Reinforcement Learning as Probabilistic Inference

Farzad Abdolhosseini
Department of Computer Science
University of British Columbia
Vancouver, BC
farzadab@cs.ubc.ca

Ainaz Hajimoradlou
Department of Computer Science
University of British Columbia
Vancouver, BC
ainaz@cs.ubc.ca

Jonathan W. Lavington
Department of Computer Science
University of British Columbia
Vancouver, BC
jola2372@colorado.edu

Abstract

In recent years, growth in computing power and an increased interest in more principled machine learning models has led to a resurgence of research into probabilistic inference. Such methodologies are desirable, as they lend themselves well to rigorous mathematical and statistical analysis, and as a result allow researchers to reason how a given algorithm will perform outside of the available test environments. In this article we will discuss how probabilistic inference might be utilized in the modeling of agent locomotion by reformulating existing model approaches within classical reinforcement learning. We will begin by deriving the graphical model for RL as inference, and then reformulate the problem as variational inference following [5]. We will then discuss how the problem can be implemented via maximum entropy policy gradients and variants of it, then provide examples and results on a 2d point mass problem. Next, we will provide an implementation and analysis of the same problem as a probabilistic program, and compare it to the previous implementation under maximum entropy. We will conclude with future directions, and useful heuristics for general implementation. The majority of this work is based off of the following papers [5, 11, 2] which discuss reinforcement learning as probabilistic inference, maximum entropy reinforcement learning, and variance reduction in gradient estimates.

1 Introduction

The principle task within reinforcement learning is teach an autonomous agent to determine the optimal policy (i.e. the policy that will culminate in the maximum possible reward) given the observed world state. Probabilistic inference allows us to re-frame the problem of reward maximization as a sampling problem. More specifically we can re-frame reward maximization as a graphical model where states with higher reward have a higher probability of occupancy. This distribution can then be approximated via variational inference, which leads to an objective similar to that of policy gradients; where the only difference between the two is that the advantage estimate is now regularized by the entropy of the policy. In this way we frame the problem as density estimation over the set of optimal trajectories, and in doing so allow the agent to directly reason about the uncertainty associated with its policy. This objective attempts to not simply find an agent that achieves a maximal reward within the space, but also attempts to find an agent that can pick uniformly amongst different optimal trajectories. Heuristically this may seem desirable, as it incentivises an agent to automatically explore the space. However in practice this means that additional measures have to be taken in order for updates to the

policy to remain stable, and convergence to an near optimal policy takes significantly longer then standard policy gradient updates. In many cases extremely sub optimal solutions that exhibit large entropy terms end up dominating the advantage, which leads not only to large swings in the gradient, but also local optimal that constitute extremely poor solutions to the problem.

While these issue can be mitigated when directly optimizing the function, additional weight is placed on the programmer when using a probabilistic programming system for approximate inference. Part of this paper will address how some of these issues can be either avoided, as well as adjustments to gradient updates of the evidence lower bound that can be inferred based upon observations of the direct implementation. Specifically, we will consider: how the addition of a baseline is necessary for variance reduction in the gradient of an agents policy, how natural gradient decent might be able to further reduce the variance of gradient updates, and finally a Bayesian form of policy gradient where a posterior over gradients are inferred by trajectory samples to further reduce variance. The analysis of the max entropy framework is then extended to the probabilistic programming setting, where the objective being optimized is equivalent. Unfortunately, we were not able to make many of these additions ourselves, but have laid the groundwork for how one might go about solving non-trivial reinforcement learning problems using probabilistic programming languages. Finally, although the discussion will be largely language agnostic, we will primarily focus on the probabilistic programming language Pyro, and its variational inference tool SVI.

2 Background and Related Work

The RL framework is motivated by the problem of learning via interaction with an environment. In this framework the agent learns by taking actions within the environment, observing changes in the world state, and collects an associated reward. More concretely, at each time step t, the agent observes the state of the environment s_t, which comes from the distribution $p(s_t)$, and takes an action denoted by a_t. The environment then transitions to a new state s_{t+1} based on the transition distribution $p(s_{t+1}|a_t, s_t)$ and outputs a reward $r(s_t, a_t)$. The states should follow the Markov property, meaning that each state s_{t+1} is conditionally independent of all previous states given s_t. If the actor can fully observe all information associated with the state of the world (baring dynamics), then the problem is known as a Markov decision process (MDP). If the agent is given an incomplete representation of the states, the problem is known as a partially observable MDP (POMDP). Solving a task in both cases typically involves finding a policy $p(a_t|s_t, \theta)$, which specifies the action that the agent will to take at each time step conditioned on observing the state s_t. Within this article we assume that the policy is distributed as a multivariate normal distribution parameterized by θ, and only focus on episodic tasks with a fixed time-horizon T.

2.1 Graphical Model

For simple cases, a standard RL approach known as policy search can be specified by the following optimization problem [5]:

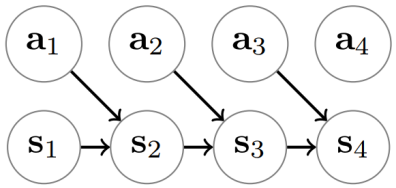
$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim p(s_t,a_t|\theta)} \left[r(s_t, a_t) \right]. \tag{1}$$

To re-frame this problem as inference, we can start by looking at the problem embedded in a PGM. In this case we can treat the states and the actions as random variables in the graphical model, and get the dependency graph using Markov assumptions of the environment, as displayed in 1a. In order to incorporate reward information we can include a set of binary random variables $\mathcal{O}_t$ which are defined as [1]:

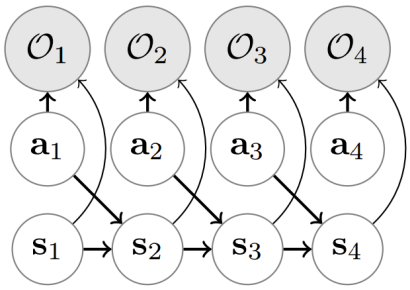
$$p(\mathcal{O}_t = 1|s_t, a_t) = \exp\left(r(s_t, a_t)\right) \tag{2}$$

The $\mathcal{O}_t$ variables can be thought of as indicating whether or not the action at time step t is *optimal*, and are therefore called the *optimality variables*. By conditioning on $\mathcal{O}_t = 1$ for all time steps, we

[1] Here we assume that the rewards are non-positive, but in other cases we can assume that the final model is a factor graph. It is also possible to transform rewards in order to make them non-positive, but this should be done with care as reward transformations can lead to modified optimality criteria.



(a) graphical model of states and actions [5]



(b) graphical model with optimality variables [5]

Figure 1: PGM structure used to turn a reinforcement learning problem into probabilistic inference.

arrive at the final representation of the PGM structure in Figure 1b with the $\mathcal{O}_t$s as observed variables. The posterior of this PGM can be thought of as a distribution over optimal trajectories[2],

$$\begin{aligned} p(\tau|o_{1:T}) \propto p(\tau, o_{1:T}) &= p(s_1)\prod_{t=1}^{T} p(\mathcal{O}_t = 1|s_t, a_t)p(s_{t+1}|s_t, a_t) \\ &= \left[p(s_1)\prod_{t=1}^{T} p(s_{t+1}|s_t, a_t)\right] \exp\left(\sum_{t=1}^{T} r(s_t, a_t)\right). \end{aligned} \tag{3}$$

This leads to a natural posterior over actions. More specifically, the probability of observing some trajectory is given as its probability to occur according to the dynamics times the exponential of the total reward along that trajectory, where deterministic dynamics reduces the objective to only the exponential of the total reward along the trajectory. Using this basic framework, we can now discuss how to implement, and improve upon it.

3 Maximum Entropy Reinforcement Learning as Probabilistic Inference

Having derived the joint distribution for PGM in Figure 1b, we can now attempt to inference inference. As we would like to determine a policy, but are only able to sample from optimal trajectories, the goal becomes approximateing this optimal trajectory distribution $p(y)$ which is hard to compute (and sample from), with a potentially simpler distribution $q(y)$ that we know how to compute. In our case, the desired distribution that we want to approximate is the posterior $p(\tau|o_{1:T})$ from 3. The approximate distribution $q(\tau)$ is:

$$q(\tau) = q(s_1)\prod_{t=1}^{T} q(s_{t+1}|s_t, a_t)q(a_t|s_t). \tag{4}$$

While we are only really interested in $q(a_t|s_t)$, neither the posterior dynamics $q(s_{t+1}|s_t, a_t)$ or the initial state $q(s_1)$ are specified, and may not necessarily match true dynamics $p(s_{t+1}|s_t, a_t)$ and $p(s_1)$. Incidentally, allowing the dynamics to vary can drastically change the policy the agent samples, leading to so called "risk seeking behavior"[5]. This is because, if the agent can affect how it samples states, it will always pick the policy that transitions to the highest reward, not necessarily the policy with the highest expected return. In order to solve this issue, we enforce the posterior dynamics and the initial state to exactly match the true dynamics.

$$q(\tau) = p(s_1)\prod_{t=1}^{T} p(s_{t+1}|s_t, a_t)q(a_t|s_t). \tag{5}$$

[2]Where the optimality just means that all the optimality variables are equal to one, which are in turn distributed according to the rewards. Therefore, every possible trajectory can be optimal but with a really low probability.

In order to perform variational inference, we need to derive an optimization objective known as the evidence lower bound (ELBO) which is given below:

$$\log p(\mathcal{O}_{1:T}) = \log \mathbb{E}_{(s_{1:T},a_{1:T})\sim q(s_{1:T},a_{1:T})}\left[\frac{p(\mathcal{O}_{1:T}, s_{1:T}, a_{1:T})}{q(s_{1:T}, a_{1:T})}\right]$$
$$\geq \mathbb{E}_{(s_{1:T},a_{1:T})\sim q(s_{1:T},a_{1:T})}\big[\log p(\mathcal{O}_{1:T}, s_{1:T}, a_{1:T}) - \log q(s_{1:T}, a_{1:T})\big]. \quad (6)$$

As both the dynamics and initial distribution match for both q and p, they cancel, and ELBO can be further simplified to[3]:

$$\log p(\mathcal{O}_{1:T}) \geq \mathbb{E}_{(s_{1:T},a_{1:T})\sim q(s_{1:T},a_{1:T})}\left[\sum_{t=1}^{T} r(s_t, a_t) - \log q(a_t|s_t)\right]. \quad (7)$$

The distribution $q_\theta(a_t|s_t)$ is parameterized with θ, which act as weights for the chosen model architecture. This distribution, which we will call policy, is also shown as $\pi_\theta(a_t|s_t)$. Now we are interested in optimizing parameters θ such that the objective $J(\theta)$ is minimized:

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim q(s_t,a_t)}\Big[(r(s_t, a_t) + \mathcal{H}(q_\theta(\cdot|s_t)))\Big], \quad (8)$$

where the gradient $\nabla_\theta J(\theta)$ is:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim q(s_t,a_t)}\left[\nabla_\theta \log q_\theta(a_t|s_t)\left(\sum_{t'=t}^{T} r(s'_t, a'_t) - \log q_\theta(a_{t'}|s_{t'}) - b(s_t)\right)\right]. \quad (9)$$

$b(s_{t'})$ is a state dependant baseline which helps to reduce the variance of the expectation. This is due to the fact that the gradient estimator is not affected by adding state dependant constants. If we are not interested in using a baseline, $b(s_{t'})$ should be replaced by 1 which comes from the derivative of the entropy term. From this definition, *surrogate loss* can be written as:

$$\hat{A}(s_t, a_t) = \left(\sum_{t'=t}^{T} r(s'_t, a'_t) - \log q_\theta(a_{t'}|s_{t'}) - b(s_t)\right) \rightarrow$$
$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim q(s_t,a_t)}\Big[\log q_\theta(a_t|s_t)\hat{A}(s_t, a_t)\Big], \quad (10)$$

where the $\hat{A}(s_t, a_t)$ term is treated as a scaling factor and should not be back-propagated through the computation graph[5]. This objective corresponds to the maximum entropy reinforcement learning agent, and only differs from the classical policy gradient objective in the addition of entropy regularization in the advantage.

In order to find the baseline, we generate a function that reduces the variance of the gradient estimator which can be defined as:

$$Var\left(\sum_{t=1}^{T} \nabla_\theta \log q_\theta(a_{t'}|s_{t'})\left[R(\tau) - b(s_t)\right]\right) \quad (11)$$

Where $R(\tau)$ is the expected cumulative reward of a trajectory τ. As one might guess, the easiest way to send the variance term above to zero, is to approximate $R(\tau)$ with $b(s_t)$, that is:

$$b(s_t) \approx \sum_{t'=t}^{T} r(a_{t'}, s_{t'}) - \log q_\theta(a_{t'}|s_{t'}) = R(\tau) \quad (12)$$

To see why this drives the variance towards zero more explicitly, consider:

$$\approx \sum_{t=1}^{T} \nabla_\theta \mathbb{E}_{(s_t,a_t)\sim q_\theta(s_t,a_t)}\left[\log q_\theta(a_{t'}|s_{t'})\left(R(\tau) - b(s_t)\right)\right]^2$$
$$\approx \sum_{t=1}^{T} \nabla_\theta \mathbb{E}_{(s_t,a_t)\sim q_\theta(s_t,a_t)}\left[\log q_\theta(a_{t'}|s_{t'})^2\right] \mathbb{E}_{(s_t,a_t)\sim q_\theta(s_t,a_t)}\left[\left(R(\tau) - b(s_t)\right)^2\right] \quad (13)$$

[3]Evidence here is $\mathcal{O}_t = 1$ for all $t \in \{1, ..., T\}$.

Where the first step follows from the assumption that the process is unbiased, and the second follows from the assumption that the values in the expectation are approximately independent. Thus as the baseline goes towards the expected cumulative reward at that state, the variance of the objective in policy gradients goes to zero (provided that the gradient is bounded).

4 Implementation

This section will detail how to implement maximum entropy policy gradients, both classically, as well as via a probabilistic programming language. Specifically, we will look at the addition of a critic, the effects of adding a temperature coefficient to the entropy term, and finally possible avenues to improve gradient approximations. We will then look at training the same variational objective in the Pyro probabilistic programming language, highlighting the ease of use for a general framework RL engine, as well as the shortcomings that arise from not being able to utilize some of the heuristics present in classical RL methodologies.

4.1 Maximum Entropy Policy Gradients

The most basic instance of maximum entropy policy gradients has several issues that have to be addressed. As was discussed above, a good place to start is the inclusion of a baseline. From (12), we know the approximate form of b_{s_t}, and can thus model it by solving the minimization problem:

$$\min_{\theta} \quad \frac{1}{M*T}\sum_{i=1}^{M}\sum_{t=1}^{T}\left(b_\theta(s_{t,i}) - \sum_{t'=t}^{T} r(a_{t',i}, s_{t',i}) - \log q(a_{t',i}|s_{t',i})\right)^2 \tag{14}$$

This leaves a very convenient way to approximate the baseline by directly minimizing the MSE of the baseline and the expected cumulative reward at a state given a set of trajectories. However the introduction of the baseline (or critic) does in practice drastically slow down the algorithm. Therefore in practice the critic was only trained once before the agent is updated, and then trained concurrently with the agents using a reduced number of steps. More efficient algorithms do exist, but were outside the scope of this project.

The next approach we looked at took to improve convergence of the algorithm was the method of natural gradients. In the context of RL, natural policy gradients as they are called [3], have improved convergence properties compared to simple gradient updates, and can be thought of as making gradient steps on the Reimannian manifold associated with the objective parameters θ. This transformation is achieved simply by multiplying the inverse Fisher information by the original gradient approximation. The true fisher information can be approximated (somewhat naively) via the gradient of our score functions; this MCMC approach to approximation of the Fisher information is given by:

$$F = \frac{1}{M*T}\sum_{i=1}^{M}\sum_{t=1}^{T}\nabla_\theta \log q_\theta \nabla_\theta \log q_\theta^T \tag{15}$$

Under natural gradients we can take the gradient created through torches auto-grad function, and scale it by F^{-1} to attempted to come up with more stable gradient updates that converged more rapidly. If we assume a step-size defined as α, the gradient update would become,

$$\theta_{i+1} = \theta_i - \alpha F^{-1}\nabla\theta_i \tag{16}$$

However this algorithm did not seem to offer any noticeable improvement in performance or variance. Additionally issues with computing the inverse arise when the derivatives of any parameters are zero, thus an additional layer of handling was required in order to guarantee the system solve used to approximate the inverse was correct.

The second to last last algorithm we investigated for improving updates within max-ent policy gradients was the so called “Bayesian Policy gradient algorithm" described in [2]. In this algorithm, we approximate the gradient of the expectation described in the expression above, by assuming that the gradient is distributed as a Gaussian process. Because we can solve for the posterior of a GP analytically, we can update our posterior estimate of the gradient after every time we sample new trajectories. This ensures that the gradient can be updated, but allows previous esimates (which act

as priors) to stabalize the direction of the gradient. While this process is computationally expensive (requiring multiple system solves), we found that it was not a limiting factor in computation time (compared to the optimization problem posed by the critic). Additionally, it provides us with the inverse Fisher information, meaning that we can perform natural gradient updates for free. It also provides an avinue to create posterior covariance matrix of all the samples used in the creation of the kernal for the GP, which in theory could be used to inform the step-size taken at each iteration. Unfortunately, there was no noticeable improvement over vanilla policy gradients.

4.2 Probabilistic Program with Pyro

Levine [5] argues that performing variational inference with specific constraints enforced on the structure of the approximate distribution family is equivalent to performing maximum entropy policy gradient. Enforcing these constraints can be achieved by doing structured variational inference.

Pyro [1] is a universal probabilistic programming language written as a Python library. It includes different types of inference algorithms including sampling like HMC, and more importantly for our purposes, stochastic variational inference (SVI) with arbitrarily specified structural constraints. This means that we can implement the maximum entropy policy gradient algorithm in Pyro with minimum effort. Additionally, Pyro is tightly coupled with PyTorch [6], an open source deep learning platform with support for dynamic auto-differentiation. They provide simple wrappers around PyTorch optimizers and distributions.

Pyro supports direct optimization of the evidence lower bound (ELBO) for variational inference[4]. In general, ELBO is a function of the parameters of the model, p_θ, and the approximate distribution q_ϕ, also known as the *guide*:

$$\text{ELBO}(\theta, \phi) = \mathbb{E}_{q_\phi(\boldsymbol{z})} \left[\log p_\theta(\boldsymbol{x}, \boldsymbol{z}) - \log q_\phi(\boldsymbol{z}) \right] \tag{17}$$

Importantly, this objective is defined as an expectation with respect to the guide q_ϕ, therefore it can be approximated by sampling from this distribution. Pyro optimizes this objective directly by taking stochastic gradient updates using standard optimizers such as Adam [4].

In order to perform SVI in Pyro, one must define a pair of functions[5], one for the model and one for the guide. Conceptually, these functions provide the following:

1. A way to sample from the distribution $q_\phi(\boldsymbol{z})$;
2. A way to calculate the log densities $p_\theta(\boldsymbol{x}, \boldsymbol{z})$ and $q_\phi(\boldsymbol{z})$;
3. Specifies the set of parameters θ and ϕ used to compute these distributions;
4. (Optional) A way to calculate the gradient of the log probabilities with respect to their parameters, i.e. $\nabla_\theta \log p_\theta(\boldsymbol{x}, \boldsymbol{z})$ and $\nabla_\phi \log q_\phi(\boldsymbol{z})$.

In order to do this, the user must specify the parameters via `pyro.param` or `pyro.module` functions and describe the sample sites using `pyro.sample`. The fourth point is optional as the algorithm works without it, but it can be used to speed up the learning; more specifically, if the third option is present, the algorithm will use the pathwise derivative estimation [8], otherwise it will use the score function estimator (a.k.a. the REINFORCE estimator [10]).

In most RL contexts we are forced to use the score function estimator as we don't have direct access to a differentiable forward model of the world. In this case, we must explicitly disallow the use of the pathwise derivative estimator by using non-reparametrizable distributions. Otherwise Pyro will assume that the gradient is computable and simply uses the `backward` method from the PyTorch's auto-differentiation library to compute the gradient. This attempt will unfortunately fail silently as the computation graph does not include information about the forward model's computation. The first implementation of our code suffered from this problem and we found the bug only after closer inspection of the Pyro library.

[4] Alternative objectives are to be added later, but currently only different implementations of ELBO are supported.

[5] Any object that implements the `__call__` method.

4.2.1 SVI Implementation Details

We already discussed that Pyro directly optimizes the ELBO objective in Equation 17, but let us dive into some of the implementation details that will make the algorithm more clear.

At each step of the optimization Pyro draws a set of samples, controlled by the variable `nb_particles`, and empirically evaluates the objective as well its gradient using the techniques discussed in [8]. It then takes a single gradient step using an optimizer specified by the user to decrease the negative ELBO. It is also notable to point out that new samples need to be drawn at every single step and the previous samples are thrown away. Let us take a closer look at how this procedure is done.

First, the algorithm needs to draw samples $\boldsymbol{z}$ from the guide distribution $q_\phi(\boldsymbol{z})$. Recalling from earlier, the guide is specified as a Python function that calls the `pyro.sample` function in order to define the sample sites. A sub-library of Pyro called Poutine is responsible for running the guide and storing these samples. This is done by running the guide or the model and storing the execution trace inside a graph data structure called `Trace` which keeps all the relevant information about the sample sites, called `nodes`, and their dependency structure, stored as edges. Additionally, the `Trace` gives us access to the log probability function of each node, i.e. $\log q_\phi(\boldsymbol{z})$.

Next, we need to evaluate $\log p_\theta(\boldsymbol{x}, \boldsymbol{z})$. There are two things to note here. First, this probability is not required to be normalized since a constant term will change neither the solution nor the direction of the gradient. Second, we have already sampled $\boldsymbol{z}$ and we need to make sure to use the same sample rather than drawing a new one. The Poutine library provides the latter functionality as it lets us `replay` the model but instead of sampling, replaces the sampled values that were taken from running the guide program earlier. The guide is not required to provide values for all the sample sites in the model, rather if a site name is not present in the guide's `Trace`, it will automatically be sampled when running the model program forward. A crucial point to get right here is that any sort of non-determinism in the course of running the model and the guide programs should only be introduced through `pyro.sample` statements. Otherwise, the model program might go down a different path than the guide which can have catastrophic effects such as getting a probability of zero (log probability of negative infinity) for one of the samples.

Using the samples drawn from the guide and the log probabilities stored in the `Trace` data structures for both the model and the guide, the objective can be evaluated and with the help of automatic differentiation its gradient can be computed.

It is also important to note that Pyro supports multiple types of baselines in order to decrease the variance of the score function estimator discussed earlier. Schulman et al. [8] argues that this estimator can be modified by adding a baseline in order to get a lower variance estimator:

$$\frac{\partial}{\partial\theta}\mathbb{E}_{v\sim p(\cdot;\theta)}\left[f(v)\right] = \mathbb{E}_{v\sim p(\cdot;\theta)}\left[\frac{\partial}{\partial\theta}\log p(v;\theta)f(v)\right] = \mathbb{E}_{v\sim p(\cdot;\theta)}\left[\frac{\partial}{\partial\theta}\log p(v;\theta)(f(v)-b)\right]$$

and the above formula stays correct even if b is a function of anything that is not influenced by the variable v. In Pyro, the user can seperately assign baseline values in one of the following ways[6]:

1. Specifying a constant value;
2. Using a running mean of the previous values seen by using the decaying average option;
3. Specifying a neural network that depends on the previously sampled nodes, ensuring that they are not influenced by the current variable.

4.2.2 RL as SSVI in Pyro

Now, we can define a RL problem in Pyro and solve it with structured stochastic variational inference:

```
def model(env, _):
    s = pyro.sample("s_1", env.InitDistribution)
    for i in range(1, env.T):
```

[6]Currently, only the `TraceGraph_ELBO` objective supports this argument and not the `Trace_ELBO` objective.

```
        a  = pyro.sample("a_%d" % i, Uniform)
        sp = pyro.sample("s_%d" % i+1, env.Transition(s, a))
        r  = env.Reward(s, a, sp)
        O_dist = pyro.dist.Bernoulli(exp(r))
        pyro.sample("O_%d" % i, O_dist, obs=1)
        s = sp

def guide(env, policy):
    pyro.module("policy", policy)
    s = pyro.sample("s_1", env.InitDistribution)
    for i in range(1, env.T):
        a  = pyro.sample("a_%d" % i, policy(s))
        sp = pyro.sample("s_%d" % i+1, env.Transition(s, a))
        s  = sp
```

We will go over the model program first. It takes the environment as input and ignores the second argument which is the policy here. First, it samples s_1 and then takes the first action a_1. This is assumed to be a uniform distribution, since without any loss of generality, any prior on the action can be added to the reward function instead. After that the next state is sampled from the transition probability. Next, the environment calculates the reward and we define the optimality variables O_t as before. The only thing that is left is to *observe* the optimality variables to be always equal to 1. There are two ways to do this, but the simplest approach is just to add an `obs` variable to the `sample` function.

A keen observer might notice that the optimality variables in our model are dependent on the next state as well as the previous state and action. This slight deviation from the original graphical model will not adversely affect the derivations, however it will make the framework more general as the reward function in many cases depends on the next state as well.

The structure of the guide program is really similar to the model program with some differences. Most importantly, the action distribution is chosen according to the policy rather than a uniform distribution, but the distribution of the states is unchanged. This way we are enforcing that the guide program has authority over choosing the actions, but it cannot influence the distribution of the states themselves. Next, we need to inform Pyro that the parameters of the `policy` can be optimized. This can be via the `pyro.param` or `pyro.module` directives. Additionally, the optimality variables are removed as well.

4.2.3 Generalizing the Original Framework

The framework that we talked about in the previous section closely followed Levine [5], however in practice this framework proved to be limiting. This section will discuss some of the improvements as well as practical concerns when implementing RL in Pyro. We will also discuss some of the limitations that we could not address which can be explored in the future work.

One of the limiting assumptions in the original framework was that the rewards should always be non-positive. This assumption can be relaxed and our experiments show that the relaxation does not alter the results. We first need to notice that Pyro does not require sampling from the `model` program as we discussed before and only the log density is required. Its also easy to see that using an un-normalized distribution in the place of p_θ will not change the gradient of the objective and therefore is allowed. This means that we can define $p(O_t = 1) \propto \exp(r_t)$ instead. This way we can use positive rewards. As an extra implementational detail, there's no need to exponentiate the reward since Pyro only requires the log probability which is equal to $\log p(O_t = 1) \propto \log(\exp(r_t)) = r_t$. Based on the same principle, we can implement an `InfiniteUniform` distribution for the actions that will simply assign an un-normalized log probability of zero to all possible actions. In theory, one might want to be more careful as to define finite support for the prior over actions, but in practice using such a distribution will make the implementation more smooth and robust to different choices of policy distributions.

Next, the original framework requires a fixed episode length since it is based on a graphical model. However, Pyro is a higher order probabilistic programming language (HOPPL), therefore it would be trivial to get around this limitation. Assuming the environment emits a *done* signal whenever the episode finishes, it would be straightforward to exit the model and guide programs based on this

signal with a simple condition. One must be careful as to do this in both the model and the guide programs, otherwise the execution traces will not match.

Finally, we should point out that the model and the guide programs that are defined here assume having access to both the initial and transition distributions through `env.InitDistribution` and `env.Transition`. This is not true in practice, but we cannot simply remove them as Pyro should be the only source of randomness in these programs. Overlooking this point can result in a catastrophic failure. Fortunately, the standardized Gym [7] environments expose a `seed` method which can control the randomness of the environment, which we can use. Though this might not be a satisfying solution as it changes the structure of the graphical model, it takes care of the problem and lets us use this approach on any environments that correctly supports this interface.

As for the limitations of this framework, we should mention that even though [5] proposes a way to incorporate a *discount factor* into the graphical model, this approach is not practical as it will increase the variance of the algorithm which is not desirable. Unfortunately, we were not able to find a way around this problem.

The next problem that we faced is that even though we can use critics in the form of a baseline, we were not able to incorporate more sophisticated ways of using the critic, such as generalized advantage estimation [9], into our framework.

4.2.4 SVI Objectives in Pyro

SVI supports two types of objective for ELBO: *TraceELBO* and *TraceGraph ELBO*. As the names suggest they are both optimizing the ELBO loss or more accurately minimizing the negative ELBO but they also have some major differences. TraceELBO does not support baseline for variance reduction. Therefore, it minimizes the objective (10) with $b(s_{t'}) = 1$. On the other hand, TraceGraph ELBO supports three types of baseline: baseline value, decaying average and neural baseline. As discussed earlier in section 4.2.1, baseline value is simply a constant that is specified by the user. Decaying average is the running average of the recent samples and is influenced by a decay rate. The last baseline is specified by the parameters of a neural network which can be adapted during the learning process. The other major difference between these two objectives is in the way they construct the surrogate loss for gradient estimation. TraceELBO only supports reparameterizable random variables (reparameterization trick) for estimating the gradient:

$$\mathbb{E}_{q_\phi(z)}[f_\phi(z)] = \mathbb{E}_{q(\epsilon)}[f_\phi(g_\phi(\epsilon))] \rightarrow \nabla_\phi \mathbb{E}_{q(\epsilon)}[f_\phi(g_\phi(\epsilon))] = \mathbb{E}_{q(\epsilon)}[\nabla_\phi f_\phi(g_\phi(\epsilon))]$$

It also uses a specific dependency structure that cannot be easily adapted. In other words, TraceELBO assumes that all random variables specified in the model/guide are dependent. However, TraceGraph ELBO does not have this assumption. In other words, it keeps track of the dependency structure within execution traces of model and guide and constructs a surrogate objective that has all the unnecessary terms removed. It uses score function method to take care of the non-reparameterizable random variables:

$$\begin{aligned}\mathbb{E}_{q_\phi(z)}[f_\phi(z)] &= \mathbb{E}_{q_\phi(z)}\big[(\nabla_\phi \log q_\phi(z)) f_\phi(z) + \nabla_\phi f_\phi(z)\big] \\ &\rightarrow objective \equiv \log q_\phi(z)\overline{f_\phi(z)} + f_\phi(z),\end{aligned}$$

where $\overline{f_\phi(z)}$ should be treated as a scalar and not back-propagated through. As TraceGraph ELBO considers dependency structure and uses score function for gradient estimators, it has high variance and is much slower compared to TraceELBO.

4.3 Proximal Policy Optimization as a Custom Loss Function in Pyro

The methods discussed so far are based on vanilla policy gradient optimization. Introducing baselines to this objective helps a lot but it still has many problems. Vanilla policy gradient suffers form two major limitations: It's hard to choose the right step size and it is not sample efficient. The input data is non-stationary due to the changing policy and reward distributions which makes the optimization much harder. A bad step size results in a bad policy which can further result in performance collapse as the data is sampled under a bad policy and a wrong distribution. Moreover, taking only one gradient step per a batch of sample trajectories is not efficient and makes it harder to converge.

In order to solve these issues, PPO or in general trust region methods propose a new objective function from which we are able to optimize the policy multiple times and the gradient estimate of the objective stays the same. This new objective can be written as:

$$\mathbb{E}_{(s_t,a_t)\sim q_\theta(s_t,a_t)}\left[\frac{q_\theta(a_t|s_t)}{q_{\theta_{old}}(a_t|s_t)}\hat{A}(s_t,a_t)\right]. \tag{18}$$

Now, we are able to optimize the objective over θ w.r.t. θ_{old} and the nice thing is that the gradient of this objective w.r.t. θ is the same as the gradient of the original objective.

$$\mathbb{E}_{(s_t,a_t)\sim q_\theta(a_t,s_t)}\left[\nabla_\theta \log q_\theta(a_t|s_t)\hat{A}(s_t,a_t)\right] = \mathbb{E}_{(s_t,a_t)\sim q_\theta(s_t,a_t)}\left[\frac{\nabla_\theta q_\theta(a_t|s_t)}{q_{\theta_{old}}(a_t|s_t)}\hat{A}(s_t,a_t)\right]. \tag{19}$$

Although this objective lets us do multiple steps of optimization, it doesn't fully solve the problem as the updates to the policy should be rather small. This can be enforced by adding a constraint on the parameters of new policy and the old policy such that the KL divergence between these two is smaller than some threshold. There are a lot of different methods to apply this. Two of the common approaches which are generally used by PPO are using an adaptive KL penalty and clipping the gradients to avoid policies that are very different from the previous policy.

$$J(\theta) = \sum_{t=1}^{T}\mathbb{E}_{(s_t,a_t)\sim q(s_t,a_t)}\left[\min\left(\frac{q_\theta(a_t|s_t)}{q_{\theta_{old}}(a_t|s_t)}\hat{A}(s_t,a_t), clip(1-\epsilon,1+\epsilon,\frac{q_\theta(a_t|s_t)}{q_{\theta_{old}}(a_t|s_t)})\hat{A}(s_t,a_t)\right)\right]. \tag{20}$$

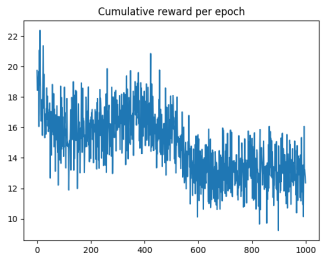
This objective can easily be incorporated in Pyro as a custom loss. As discussed in section 4.2.4, TraceELBO only supports parameterizable variables so it is not suited for our objective. On the other hand, TraceGraph ELBO uses score function gradient estimators but does not have access to the optimizer so that we can do multiple steps of optimization on finding a new policy. In addition, the structure for obtaining the loss is a bit different from what is needed for PPO. Therefore, it was necessary to develop a custom loss function. As we developed a new objective, we had to take care of getting the gradients, back-propagating and doing the optimization steps. Pyro uses a nice data structure (Trace) which keeps track of the execution trace of the program and samples from all random variables of the guide and the model. It has access to name, type, log probability and the joint probabilities of each random variable and the value of each parameter. Therefore, we were able to implement PPO's loss in Pyro and do inference on it.

5 Results and Comparison

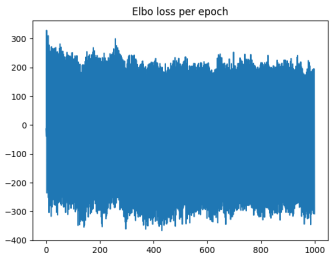
As can be seen from the experiments with a direct implementation of maximum entropy policy gradients when a constant variance is assumed for the distribution, the gradient updates for the policy are well behaved, and the cumulative reward trains more smoothly. While in the case where the standard deviation is allowed to vary, the gradients tend to become extremely unstable. Additionally, there is a trade off as the algorithm struggles to minimize its objective, when it can only change the mean of the distribution, it focuses on specifically minimizing the cost.

5.1 Results with Pyro

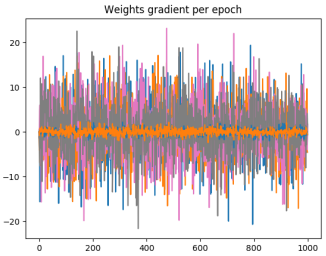
The first thing that we discovered is that even though we started with a simple enough example, the algorithm did not work out of the box and the ELBO objective was highly noisy throughout the experiment. We first tried exploring different hyper-parameter settings, but as there was little to go on from the unchanging objectives, we could not find the right hyper-parameters. Therefore we decided to start with the simplest possible examples that we could think of: a deterministic 2D environment where $T = 1$ and the reward is simply the mean-squared-error between the action and the correct direction. This experiment can also be seen as a simple example of supervised learning with just one data-point. The algorithm was able to solve this problem, but sadly, it took around 20 thousand samples to achieve reasonable results.



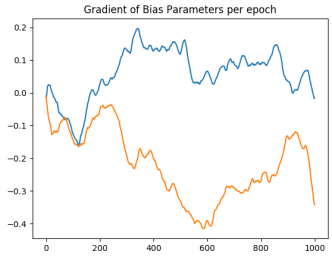
(a) Cumulative Reward Per Epoch



(b) Negative ELBO Loss Per Epoch

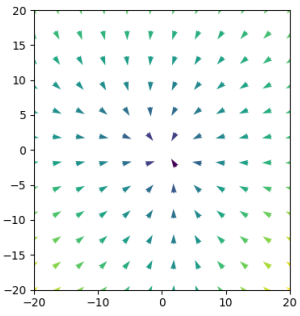


(c) Gradient of Weight Parameters Per Epoch

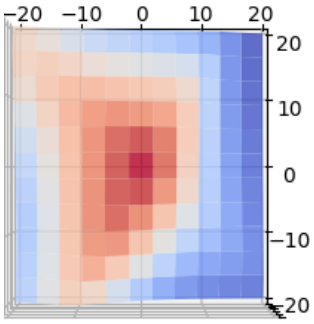


(d) Gradient of Bias Parameters Per Epoch

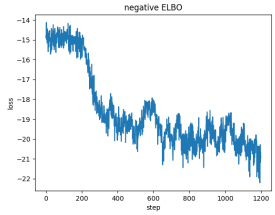
Figure 2: Maximum entropy Policy gradients with Critic applied to parameterization of multivariate normal distribution; in this case, variance is held constant.



(a) Trained Policy



(b) Trained value function

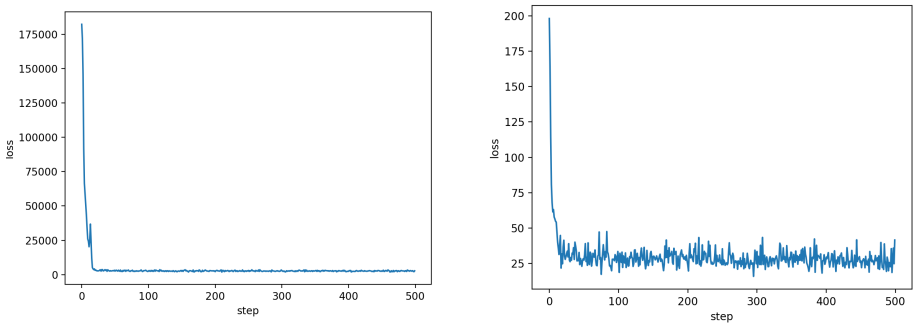


(c) Negative ELBO calculated at each step of training

Figure 3: The results of training the 2D Point-Mass environment with SVI in Pyro.

Next, we decided to make the problem more difficult in an incremental fashion until the actual problem was solved. The details of the experiments are not important, but doing this gave us a principled approach to finding good hyper-parameters and at the same time find or debug the parts that did not work well. Just to demonstrate its difficulty, we should note that we experimented with ten levels of difficulty each with multiple variations in the hyper-parameters and approaches.

Figure 3 shows the results of the final solution that we achieved. This example used 500 particles with a fixed episode length of 10 per each update which amounts to around 3 to 6 million time-steps in total. As stated in section 4.3, we also developed the PPO loss function for our problem. However, the loss function did not converge to a good solution. One of the reasons may be the fact that we are optimizing a very simplified version of PPO without using the generalized advantage estimators. Other reasons that affect the performance are hyper parameters values and the number of total samples to converge. As shown in figure 4, the loss function has a high decrease in the first few steps but then stays the same.



(a) number of particles = 100.

(b) number of particles = 10.

Figure 4: The results of using PPO loss with SVI in Pyro. Both of the figures are using 10 gradient updates per step.

6 Conclusions and Future Work

This paper discusses how probabilistic inference can be applied practically to reinforcement learning problems. By describing how a Markov decision process can be re-formulated as a graphical model, we can then search for an optimal policy via structured variational inference following [5]. Noting that this problem solves exactly the same objective as maximum entropy policy gradients, we analyze this methodology and determine what variations are required to ensure that it functions correctly. Specifically, we review the necessity for a critic to stabilize variance of the gradient, and show how to include a discount factor in the policy gradient objective. To conclude our discussion of policy gradients, we briefly review other possible ways to augment the gradient update to improve convergence, and stability. With a practical perspective of how optimization will occur within our objective, and a framework to perform variational inference, we then derive RL in the context of a probabilistic programming language. As a probabilistic program, we review how what we learned from an analysis of policy gradients, can be applied to improve performance of the variational inference procedure utilized in Pyro. We conclude with an explanation of how this performance could be further improved by the incorporation of Proximal Policy Optimization as a custom loss function.

The final results section highlights the core lesson of this topic; which is that while Pyro displays an ease of use at a superficial level, implementation of even simple RL problems requires a great deal of hyper parameter tuning. Additionally, these implementations are computationally costly when compared to practical methods. While improvements can be made to better adapt probabilistic programming to RL, there are still many questions that have to be addressed. Chief amongst these issues being proper integration of a discount factor into the framework to further reduce variance in the gradient estimates for long trajectories. Even with all of these drawbacks, this paper shows that it is possible to utilize PPL in reinforcement learning and in future work under improved implementations, this concept could lead to more practical usage. Additionally, this framework allows for a much more direct injection of expert knowledge into RL model frameworks. An interesting extension might be using hierarchical Bayesian models to perform more difficult tasks under expert prior assumptions, or perhaps to use expert examples to improve sampling from the exact distribution. Additionally further exploring how trust region methods native to classical RL might lead to the necessary leaps in performance that could make this methodology tractable. Finally, is it possible to perform amortized inference on more difficult RL problems to find new ways to distill information. While, there were minor setbacks, the future for PPL in reinforcement learning looks bright and although it does not necessarily set any benchmarks, it does improve our tools for reasoning about established methods, and opens possible avenues for new implementations in the future.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[2] Mohammad Ghavamzadeh, Shie Mannor, Joelle Pineau, Aviv Tamar, et al. Bayesian reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 8(5-6):359–483, 2015.

[3] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[5] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint arXiv:1805.00909*, 2018.

[6] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[7] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research, 2018.

[8] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *CoRR*, abs/1506.05254, 2015.

[9] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

[10] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.

[11] Brian D Ziebart. Modeling purposeful adaptive behavior with the principle of maximum causal entropy. 2010.