = \varnothing$ when the root terminates.

For each h, the PHP $p_\theta(u|h, \tau, o)$ is represented by a neural network (Figure 1), and jointly they all induce the stochastic process $z = \{u_i\}$. Note that the observation o_t is not stored on the stack, and that the time t advances only in PHP steps i that take control actions $a_t = u_i \in \mathcal{A}$.

Each PHP can in principle call any other PHP, including itself recursively. In practice, it is useful to define a *call-graph* over procedures, with edges from each PHP to those it can call as sub-procedures. The call-graph is a way for users to specify any available prior knowledge on the desired program structure. In our experiments, unless otherwise specified, we restrict call-graphs to be full 5-ary trees.

3.3. Variational Inference

Since we cannot directly optimize the objective (1), we will optimize a proxy, namely the evidence lower bound (ELBO). For any distribution $q_\phi(z|x)$, we have

$$\log p_\theta(x) \geqslant \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{p_\theta(z, x)}{q_\phi(z|x)}\right], \tag{2}$$

Algorithm 2 HVIL

```
Initialize p_θ and q_φ
loop
  Sample batch of demonstration traces from 𝒟
  Initialize g = 0
  for each trace x do
    Get posterior context {b_t} from bidirectional RNN
    Initialize t = 0, ℓ = 0
    for each PHP step i do
      Let (h_i, τ_i) be the top stack frame
      D ← 𝔻[q_φ(u_i|h_i, τ_i, b_t) ‖ p_θ(u_i|h_i, τ_i, o_t)]
      g ← g + ∇_{θ,φ} D + Dℓ
      Sample u_i|h_i, τ_i, b_t ∼ q_φ
      ℓ ← ℓ + ∇ log q_φ(u_i|h_i, τ_i, b_t)
      Operate the stack using Algorithm 1
      If u_i ∈ 𝒜, set t ← t + 1
    end for
  end for
  Take gradient descent step g
end loop
```

where the bounding gap is $\mathbb{D}[q_\phi(z|x)\|p_\theta(z|x)]$, the Kullback–Leibler (KL) divergence of the inference model $q_\phi(z|x)$ from the true, computationally infeasible posterior of the generative model $p_\theta(z|x)$. Maximizing the lower bound on the right-hand side of (2) over θ and ϕ is therefore trading off maximizing our log-likelihood objective on the left-hand side, with minimizing the gap.

We optimize the ELBO by sampling a batch of traces from the dataset, for each trace x sampling $z|x \sim q_\phi$, and computing the *score*

$$f_{\theta,\phi}(z,x) = \log \frac{p_\theta(z,x)}{q_\phi(z|x)}. \tag{3}$$

The gradient of f with respect to θ is an unbiased estimator for that of the ELBO, but the same is not true with respect to ϕ due to the dependence of the sampling distribution on ϕ. Since z is discrete and non-reparametrizable, we use the score-function trick, estimating ELBO's gradient as

$$\nabla f(z,x) + f(z,x)\nabla \log q_\phi(z|x). \tag{4}$$

4. Hierarchical Variational Imitation Learning

4.1. Variational Inference of Control Programs

Hierarchical Variational Imitation Learning (HVIL), summarized in Algorithm 2, uses variational inference to learn a PHP from demonstrations. Following the notation of Section 3, we need to be able to compute $\log p_\theta(z,x)$ and $\log q_\phi(z|x)$, and to sample from $q_\phi(z|x)$. For the control structure defined in Section 3.2, we have

$$\log p_\theta(z,x) = \sum_t \sum_{i\in\mathcal{I}_t} \log p_\theta(u_i|h_i,\tau_i,o_t) + \text{const},$$

where $\mathcal{I}_t$ are the PHP steps taken in time t, and the constant is the same as in (1). Similarly, it is convenient to choose for q_ϕ the structure

$$\log q_\phi(z|x) = \sum_t \sum_{i\in\mathcal{I}_t} \log q_\phi(u_i|h_i,\tau_i,x).$$

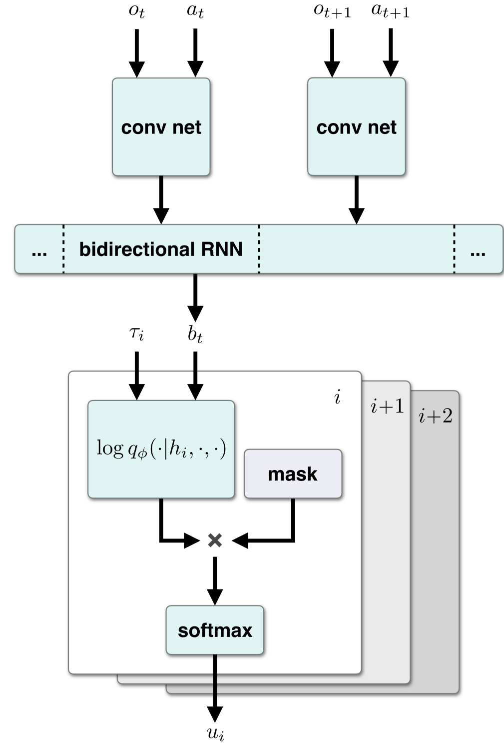

Figure 2. Architecture of an inference procedure. The demonstration trace $x = \{o_t, a_t\}$ is processed by a bidirectional RNN to generate a posterior context b_t, which feeds into a network computing a distribution $\log q_\phi(u_i|h_i,\tau_i,b_t)$ over the PHP step (sub-procedure call, control action, or termination). The u_i logits are masked to preclude PHP steps that are inconsistent with the trace x.

This structure suggests parametrizing q_ϕ using a second copy of the set of procedures used by p_θ. Such *inference procedures*, illustrated in Figure 2, would operate similarly to the original *generative procedures*, with two differences. First, the distribution q_ϕ is conditioned on the entire trace x, both its past and future, which implies that inference procedures should get x as input. Following the SRNN architecture (Fraccaro et al., 2016), we process the trace x using a deterministic bidirectional RNN $b_\phi(x)$, to obtain a time-dependent *posterior context* b_t. We use b_t as input to inference procedures in the same way that we input o_t to

generative procedures, yielding $q_\phi(u_i|h_i, \tau_i, b_t)$. Note that gradients of q_ϕ and p_θ are not back-propagated through the discrete variables h_i, τ_i, or u_i, but are back-propagated to the bidirectional RNN through b_t.

The second difference from generative procedures is that inference procedures are constrained to make choices that includes all the correct actions of the demonstrated trace, i.e. having $u_i = a_t$ for the last i in $\mathcal{I}_t$ for all t. Otherwise, q_ϕ has support for inconsistent values of z for which $p_\theta(z, x) = 0$, leading to an undefined ELBO. To achieve consistency, we mask the output of the active inference procedure to set the logit of any inconsistent u_i to $-\infty$, before normalizing q_ϕ with softmax. If any prior information is available in the form of a call-graph, restricting which sub-procedures each procedure can call, then we ensure that inference procedures only take paths in this graph that can lead to the known action a_t. In particular, we preclude any control action selection other than a_t.

4.2. Variance Reduction

Analytic KL computation. Computing the ELBO (2) "analytically", with an explicit sum over all values of z, often has better convergence properties than sampling $z|x \sim q_\phi$ (Kingma & Welling, 2013). This is usually attributed to the high variance of the score (3), particularly when $q_\phi(z|x)$ is a poor approximation of the true posterior $p_\theta(z|x)$. In our experiments, we found that analytic KL was indeed necessary, but for a different reason: the ELBO gradient estimator (4) becomes effectively biased as $q_\phi(z|x)$ tend to 0 for some values of z that are supported by p_θ. This occurs frequently in our setting when q_ϕ is successful in using future information in x, unavailable to p_θ, to rule out "incorrect" latent structure.

We prove the existence of this effect in a simple binary example, where $q(z|x)$ is ϵ for $z = 0$, and $1 - \epsilon$ for $z = 1$. We note that, as $\epsilon \to 0$, samples of z of any practical size eventually cease to include instances of $z = 0$. The score gradient estimator (to maximize) then becomes

$$\begin{aligned}
&\partial_\epsilon f + f\, \partial_\epsilon \log q(z|x) \\
&\quad = -\,\partial_\epsilon \log(1-\epsilon) + \frac{\log p(1,x)}{\log(1-\epsilon)}\, \partial_\epsilon \log(1-\epsilon) \\
&\quad \approx 1 - \frac{\log p(1,x)}{1-\epsilon} > 0.
\end{aligned}$$

Intriguingly, even as q approaches the correct distribution and samples correct values of z, the gradient indicates, incorrectly, that the objective would be improved by increasing ϵ. This would prevent the convergence of q to a correct deterministic distribution.

To compute the ELBO (2) analytically, we must factorize the exponentially large support of q_ϕ by breaking the ELBO down into individual PHP steps, and then compute the KL of each step with an exhaustive sum. We define

$$D_{t,i}(u_{<i}, x) = \mathbb{D}[q_\phi(u_i|u_{<i}, x) \| p_\theta(u_i|u_{<i}, o_t)],$$

such that the ELBO is

$$-\sum_t \sum_{i \in \mathcal{I}_t} \mathbb{E}_{u_{<i}|x \sim q_\phi}[D_{t,i}(u_{<i}, x)]. \tag{5}$$

For each PHP step i, we sample $u_{<i} = u_0, \ldots, u_{i-1}$ by extending $u_{<i-1}$ from the previous step, and compute the one-step KL $D_{t,i}(u_{<i}, x)$ analytically. This process is summarized in Algorithm 2.

Rao–Blackwellization of analytic-KL score gradient estimators. Rao–Blackwellization is the reduction of the variance of an estimator by taking its expectation with respect to a sufficient statistic. The score gradient estimator (4) can be Rao–Blackwellized by including in $\nabla \log q_\phi(z|x)$ only factors of z that are used in computing f (Schulman et al., 2015).

We leverage our ELBO factorization (5) to propose a Rao–Blackwellization of a score gradient estimator that is computed with analytic KL:

$$\sum_t \sum_{i \in \mathcal{I}_t} (\nabla D_i(u_{<i}, x) + D_i(u_{<i}, x) \nabla \log q_\phi(u_{<i}|x)).$$

4.3. Leveraging Acausal Information

In this section we identify a source of information in demonstration traces that, to our knowledge, has previously been unused. We show that HVIL can extract this information and leverage it to improve data efficiency. We demonstrate this principle with an illustrative example.

Consider a conditional generative model $p(y|x)$. There may exist a variable z, such that knowing z at training time breaks down the problem into two easier learning problems: of estimating $p(z|x)$ and $p(y|x, z)$. For example, let x be an image, z a label in one of 10 classes, and y an arbitrary partition of the classes into two categories. Such y is less informative than z about important image features, and its dependence on those features is higher-order than that of z, leading to higher sample complexity when z is latent.

Similar arrangements can occur naturally in imitation learning problems. Consider a robot learning to manipulate objects on a table. The robot perceives an image x of an object, and decides on an action y, e.g. pick or push. Later, the robot reaches for the object to pick or push it, and can take a closer and clearer image, in which the object's class z is obvious. Since z is not causally linked to y, most imitation learning algorithms do not extract from z any information useful for training the policy $p(y|x)$.

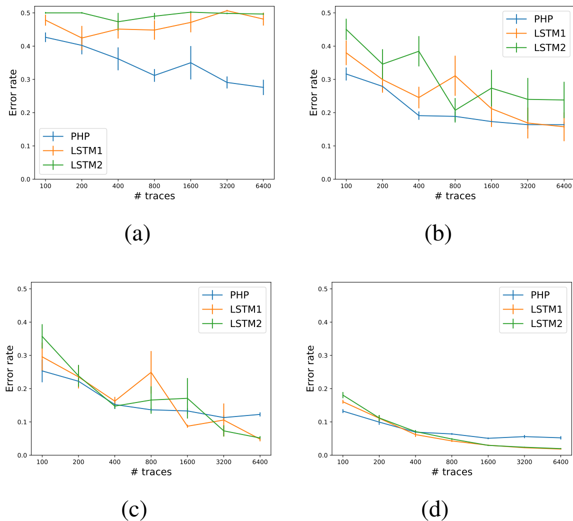

Figure 3. Error rate of imitating a teacher outputting the parity of MNIST digits and terminating. In training time, the teacher demonstrates the correct parity with probability 0.6, 0.7, 0.8, and 1, respectively in (a)–(d). The number of demonstration traces ranges from 100 to 6400 on a log scale. The error rate is averaged over 10 runs. Our method, 2-level PHP trained with HVIL, outperforms both 1- and 2-layer LSTM baselines in the low-data, high-noise regime. This indicates that HVIL leverages acausal information in the data to achieve improved data efficiency.

To illustrate this phenomenon, we define a simple POMDP in which o_0 is an MNIST image and o_1 is a one-hot encoding of the digit in that image. The teacher selects a binary action a_0 that equals the digit's parity with probability p varying from 0.6 to 1, and terminates on $a_1 = \varnothing$. An RNN trained with back-propagation through time to minimize the cross-entropy from teacher demonstrations has no gradient in time $t = 1$, because a_1 is constant. Although o_1 is very informative, it is never used in training time because it cannot be used to decide a_0 in execution time.

We generate datasets of between 100 and 6400 traces, and train three models: (1) a 64-unit LSTM, fed by a standard ConvNet[1]; (2) a similar 2-layer LSTM; and (3) a 2-level PHP with one root and 5 sub-procedures. The generative procedures are the same standard ConvNet, and the inference procedures use a 32-unit bidirectional LSTM, fed by a ConvNet and feeding into a multi-layer perceptron (MLP) with a 64-unit hidden layer.

The results, summarized in Figure 3, show a clear benefit to PHP trained with HVIL in the low-data, high-noise regime. Even without noise, our method outperforms LSTM when only 100 demonstrations are given. This suggests that the inference procedures were indeed able to infer latent structure z that helps guides training of the generative procedures.

[1]The architecture for the MNIST ConvNet is taken from https://github.com/pytorch/examples/blob/master/mnist/main.py

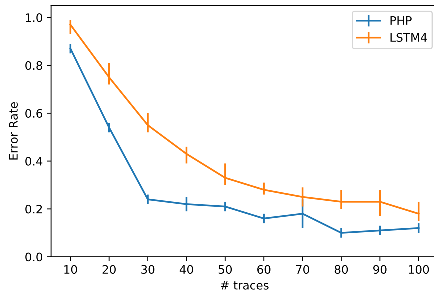

Figure 4. Error rate of imitating a teacher to perform bubble sort. We compare training a 4-level 6-procedure PHP via HVIL, with end-to-end training of a 4-level LSTM. The error rate is the fraction of test traces with at least one wrong action, averaged over 3 runs. HVIL is able to extract more information from the same data by discovering a latent hierarchical structure that generalizes better, requiring less than half as much data to achieve a 24% error rate.

5. Experiments

5.1. Bubble Sort

We train a hierarchical control policy to perform the bubble sort algorithm on a list of integers stored in external memory. The environment is an emulator of a computation device with two memory pointers. The actions can move either pointer left or right, or swap the memory values between the pointers. Each observation consists of the values addressed by the two pointers as one-hot encodings, as well as indicators for each pointer addressing the first or last integer.

Experiment setup. A teacher generates traces of the execution of the bubble sort algorithm, on lists of up to 10 integers. We use HVIL to learn a 4-level PHP, consisting of 6 procedures arranged in a partial binary tree. Each generative procedure is represented by a MLP with a 100-unit hidden layer. The inference procedures have the same architecture, applied to the output of a 32-unit bidirectional LSTM instead of the observation.

Every procedure in the hierarchy can take any control action, and in principle could perform the entire task. To encourage diversity between the procedures and avoid mode collapse into a non-hierarchical solution, we add an entropy regularization term on the inference procedures with an initial weight of 1 and exponential decay of 0.7 every 5000 steps.

We compare this model to a 4-layer 64-unit LSTM, fed by a 64-unit MLP and feeding into another 64-unit MLP to which softmax is applied to select the action.

The optimization algorithm is Adam with weight decay 10^{-3} and PyTorch default hyperparameters. Each batch consists of 10 traces. Each model is trained for 100K steps from demonstration datasets of sizes varying between 10 and 100.

Results. Figure 4 summarizes the experiment results. As expected, the error rate drops as larger datasets are used, and it drops faster for HVIL training of PHP than for the LSTM baseline. These results suggest that HVIL is able to extract more information than end-to-end RNN training from the same amount of data, and to use it to execute the bubble sort algorithm more reliably. With 30 demonstrations, HVIL already generalizes to reproduce the exact action sequence in 76% of test traces, while RNN requires more than 70 demonstrations to achieve that level of performance.

5.2. Karel

Karel is an educational programming language (Pattis, 1981) used in introductory classes. In the Karel language, the programmer writes programs (complete with constructs like `if`, `ifElse`, and `while`) that produce sequences of actions for an agent (a robot named Karel) which lives in an $m \times n$ grid world. Each cell in the grid can contain a wall, or up to 10 *markers*. The agent can occupy cells containing markers, but not walls. The actions available to Karel are `move`, `turnLeft`, `turnRight` to advance along its current orientation or change the orientation, and `pickMarker`, `putMarker` to decrease or increase the number of markers in the agent's current cell. The agent observes the values of `leftIsClear`, `rightIsClear`, `frontIsClear`, and `markersPresent`, which are the predicates (along with their negations) that can appear as the condition for `if`, `ifElse`, and `while`.

Karel has been used in past work on program synthesis and induction (Devlin et al., 2017; Bunel et al., 2018; Shin et al., 2018; Chen et al., 2019). In their setting, the method receives a number of input and output pairs that specify the semantics of an unknown program. Then, the model should either learn to directly operate the robot to reproduce the correct final grid for unseen input grids (*program induction*), or output a program in the Karel language that matches the unknown program (*program synthesis*). In our work, we perform program induction, represented as PHP, by imitating complete demonstration traces of a target Karel program. That is, the trained PHP should performs the same actions as that Karel program on test inputs.

Experiment setup. We evaluate HVIL on the 7 Karel programs from *Hour of Code* which contain loops (`https://bit.ly/karel-dataset`); they are listed in Appendix A. We prepend `turnRight` to each program, so that at least one step is taken before termination, as PHP requires. In our experiments, we sought to compare the training data efficiency as well as generalization ability of our PHP training method, compared to an RNN baseline.

To generate our training and test data, we follow the methodology of Bunel et al. (2018). We randomly sample a set of 5 initial grid states. For each grid, we: (1) randomly pick a world size between 2×2 and 16×16, excluding the outer walls; (2) sample whether each cell should contain walls or markers from a Bernoulli distribution (with its parameter drawn from a normal distribution); (3) for the marker-containing cells, sample a marker count from a geometric distribution; and (4) sample a location and orientation for the agent amongst the cells lacking walls. After sampling 5 grids, we execute the program on them and check whether they execute without crashing (without attempting to move into a wall) and sufficiently exercise all paths through it (i.e. for each statement in the program, at least one grid causes it to be executed; each branch of if/while blocks are taken at least once). If they do, we record the demonstrations using these initial grid states and add them to our dataset; otherwise, we reject them. We repeat this process until we have a total of 10,000 demonstrations for each program.

We compare four different models. The first is an LSTM baseline containing two layers with 64 hidden units each, fed by a 64-unit MLP to process the observation and feeding into another 64-unit MLP to decide the action (where each MLP contains two fully-connected layers with a ReLU activation function in between). The other three models are PHPs with full 5-ary trees of depth 1, 2, and 3 as call-graphs, each containing 1, 6, and 31 procedures respectively with a 256-unit hidden layer. As the call-graphs are full 5-ary trees, they encode no domain-specific information about the structure of the learned program.

Data efficiency experiments. In these experiments, we compare the performance of the four models on a test set containing 100 demonstrations, after training on 10, 20, 40, or 80 demonstrations randomly sampled from the remaining 9900 demonstrations. Figure 5 summarizes the results.

Overall, both kinds of models benefit from having a greater number of training demonstrations. However, our results suggest that HVIL training of PHPs can again extract more information from the same amount of data compared to the LSTM baseline, as they generally achieve lower error. The depth of the hierarchy has varying effect on the accuracy: in program C, the 3-level PHP performs best, but in program A worst.

Generalization experiments. In these experiments, we compare the performance of the four models on the ability to generalize to inputs requiring longer execution times than those seen during training. From the 10,000 demonstrations

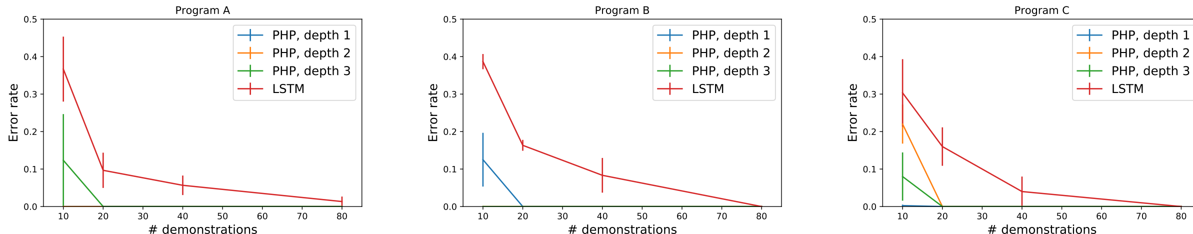

Figure 5. Error rate in imitation learning of Karel programs for datasets of training demonstrations of sizes 10 to 80. We compare three different PHP arrangements, full 5-ary trees of depths 1 to 3, trained via HVIL, as well as an end-to-end trained LSTM baseline. Results for more Karel programs are included in Appendix B.

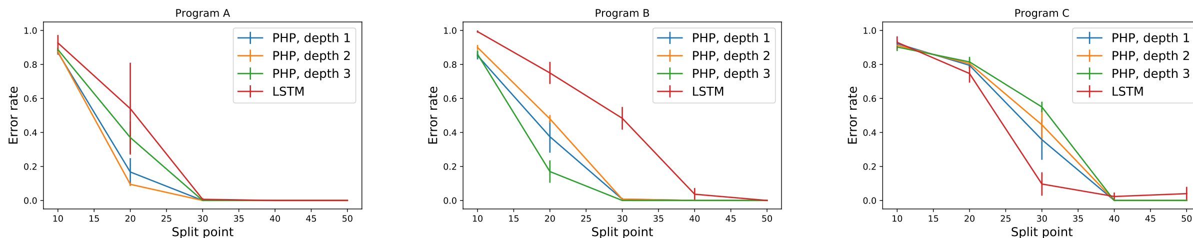

Figure 6. Error rate in imitation learning of Karel programs trained on demonstrations that are shorter than the test data. As the split point increases from 10% to 50%, longer demonstrations are included in the training dataset; however, test demonstrations are always longer still. Except on program C, the hierarchical control policies exhibit better generalization. Results for more Karel programs are included in Appendix B.

for each program, we computed the set of unique demonstration lengths seen throughout those demonstrations. For each set of demonstration lengths, we computed the 10th, 20th, 30th, 40th, and 50th percentile lengths. For each Karel program and for each percentile, we create a training and test set where the training set consists of 250 demonstrations with length less than that percentile trace length, and the test set consists of 100 demonstrations with length at least that percentile trace length. Thus, each test demonstration is longer than every training demonstration. Figure 6 summarizes the results.

On all but program C, our results suggest that the LSTM exhibits poorer generalization to longer test demonstrations than HVIL. We hypothesize that the hierarchical structure of PHPs acts as a prior which makes it more likely that they will learn the correct procedures of the program, which can perform reliably on longer demonstrations than ever seen in training.

6. Conclusion

In this paper we proposed HVIL, a variational inference method for training hierarchical control policies from demonstrations in which the structure is latent, including a novel architecture for the inference model. We detailed variance reduction methods that we found necessary for successful training. We also identified a novel benefit of HVIL, namely the ability to leverage acausal information in decomposing a task into a hierarchy of simpler procedures.

Our method can benefit from further innovation both in variational inference and more generally in stochastic optimization. We expect that introducing control variates, such as RELAX (Grathwohl et al., 2017), can significantly reduce the variance of the gradient estimators and improve convergence rates and robustness. Tighter bounds on the log-likelihood objective, such as IWAE (Burda et al., 2015), can also help in cases where q_ϕ is insufficiently expressive to match the posterior, however an open question is how to combine such bounds with the variance reduction methods of Section 4.2.

Training hierarchical policies with HVIL opens up the possibility of extending the model with components that enlarge the latent space but provide more expressiveness and regularization. For example, the procedures of a PHP can take arguments and terminate with return values (Fox et al., 2019). PHP would also be more expressive if they were augmented with a recurrent state.

In this work, we refrained from supplying meaningful prior knowledge in the form of a call-graph of procedures. In future work, we will explore the benefits of a compatible call-graph in improving data efficiency and generalization, and study data-driven methods for assisting the discovery of such call-graphs.

Acknowledgements

This research was performed at the RISELab at UC Berkeley in affiliation with the AUTOLAB and Berkeley AI Research (BAIR). This research was supported in part by: the NSF CISE Expeditions Award CCF-1730628; the NSF National Robotics Initiative Award 1734633; and the Scalable Collaborative Human-Robot Learning (SCHooL) Project. The authors were supported in part by donations from Alibaba, Amazon Web Services, Ant Financial, Arm, Autodesk, CapitalOne, Comcast, Ericsson, Facebook, Google, Hewlett-Packard, Honda, Huawei, Intel, Knapp, Microsoft, Nvidia, Scotiabank, Siemens, Splunk, Toyota Research Institute, and VMware.

References

Achiam, J., Edwards, H., Amodei, D., and Abbeel, P. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018.

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

Bayer, J. and Osendorfer, C. Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*, 2014.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1Xw62kRZ.

Burda, Y., Grosse, R., and Salakhutdinov, R. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1gfOiAqYm.

Chung, J., Kastner, K., Dinh, L., Goel, K., Courville, A. C., and Bengio, Y. A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pp. 2980–2988, 2015.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pp. 990–998, 2017.

Fellows, M., Mahajan, A., Rudner, T. G., and Whiteson, S. Virel: A variational inference framework for reinforcement learning. *arXiv preprint arXiv:1811.01132*, 2018.

Fox, R., Shin, R., Krishnan, S., Goldberg, K., Song, D., and Stoica, I. Parametrized hierarchical procedures for neural programming. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJl63fZRb.

Fox, R., Berenstein, R., Stoica, I., and Goldberg, K. Multi-task hierarchical imitation learning for home automation. In *15th IEEE Conference on Automation Science and Engineering (CASE)*, 2019.

Fraccaro, M., Sønderby, S. K., Paquet, U., and Winther, O. Sequential neural models with stochastic layers. In *Advances in neural information processing systems*, pp. 2199–2207, 2016.

Ghahramani, Z. and Hinton, G. E. Variational learning for switching state-space models. *Neural computation*, 12 (4):831–864, 2000.

Grathwohl, W., Choi, D., Wu, Y., Roeder, G., and Duvenaud, D. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *arXiv preprint arXiv:1711.00123*, 2017.

Ho, J. and Ermon, S. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pp. 4565–4573, 2016.

Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):21, 2017.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pp. 190–198, 2015.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Kulkarni, T. D., Saeedi, A., and Gershman, S. Variational particle approximations. *stat*, 1050:1, 2014.

Levine, S. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint arXiv:1805.00909*, 2018.

Li, C., Tarlow, D., Gaunt, A. L., Brockschmidt, M., and Kushman, N. Neural program lattices. 2016.

Pattis, R. E. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

Reed, S. and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*, 2014.

Ross, S. and Bagnell, J. A. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635, 2011.

Schaal, S. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

Schulman, J., Heess, N., Weber, T., and Abbeel, P. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pp. 3528–3536, 2015.

Shin, R., Polosukhin, I., and Song, D. Improving neural program synthesis with inferred execution traces. In *Advances in Neural Information Processing Systems*, pp. 8931–8940, 2018.

Sun, W., Venkatraman, A., Gordon, G. J., Boots, B., and Bagnell, J. A. Deeply aggrevated: Differentiable imitation learning for sequential prediction. *arXiv preprint arXiv:1703.01030*, 2017.

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

Zhang, R., Isola, P., and Efros, A. A. Split-brain autoencoders: Unsupervised learning by cross-channel prediction. In *CVPR*, volume 1, pp. 6, 2017.

A. List of Karel Programs

1. Program A: `def run() { turnRight(); while (noMarkersPresent()) { move(); if (rightIsClear()) { turnRight(); } } }`
2. Program B: `def run() { turnRight(); while (noMarkersPresent()) { move(); if (leftIsClear()) { turnLeft(); } } }`
3. Program C: `def run() { turnRight(); while (noMarkersPresent()) { if (rightIsClear()) { turnRight(); } move(); } }`
4. Program D: `def run() { turnRight(); while (noMarkersPresent()) { if (frontIsClear()) { move(); } else { turnLeft(); } } }`
5. Program E: `def run() { turnRight(); while (noMarkersPresent()) { if (frontIsClear()) { move(); } else { turnRight(); } } }`
6. Program F: `def run() { turnRight(); while (noMarkersPresent()) { if (frontIsClear()) { move(); } else { if (rightIsClear()) { turnRight(); } else { turnLeft(); } } } }`

B. All Karel Experiment Results

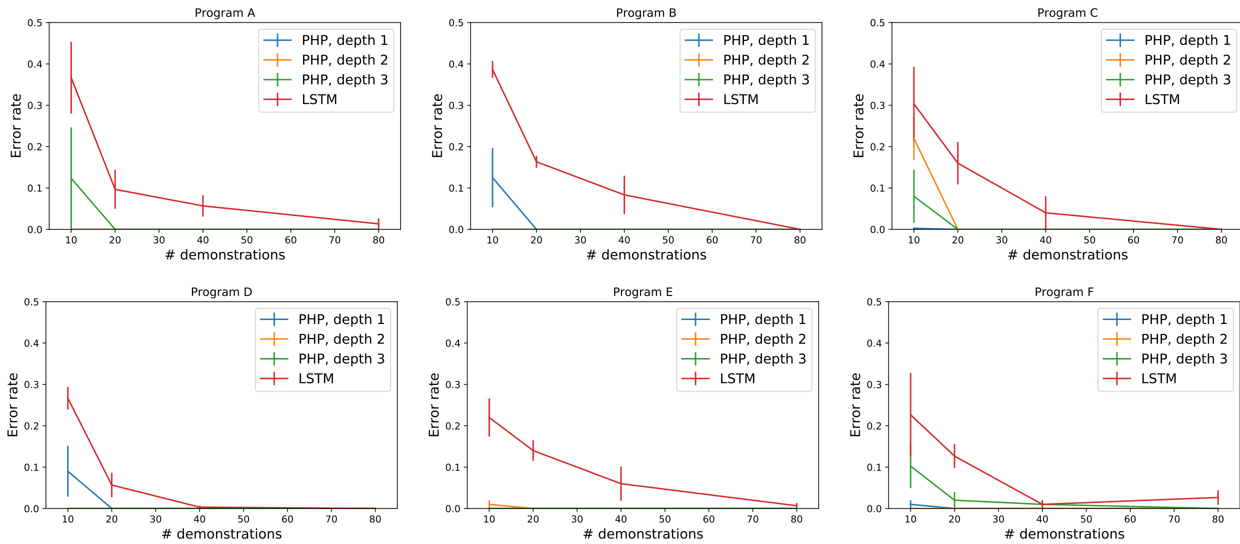

Figure 7. Error rate in imitation learning of Karel programs for datasets of training demonstrations of sizes 10 to 80.

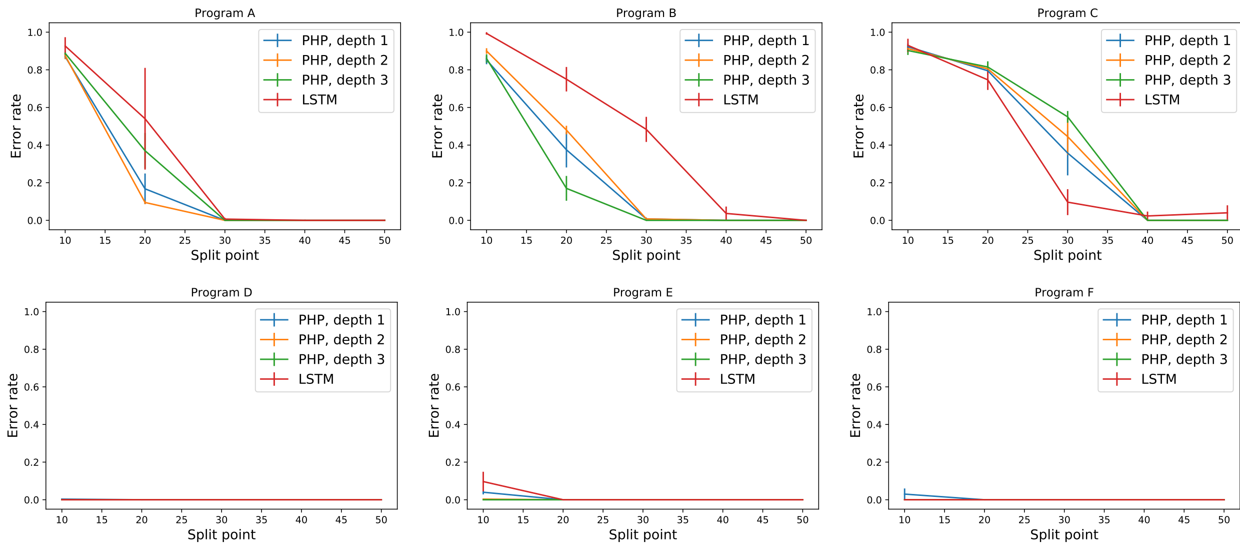

Figure 8. Error rate in imitation learning of Karel programs trained on demonstrations that are shorter than the test data.