中国科学: 信息科学 2022 年 第 52 卷 第 5 期: 804–821
SCIENTIA SINICA Informationis
论文

珠算: 可微概率编程库的设计与实现

石佳欣[1,2], 陈键飞[1], 朱军[1*]

1. 清华大学计算机科学与技术系, 北京信息科学与技术国家研究中心, 清华 – 博世机器学习联合中心, 清华大学人工智能研究院, 智能技术与系统国家重点实验室, 北京 100084, 中国
2. Microsoft Research New England, Cambridge MA 02142, USA
* 通信作者. E-mail: dcszj@tsinghua.edu.cn

收稿日期: 2021–01–08; 修回日期: 2021–04–30; 接受日期: 2021–06–08; 网络出版日期: 2022–05–12

国家自然科学基金 (批准号: 61620106010, U19B2034, U181146)、科技部重点研发计划 (批准号: 2017YFA0700900)、北京市自然科学基金 (批准号: JQ19016)、清华国强研究院、清华 – 华为大颗粒合作、北京智源人工智能研究院、鹏城实验室重大攻关项目 (批准号: PCL2021A12) 和人工智能与数字经济广东省实验室 (广州) 等项目资助

摘要 概率模型为机器学习处理广泛存在的不确定性提供了强大的工具. 概率编程利用计算机程序表示概率模型, 支持采样和以任意观察值为条件进行的概率推断. 长期以来, 概率程序中的依赖关系往往是线性或广义线性的, 许多成功的模型和推断算法往往都依赖于这一简化. 然而, 这也限制了概率程序的表达能力和灵活性. 可微概率编程允许构建具有参数化的非线性依赖关系 (如神经网络) 的概率程序, 并使用基于梯度的方法从数据中学习未知参数. 这种编程范式容易扩展, 极大地避免了繁琐的模型选择过程, 且允许端到端地部署概率模型. 本文介绍珠算 (ZhuSuan), 一种开源的可微概率编程库, 并以此为例, 探讨可微概率编程系统的设计与实现.

关键词 概率模型, 概率编程, 贝叶斯推断, 变分推断, 深度学习

1 引言

以深度神经网络为代表的深度学习 [1] 技术在许多应用中取得了巨大成功, 包括语音识别 [2]、计算机视觉 [3]、自然语言处理 [4] 和计算机游戏 [5], 但也存在诸多挑战. 例如, 单一神经网络通常缺乏对不确定性的合理建模, 往往给出 “过于自信” 的预测结果, “不知道自己不知道” —— 假设给定一个在动物图像数据集上训练好的分类器, 当输入一个苹果的图像时, 该分类器会 “自信” 地预测为某一种动物类别; 同时, 在存在对抗噪声时, 可能被欺骗 [6]; 此外, 神经网络的输出是由大量权值组合得到, 缺乏可解释性. 这些挑战会阻碍深度学习在交通、医疗、金融等关键领域中的应用.

概率机器学习为不确定性建模与推断提供了强大的工具 [7,8], 通过构建合理的概率模型, 可以对不确定性进行刻画 [9], 拒绝/修正不确定性高的样本 [10], 提高数据的利用效率进行小样本学习 [11] 或

引用格式: 石佳欣, 陈键飞, 朱军. 珠算: 可微概率编程库的设计与实现. 中国科学: 信息科学, 2022, 52: 804–821, doi: 10.1360/SSI-2021-0005
Shi J X, Chen J F, Zhu J. ZhuSuan: design and implementation of differentiable probabilistic programming libraries (in Chinese). Sci Sin Inform, 2022, 52: 804–821, doi: 10.1360/SSI-2021-0005

半监督学习 [12] 等. 长期以来, 概率机器学习的进展主要是由概率图模型 [13] 的语言驱动的, 它以图的方式表示多个变量的联合分布和其中的条件依赖关系, 具有简洁直观、可解释性强等特点, 同时也具有通用的参数学习和概率推断算法. 构建在这一语言之上的是一种基于模型的思考方式, 目标是理解真实世界中的现象, 手段是设计与之相对应的模型, 并通过观测数据反推整个系统的运行方式和未知细节.

使用概率模型的主要挑战之一是后验推断, 对于绝大多数模型此类推断都是极难处理的, 因而需要复杂的近似技术. 虽然变分推断 [14] 和马尔可夫链蒙特卡洛 (Markov Chain Monte Carlo, MCMC) 方法 [15] 经过了多年发展, 但对于非专业的从业者来说, 在复杂模型中使用它们仍然是棘手的任务. 而且, 尽管推断方法有它们的一般教程, 但对于特定模型的每个细节, 即使是经验丰富的研究人员实现它们也需要花费大量时间和精力. 这样的过程容易出错, 且调试时间较长. 相比之下, 一个自动的推断系统将有利于更快速地建立原型和在数据上测试不同的模型, 这可能会潜在地加速科学进程.

概率编程 (probabilistic programming) 是一个旨在使用计算机程序表示概率模型的研究方向 [16,17]. 概率编程将概率图模型扩展为更丰富的表示, 以囊括编程语言中的原语. 编写概率程序的过程十分简单, 只需将数据的生成过程用程序语言描述, 这样的程序也被称为模拟器 (simulator). 例如, LDA (latent Dirichlet allocation) [18] 是一个经典的概率主题模型, 它将文档的编写描述为: 首先从主题的概率分布中采样该文档的主题, 然后根据该主题, 从主题决定的词汇分布中独立采样每一个词语. 构建这样的模拟器的目标是近似真实的数据生成过程. 概率编程则利用用户编写的模拟器, 企图实现其逆过程, 即从真实数据反推其中的参数和未观察到的随机变量 (即隐变量) 的分布. 这与概率模型的推断和学习过程相对应.

概率编程的核心哲学是将构建模拟器和推断的过程分离 [19]. 更确切地说, 用户可以使用提供的建模语言构建任意的概率模型, 而对应的推断程序将由概率编程系统自动执行. 这样的系统将鼓励基于模型的思考, 并且无需用户手动推导模型的推断方法, 因而会对科学建模产生重大影响 [7].

传统的概率图模型中最常见的条件依赖关系是线性的. 例如, 一大类层次概率图模型 (协同过滤、线性动态系统、稀疏编码等) 中的条件依赖关系均可以通过矩阵乘积的形式表达 [20]. 线性关系也使得共轭先验 (conjugate prior) 的使用成为可能, 大多数得到广泛运用的概率图模型均不同程度地依赖于共轭结构给推断算法带来的简化 [18,21]. 然而, 由于真实世界中普遍存在的数据 (如自然图像) 通常具有高维结构, 简单的条件依赖关系很可能不足以刻画它们 [22]. 另一方面, 近年来一种被称作可微编程的机器学习范式取得了重大进展, 这一范式是指编写具有许多参数的模型组成的程序以解决特定的问题, 通常通过基于梯度的优化使损失函数最小化来达到目的. 其中最广为人知的代表即是深度学习. 深度学习的成功启示我们: 通过使用基于梯度的优化算法 (例如随机梯度下降), 大规模的训练集 (例如 ImageNet) 和功能强大的计算设备 (如 GPU), 一个复杂的深层非线性模型可以得到很好的训练.

受此启发, 在概率模型中引入可训练的非线性条件依赖关系受到了越来越多的关注. 深度生成模型的成功 [23,24] 表明, 通过使用神经网络在概率图模型中建模条件依赖关系, 我们可以有效地将端到端学习和结构化知识相结合, 从数据中获得可解释的表示. 这样的想法在结构化数据生成 [25,26]、半监督分类 [12,27,28]、异常检测 [29]、小样本学习 [30] 等任务中产生了卓越的成果. 将神经网络与概率模型融合的另一个方向是对神经网络模型进行贝叶斯 (Bayesian) 推断, 已有研究表明, 这是一种有效提升神经网络泛化性的手段 [31,32], 同时可以合理刻画不确定性 [9,33].

但是, 将神经网络与概率模型相结合也给概率推断带来了新挑战, 为此, 基于变分推断和蒙特卡洛模拟的多种算法被提出 [33,34]. 与深度神经网络相比, 融入神经网络的概率模型在概念上更难理解, 在算法实现上更加复杂, 因此, 研制功能完善, 易用的概率编程库成为一个重要问题. 为此, 本文介绍一个

开源概率编程库 “珠算” (ZhuSuan)[1)]的设计以及实现原理. 本文将支持此类建模方式的概率编程称为可微概率编程. 它的一个独特特征是随机变量的条件依赖关系可以通过具有强大表达力的非线性模型(例如深度神经网络[35]) 以基于梯度的方式从数据中自动学习, 而在传统的概率编程语言中, 这些关系往往具有简单的形式. “珠算” 是基于 Tensorflow[36] 的可微概率编程库, 并与华为公司的 MindSpore 以及百度公司的 PaddlePaddle 等国产深度学习框架进行融合, 推动概率编程平台的国产化.

2 已有的概率编程系统回顾

传统的概率编程系统可以分为两大类. 诸如 BUGS[37], Stan[38], 以及 Infer.NET[39] 这一类系统专门针对特定的推断算法, 而其他系统则更多地关注通用的图灵完备 (Turing complete) 语言. 它们包括 BLOG[40], Church[41], Venture[42], Anglican[43], WebPPL[44] 等.

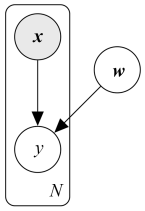
BUGS[37] 是早期的概率编程系统, 其主要针对层次贝叶斯模型应用吉布斯采样 (Gibbs sampling) 进行推断. 用户可以使用其提供的建模语言描述一个有向图模型. Stan[38] 与 BUGS 具有类似的界面, 它们都提供了自己的建模语言. 但 Stan 更侧重于依赖哈密顿蒙特卡洛 (Hamiltonian Monte Carlo, HMC) 算法, 因而两者的适用范围有较大区别. Stan 主要适用于联合概率可微的模型, 这限制了离散随机变量只能在部分容易处理的场景 (如可遍历的情况) 使用. Stan 支持非常有限的变分推断算法, 对连续隐变量均使用高斯 (Gauss) 的变分分布 (若定义域不同则进行自动变换) 以及重参数化的梯度估计器[45]. Infer.NET[39] 是基于.NET 语言, 主要针对概率图模型中基于消息传播的推断算法, 包括变分消息传播 (variational message passing, VMP)[46] 以及期望传播 (expectation propagation)[47]. 这些算法大部分使用较强的假设, 如 VMP 依赖于平均场假设, 且需要近似处理模型中的非共轭结构.

Church[41] 构建在 Scheme 语言的一个子集上, 以图灵完备、支持高阶程序为特征. 这一类型的语言被称为通用 (universal) 概率编程语言. 后来许多概率编程系统 (如 Venture, Anglican, WebPPL, Pyro 等) 的设计均采用了类似的思路. 在这类系统中模型语言强大的描述能力导致实现推断算法的复杂度急剧增加. Church 的原始实现中采用了较为通用的采样算法如拒绝采样 (rejection sampling), 然而算法的效率却没有保证, 难以应用到真实的机器学习问题中. Venture[42] 保持了语言的通用性, 重点提升了推断系统的灵活性, 使得多种算法可以灵活组合. 例如, 其中允许为蒙特卡洛算法编写定制的建议分布. 在变分推断算法方面, Venture 支持了基于 REINFORCE 梯度估计器[48, 49] 的黑盒算法. Anglican[43] 扩展了 Clojure 语言. 其与前面两个系统不同的地方是引入了基于粒子 (particle-based) 的推断算法, 包括序列蒙特卡洛 (sequential Monte Carlo) 和粒子马尔可夫链蒙特卡洛 (particle MCMC). 此类算法利用了程序本身的结构, 因而提升了通用概率编程系统的推断效率. WebPPL[44] 吸收了上述系统的优点, 在 JavaScript 语言之上进行了轻量级的实现, 使得概率编程系统可以在浏览器中运行.

通用概率编程系统的思想与可微编程紧密相关, 因为后者往往是由一些支持自动微分的通用计算框架实现的, 如 Theano[50], Tensorflow[36] 和 PyTorch[51]. 近年来, 已经有若干融合概率编程和自动微分 (automatic differentiation, autodiff) 框架的工作出现. 例如, PyMC3[52] 和 Edward[53] 是分别基于 Theano 和 Tensorflow 建立的此类系统. 但是, 由于随机变量的不确定性, 这些框架很难做到在观测状态发生改变后既能重用已有模型, 同时又不丢失自动微分所需要的依赖信息. 因此, 它们在传统的层次模型之外的支持非常有限, 并且某些功能 (例如 Edward 中的控制流支持) 不完整.

我们注意到在 ZhuSuan[54] 之后有基于 PyTorch[51] 的概率编程库 Pyro[55] 发布. 由于 PyTorch 引入了支持动态控制流的自动微分系统, 这使得 Pyro 可以借鉴 WebPPL[44] 等通用概率编程语言的

1) https://github.com/thu-ml/zhusuan.



(a)

```python
import zhusuan as zs
import tensorflow as tf

@zs.meta_bayesian_net(scope="blr")
def build_blr(x, alpha, D):
    bn = zs.BayesianNet()
    w = bn.normal('w', mean=tf.zeros([D]), std=alpha, group_ndims=1)
    y_logit = tf.reduce_sum(tf.expand_dims(w, 0)*x, axis=1)
    bn.bernoulli('y', y_logit)
    return bn
```

(b)

图 1 (网络版彩图) BLR. (a) 概率图模型; (b) 概率程序

Figure 1 (Color online) BLR. (a) The probabilistic graphical model; (b) the probabilistic program

设计: 直接使用编程语言原生的控制流操作 (包括 `for`, `if`, `while` 以及原生递归). 此外, 另一个基于 PyTorch 的概率编程库 ProbTorch [56] 主要限于深度生成模型和部分变分推断算法.

3 珠算可微概率编程库的接口设计

本节介绍 "珠算" (ZhuSuan) 可微概率编程系统的接口设计. 该框架基于 Tensorflow 构建, 以利用其可微分的计算框架来实现灵活的建模.

3.1 用于建模的原语

通常, 由于概率程序定义的是模型而不是过程 [41], 其原语是借鉴概率图模型设计的. ZhuSuan 支持的原语主要是用于描述有向图模型, 即贝叶斯网络的结构. 与许多现有语言的风格不同, 该框架在定义模型时避免显式的采样和观测行为. 更确切地说, ZhuSuan 中的模型定义代码可以没有显式的 `sample` (采样) 或 `observe` (观测) 语句 [17]. 相反, 该框架通过懒惰的执行机制 (将在第 4 节详述) 鼓励 "模型即程序", 使得模型定义像相应的图模型一样直观, 如下面例子所示.

例1 (贝叶斯 Logistic 回归) 贝叶斯 Logistic 回归 (Bayesian logistic regression, BLR) 的生成过程写作

$$\begin{aligned} &\boldsymbol{w} \sim N(\mathbf{0}, \alpha^2 \boldsymbol{I}), \\ &y_i \sim \text{Bernoulli}(\sigma(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_i)), \quad i = 1, \ldots, N, \end{aligned} \tag{1}$$

其中 $\boldsymbol{x}_i \in \mathbb{R}^D$ 表示输入向量, $\boldsymbol{w} \in \mathbb{R}^D$ 表示参数向量, $y_i \in \{0, 1\}$ 表示类别标签, $\sigma(\cdot)$ 是 sigmoid 函数. 图 1 中显示了图模型和对应的概率程序. 注意程序中输入数据点是批量处理的.

在此示例中, 确定性变换部分是线性模型 ($\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$), 由 Tensorflow 操作 `tf.expand_dims`, `tf.reduce_sum`, `tf.multiply(*)` 实现输入数据 ($\boldsymbol{x}_i, i = 1, \ldots, N$) 的批量处理. 两个随机变量 y 和 $\boldsymbol{w}$ 由 `bn.bernoulli` 和 `bn.normal` 创建, 分别表示它们由伯努利 (Bernoulli) 和正态分布定义. `group_ndims` 参数表示 $\boldsymbol{w}$ 的最后一个维度被视为一个随机变量, 其概率值是一起计算的.

注意模型定义是由 `zs.meta_bayesian_net` 装饰的函数. 接下来将看到, 这正是在建模时避免显式采样和观测的方法.

模型类. 从概念上讲, 每个 `BayesianNet` 实例都对应一个概率程序的执行轨迹 (execution trace) [19], 其中所有随机变量的状态已经确定, 即我们知道对哪个变量进行了采样, 以及哪些变量已经被观测到了. 可以将观测值作为参数传递给 `BayesianNet` 对象的构造函数:

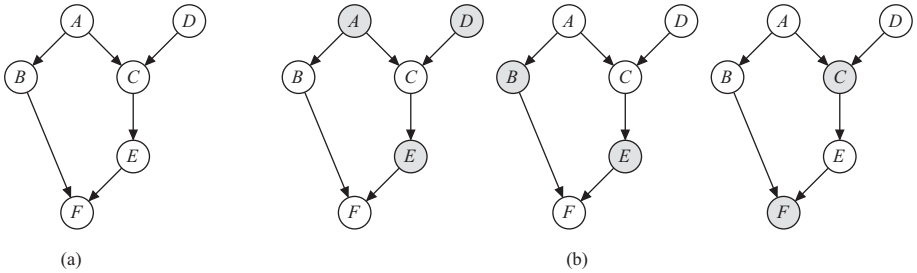


图 2　模型类和模型重用

Figure 2　Model classes and model reuse. (a) MetaBayesianNet; (b) BayesianNet

```
bn = zs.BayesianNet(observed={'w':  w_obs})
```

例如, 考虑将图 1 中的代码去掉装饰器 `zs.meta_bayesian_net`, 调用该函数将返回一个 `BayesianNet` 对象:

```
>>> bn = build_blr(x, alpha, D)
>>> print(bn)
<zhusuan.framework.bn.BayesianNet object at ...
```

返回的 `BayesianNet` 实例会将所有随机变量的值设置为他们对应分布的样本, 因为在构建 `BayesianNet` 对象时没有传入任何观测值. 然而, 由于使用 `zs.meta_bayesian_net` 装饰器, 情况发生了变化. 由 `zs.meta_bayesian_net` 装饰的函数将返回一个 `MetaBayesianNet` 实例而不是 `BayesianNet` 实例:

```
>>> meta_bn = build_blr(x, alpha, D)
>>> print(meta_bn)
<zhusuan.framework.meta_bn.MetaBayesianNet object at ...
```

直观地讲, 一个 `MetaBayesianNet` 实例在概念上与一个概率图模型是等效的. 其中的所有随机变量具有不确定的状态, 这意味着我们既可以从中采样也可以指定观测值. 两个概念之间的关系如图 2 所示.

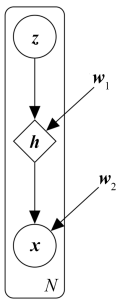
模型重用. 如前所述, `MetaBayesianNet` 可以允许不同的观测值配置. 这是通过其 `observe` 方法实现的. 我们可以将观测值作为命名参数传入该方法, 它将返回对应的 `BayesianNet` 实例, 例如:

```
>>> bn = meta_bn.observe(w=w_obs)
>>> print(bn)
<zhusuan.framework.bn.BayesianNet object at ...
```

这使得用户更容易处理大量随机节点的状态变化. 更重要的是, 我们将在 3.2 小节中看到, 这样的模型抽象形式给自动推断系统提供了统一的交互界面.

可以看到, 以上的概率程序由 Tensorflow 的原语和 `BayesianNet` 的成员方法组合得到. ZhuSuan 支持以下用于在贝叶斯网络中构造节点的原语.

确定性节点. 用户可以使用任意的 Tensorflow 操作 (operation) 定义确定性节点. 这包括各种算术运算符 (例如 `tf.add`, `tf.tensordot`, `tf.matrix_inverse`)、神经网络层 (如 `tf.layers.fully_connected`, `tf.layers.conv2d`), 以及程序控制流 (例如 `tf.while_loop`, `tf.cond`). 在 Tensorflow 计算图中, 操作的输出被命名为张量 (tensor). 与 PyMC3 [52] 不同, ZhuSuan 未在张量上添加更高级别的抽象, 而是直接将它们视为贝叶斯网络中的确定性节点. 我们将看到其他原语可以直接与张量一起很好的工作.



(a)

```python
@zs.meta_bayesian_net("vae", reuse_variables=True)
def build_vae(N, D):
    bn = zs.BayesianNet()
    z_mean = tf.zeros([N, D])
    z = bn.normal('z', z_mean, std=1., group_ndims=1)
    h = tf.layers.dense(z, 500, activation=tf.nn.relu)
    x_logits = tf.layers.dense(h, 784)
    bn.bernoulli('x', x_logits, group_ndims=1)
    return bn
```

(b)

图 3 (网络版彩图) VAE. (a) 概率图模型; (b) 概率程序

Figure 3 (Color online) VAE. (a) The probabilistic graphical model; (b) the probabilistic program

随机节点. 随机节点是通过调用 `BayesianNet` 实例的成员函数来构造的. 这意味着在构造随机节点时, 它的状态已根据该 `BayesianNet` 对象相应的执行轨迹确定. 返回的节点是一个 `StochasticTensor` 对象, 该对象继承了大多数张量的行为. 它们可以直接被送入 Tensorflow 操作. 在与张量进行计算时, 这些对象可以根据他们在目前的执行轨迹中的取值 (样本或观测值) 自动转换为张量. 框架已经支持的概率分布包括正态、拉普拉斯 (Laplace)、伯努利 (Bernoulli)、离散 (Categorical)、伽玛 (Gamma)、贝塔 (Beta)、泊松 (Poisson)、二项式 (Binomial)、狄利克雷 (Dirichlet) 等, 以及一些最近被提出的重要分布变体, 例如 Gumbel-Softmax [57, 58].

一个 `BayesianNet` 的实例将跟踪在其中构建的所有命名节点, 并提供查询的功能. 查询选项包括该节点当前状态的取值和局部概率.

例2 (变分自编码器) 变分自编码器 (variational autoencoder, VAE) [23] 是结合概率模型和神经网络的优点设计的模型, 在机器学习中得到了广泛的使用. 下面我们使用 VAE 建模二值化 MNIST 手写数字的生成过程:

$$\boldsymbol{z} \sim N(\boldsymbol{0}, \boldsymbol{I}), \quad \boldsymbol{x}_{\text{logits}} = f_{\text{NN}}(\boldsymbol{z}), \quad \boldsymbol{x} \sim \text{Bernoulli}(\sigma(\boldsymbol{x}_{\text{logits}})), \tag{2}$$

其中 $\boldsymbol{z} \in \mathbb{R}^D$ 表示隐变量, $\boldsymbol{x} \in \mathbb{R}^{784}$ 表示分辨率 28×28 的 MNIST 图片对应的向量.

该生成过程展示了绝大多数深度生成模型的思路. 它始于简单分布 (例如标准高斯) 中采样的隐含表示 ($\boldsymbol{z}$), 然后将该表示送入用于模拟高维数据生成过程的深度神经网络 (f_{NN}). 最后, 将一些噪声添加到输出中以获得数据 (如这里的图片 $\boldsymbol{x}$) 的似然函数. 对于二值化 MNIST, 观测噪声选择为伯努利分布, 其均值参数由神经网络的输出定义. 图 3 展示了 VAE 模型的概率程序实现.

3.2 推断和学习算法

除了建模原语外, 概率编程系统的另一个重要组成部分是推断和学习算法. 前文已经提到, 用户编写好概率程序后, 概率编程系统可以从真实数据中学习模型的参数以及推断未观察到的随机变量的后验分布. 为了进一步理解这一过程, 我们考虑一个高度抽象的模型 $p(\boldsymbol{z}, \boldsymbol{x}) = p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$, 其中 $\boldsymbol{z}$ 表示所有隐含变量, $\boldsymbol{x}$ 表示所有的观测到的变量. 上述过程可以归结为给定 $\boldsymbol{x}$ 的观测数据, 推断 $\boldsymbol{z}$ 的可

能取值的后验分布. 我们知道, 贝叶斯定律提供了导出后验分布的方法 [59]2):

$$p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{z},\boldsymbol{x})}{p(\boldsymbol{x})} = \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{p(\boldsymbol{x})}. \tag{3}$$

通常, 除了一些简单的情况, 使用贝叶斯定律进行后验推断是很棘手的. 这是因为边际分布 $p(\boldsymbol{x})$ 的密度函数由难于计算的积分给出: $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})\mathrm{d}\boldsymbol{z}$. 因此, 我们必须诉诸近似贝叶斯推断方法. 经过多年发展, 目前已经有许多快速且广泛适用的近似推断算法. 它们主要分为两类: 变分推断和马尔可夫链蒙特卡洛方法 [8].

变分推断 (variational inference, VI) [60] 是一种基于优化的后验近似算法. 该方法使用一个参数化分布族 $q_\phi(\boldsymbol{z})$, 通过最小化和后验分布 $p(\boldsymbol{z}|\boldsymbol{x})$ 之间的 KL (Kullback-Leibler) 散度来逼近真实后验. 最小化 KL 散度等价于最大化对数边际似然 (log marginal likelihood) $\log p(\boldsymbol{x})$ 的一个下界:

$$\mathcal{L}(\boldsymbol{x};\phi) = \log p(\boldsymbol{x}) - \mathrm{KL}\left[q_\phi(\boldsymbol{z})\|p(\boldsymbol{z}|\boldsymbol{x})\right] = \mathbb{E}_{q_\phi(\boldsymbol{z})}\left[\log p(\boldsymbol{x}|\boldsymbol{z})\right] - \mathrm{KL}\left[q_\phi(\boldsymbol{z})\|p(\boldsymbol{z})\right]. \tag{4}$$

在变分推断的文献中, $q_\phi(\boldsymbol{z})$ 通常被称作变分后验 (或变分分布), 而 $\mathcal{L}(\boldsymbol{x};\phi)$ 被称作证据下界 (evidence lower bound, ELBO).

马尔可夫链蒙特卡洛 (MCMC) 是贝叶斯统计中一种经典的从后验分布采样的算法. 与变分推断不同, MCMC 是渐近无偏的, 这允许用户权衡计算的复杂性和准确性. 因此, 通常认为它比变分推断具有更好的后验近似准确性. MCMC 的基本思想是设计一个马尔可夫链, 其平稳分布是目标分布, 则可以通过模拟链直到其收敛最终获得目标分布的样本. 在实践中, 通常丢弃掉初始阶段的样本以确保链已经收敛. 丢弃掉样本的阶段通常称为预热 (burn-in, 又称 warm-up) 阶段. 一旦我们通过 MCMC 获得后验分布的样本, 就有许多方法可以使用它们. 对于隐变量模型的参数估计, 可以在期望最大化 (expectation-maximum, EM) 算法或蒙特卡洛期望最大化 (MCEM) 算法中使用这些样本 [61,62]. 一种可有效探索高维连续分布的 MCMC 算法是哈密顿蒙特卡洛 [15]. 为了从 $p(\boldsymbol{z}|\boldsymbol{x})$ 采样, HMC 引入了辅助的动量变量 $\boldsymbol{p}$ ($\boldsymbol{p}$ 具有与 $\boldsymbol{z}$ 相同的维数), 接着从联合分布 $p(\boldsymbol{z},\boldsymbol{p}|\boldsymbol{x}) = p(\boldsymbol{z}|\boldsymbol{x})\exp(-\frac{1}{2}\boldsymbol{p}^{\mathrm{T}}\boldsymbol{M}^{-1}\boldsymbol{p})$ 中采样. 其中 $\boldsymbol{M}$ 被称为质量矩阵. 样本是通过模拟哈密顿动力学产生的. $(\boldsymbol{z},\boldsymbol{p})$ 的动力系统沿连续时间 t 按照如下方式演化:

$$\frac{\partial \boldsymbol{z}}{\partial t} = \nabla_{\boldsymbol{p}} H, \quad \frac{\partial \boldsymbol{p}}{\partial t} = -\nabla_{\boldsymbol{z}} H. \tag{5}$$

这里 $H(\boldsymbol{z},\boldsymbol{p}) = -\log p(\boldsymbol{z},\boldsymbol{p}|\boldsymbol{x})$ 是系统的哈密顿量. 这一马尔可夫过程的平稳分布正是期望的后验分布 $p(\boldsymbol{z},\boldsymbol{p}|\boldsymbol{x})$. 实践中, 我们利用广泛使用的 Leapfrog 积分器 [63] 模拟离散时间步长的转移.

下面, 我们分别讨论两类算法在可微概率编程系统中的接口设计.

3.2.1 变分推断

通常, 执行变分推断有两个步骤. 首先是选择将用作变分后验的参数族. 第 2 步是解变分参数 (ϕ) 的优化问题. 除 Edward [53] 等少数最近的编程库, 概率编程系统中典型的 VI 实现主要局限于简单的变分后验. 例如, Stan 概率编程系统 [38] 中实现的 ADVI 算法 [45] 仅使用高斯变分后验. 后面我们将展示如何利用贝叶斯网络的语言建立灵活的变分后验. 在 ZhuSuan 中, 这样的设计使得用户可以通过编程定制变分后验以取得最佳的近似效果.

至于优化方面, 近年来有许多基于梯度的变分推断算法取得了成功 [23,64~66]. 这些方法的区别主要在于他们的变分目标和梯度估计器不同. 为了使变分推断的实现更加自动化, 一个可行的方法是将

2) 正则化贝叶斯给出了一种更加普遍的推断框架 [59], 可以更加灵活地考虑目标后验分布的特性. 为了简洁, 易于理解, 这里我们考虑经典贝叶斯推断.

表 1 ZhuSuan 支持的变分推断算法

Table 1 Variational inference algorithms in ZhuSuan

Objective	Gradient estimator	Supported latent variable	`zs.variational` implementation
ELBO	SGVB [23]	• Continuous, reparameterizable • Gumbel-softmax relaxation	`elbo().sgvb`
	REINFORCE [48, 64]	Any	`elbo().reinforce`
Importance weighted lower bound	IWAE [65]	• Continuous, reparameterizable • Gumbel-softmax relaxation	`iw_objective().sgvb`
	VIMCO [66]	Any	`iw_objective(). vimco`
KL($p\|\|q$)	RWS [67]	Any	`klpq().importance`

它们整合为一系列代理损失函数, 从而为不同的梯度估计器提供一个统一的界面. 这意味着直接计算这些代理损失函数的梯度等价于利用其对应的梯度估计器 (原理将在第 4 节介绍). 表 1 [23,48,64∼67] 总结了 ZhuSuan 中支持的部分变分推断方法, 将它们按照优化目标分组. 可以看到, 其中包括了 3 种变分目标: 证据下界 (ELBO)、重要性加权目标 [65], 以及 KL $[p\|q]$, 其中 p, q 分别表示真实后验和变分后验.

例3 (贝叶斯 Logistic 回归, 续) 下面以例 1 中的贝叶斯 Logistic 回归模型为对象, 介绍变分推断算法的接口设计:

(1) 定义变分后验. $\boldsymbol{w}$ 的真实后验是难于计算的, 且应该在各个维度上具有相关性. 在这里, 我们按照惯例 [60] 进行平均场 (mean field) 假设, 即令变分后验是各维独立的: $q(\boldsymbol{w}) = \prod_{d=1}^{D} q(\boldsymbol{w}_d)$, 其中 D 是权重的维度. 这里可以使用 ZhuSuan 中的建模原语定义各维独立的正态分布, 以作为变分后验. 代码如下:

```
@zs.reuse_variables(scope=''variational'')
def build_variational(D):
    bn = zs.BayesianNet()
    w_mean = tf.Variable(tf.zeros([D]))
    w_logstd = tf.Variable(tf.zeros([D]))
    bn.normal('w', w_mean, logstd=w_logstd, group_ndims=1)
    return bn
```

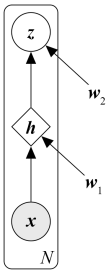
(2) 然后调用 ELBO 作为目标, 传入概率模型的实例 (`meta_bn`), 观测到的数据 (`y`) 以及变分后验 (`variational`).

```
meta_bn = build_blr(x, alpha, D)
variational = build_variational(D)
lower_bound = zs.variational.elbo(meta_bn, observed={'y': y}, variational=variational)
```

(3) 接着, 选择要使用的梯度估计器, 该成员函数会返回需要优化的代理损失. 这里因为 $\boldsymbol{w}$ 是连续的并且可以重参数化 [23] 为 $\boldsymbol{w} = \sigma_{\boldsymbol{w}}\boldsymbol{\epsilon} + \mu_{\boldsymbol{w}}$, 其中 $\mu_{\boldsymbol{w}}$ 和 $\sigma_{\boldsymbol{w}}$ 是正态分布的均值和标准偏差, $\boldsymbol{\epsilon}$ 是各维独立的标准正态随机变量, 我们选择 SGVB 梯度估计器 (各个估计器的适用范围见表 1). 由于计算的代理损失是针对一批数据的, 最后需要取平均值.

```
cost = tf.reduce_mean(elbo.sgvb())
```

(4) 最后, 调用 Tensorflow 优化器对代理损失运行梯度下降. 如前所述, 这一步实际上是在通过 SGVB 梯度估计器优化 ELBO 目标. 此外, 还可以获取 ELBO 值进行验证.



(a)

```python
@zs.reuse_variables("vae_q")
def build_q_net(h_dim, z_dim):
    bn = zs.BayesianNet()
    h = tf.layers.dense(x, h_dim, activation=tf.nn.relu)
    z_mean = tf.layers.dense(h, z_dim)
    z_logstd = tf.layers.dense(h, z_dim)
    bn.normal('z', z_mean, logstd=z_logstd, group_ndims=1)
    return bn
```

(b)

图 4　(网络版彩图) VAE 的变分后验. (a) 概率图模型; (b) 概率程序

Figure 4　(Color online) VAE's variational posterior. (a) The probabilistic graphical model; (b) the probabilistic program

```
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
infer_op = optimizer.minimize(cost)
with tf.Session() as sess:
    for i in range(iters):
        _, elbo_value = sess.run([infer_op, elbo])
```

上面是使用平均场变分后验进行推断的非常简单的示例. 在下面的示例中, 我们将看到如何利用神经网络刻画变分分布.

例4 (变分自编码器, 续)　考虑例 2 中的 VAE 模型. 与 BLR (例 1) 相比, 关键区别在于 VAE 的隐变量 ($\boldsymbol{z}$) 是局部变量, 而 BLR 中 $\boldsymbol{w}$ 是全局隐变量. 由于局部变量的数量随观测值个数呈线性增长, 为它们每一个都拟合一个单独的变分后验代价很大. 在这里, 均摊推断 [68] 的想法变得有用. 具体来说, 可以使用 $\boldsymbol{x}$ 作为输入的神经网络来生成对应隐变量 $\boldsymbol{z}$ 的变分后验. 该网络在 VAE 中通常被称为编码器 (encoder) 或识别网络 (recognition network). 相应的代码见图 4.

设计好变分分布后, 可以按照前述步骤完成推断过程. 需要注意, 因为 $\boldsymbol{z}$ 的变分分布是高斯的, 可以继续使用 SGVB 梯度估计器来优化 ELBO. 这里省略这些步骤, 因为它们与前面的示例非常相似.

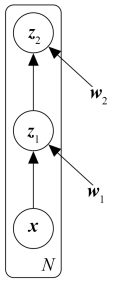
在深的层次模型中, 隐变量之间经常存在条件依赖关系, 在推断时需要考虑. 我们使用 Sigmoid 信念网络模型说明在构建变分后验时如何引入结构化的依赖关系. 这个例子也展示了如何将 VI 应用于离散的隐变量.

例5 (Sigmoid 信念网络)　Sigmoid 信念网络 (Sigmoid belief networks, SBN) 是有向离散隐变量模型, 它与前向神经网络 (feedforward neural network) 和玻尔兹曼机 (Boltzmann machine) [69] 有着紧密联系. 近年来, 深度神经网络的回归为这个旧模型带来了新的活力. 实际上, 深度学习的最早工作, 即著名的深度信念网络 (DBN), 就是无限层的权值捆绑 (tied-weight) 的 SBN. 一个 L 层的 SBN 的生成过程为

$$\begin{aligned}
&\boldsymbol{z}^{(L)} \sim \text{Bernoulli}(\sigma(\boldsymbol{p}^{(L)})),\\
&\boldsymbol{z}^{(\ell-1)} \sim \text{Bernoulli}(\sigma(\boldsymbol{w}^{(\ell)\mathrm{T}}\boldsymbol{z}^{(\ell)})), \quad \ell = L,\ldots,1,\\
&\boldsymbol{x} = \boldsymbol{z}^{(0)},
\end{aligned} \tag{6}$$

其中 σ 是 sigmoid 函数, $\boldsymbol{p}^{(L)}$ 是顶层变量分布的参数, $\boldsymbol{w}$ 是网络权值, $\boldsymbol{z}$ 和 $\boldsymbol{x}$ 分别是隐变量和观测值. 根据定义, 我们可以看到 SBN 是具有多层随机节点的模型. 一个适用于 SBN 模型的变分后验是

$$q(\boldsymbol{z}^{(1:L)}) = \prod_{\ell=2}^{L} q(\boldsymbol{z}^{(\ell)}|\boldsymbol{z}^{(\ell-1)})q(\boldsymbol{z}^{(1)}|\boldsymbol{x}). \tag{7}$$



(a)

```
@zs.reuse_variables(scope="sbn_q"):
def build_sbn_variational(z_dim):
    bn = zs.BayesianNet()
    z1_logits = tf.layers.dense(x, z_dim)
    z1 = bn.bernoulli('z1', z1_logits, dtype=tf.float32)
    z2_logits = tf.layers.dense(x, z_dim)
    bn.bernoulli('z2', z2_logits, dtype=tf.float32)
    return bn
```

(b)

图 5 (网络版彩图) SBN 的变分后验. (a) 概率图模型; (b) 概率程序

Figure 5 (Color online) SBN's variational posterior. (a) The probabilistic graphical model; (b) the probabilistic program

它的设计满足和原始模型具有相同的条件依赖结构. 图 5 展示了 $L = 2$ 时变分后验的图模型以及代码. 从程序中可以看出 $\boldsymbol{z}^{(2)}$ 通过神经网络中的全连接层依赖于 $\boldsymbol{z}^{(1)}$. 注意这里 $\boldsymbol{z}^{(2)}$ 和 $\boldsymbol{z}^{(1)}$ 是离散的隐变量. 在 ZhuSuan 中有两种对它们进行变分推断的方法.

一种方法是使用 Gumbel-softmax 松弛 [57, 58], 即在训练过程中, 将伯努利随机变量松弛为 Gumbel-softmax 变量, 后者在极限情况下服从伯努利分布. 由于 Gumbel-softmax 分布是连续的且可重参数化, 因此可以继续使用 SGVB 梯度估计器. 在测试期间, 需要使用相同的输入参数切换回伯努利随机变量. 另一种方法是直接使用适用于离散隐变量的梯度估计器, 例如 VIMCO 估计器 [66]. 因为 VIMCO 优化的是重要性加权目标, 这是一个多样本下界, 模型和变分后验都需要更改为多样本形式, 如下所示:

```
@zs.meta_bayesian_net(scope=''sbn_multisample'')
def build_sbn_multisample(N, z_dim, x_dim, n_samples):
    bn = zs.BayesianNet()
    z2_logits = tf.zeros([N, n_z])
    z2 = bn.bernoulli('z2', z2_logits, group_ndims=1, dtype=tf.float32, n_samples=n_samples)
    z1_logits = tf.layers.dense(z2, z_dim)
    z1 = bn.bernoulli('z1', z1_logits, group_ndims=1, dtype=tf.float32)
    x_logits = tf.layers.dense(z1, x_dim)
    x = bn.bernoulli('x', x_logits, group_ndims=1)
    return bn
```

请注意代码中添加的 `n_samples` 参数. 这次我们调用 VIMCO 梯度估计器以获取代理损失:

```
lower_bound = zs.variational.iw_objective(meta_bn, observed={'x': x}, variational=variational, axis=0)
cost = tf.reduce_mean(lower_bound.vimco())
```

这里需要设置将沿第 1 轴 (axis 0) 的所有样本用于获得方差缩减的梯度估计.

3.2.2 马尔可夫链蒙特卡洛

由于只需要目标分布的对数密度与其梯度, 前文提到的哈密顿蒙特卡洛 (HMC) 算法非常适合作为可微概率编程系统的推断方法. Stan 概率编程系统 [38] 中实现了这一采样器, 此外还引入一个从预热过程中提取的样本估算质量矩阵的阶段.

这里主要讨论在支持自动微分的通用计算框架 (如 Tensorflow) 上实现 HMC 的接口. 注意到 HMC 的算法本身具有多次迭代, 采样更新过程按照梯度方向进行, 这与多数基于梯度的优化算法类似. 因此可以将 HMC 的接口按照梯度优化器的方式进行设计. 此外, 由于这一过程可以简单并行, 该接口完全可以允许在 CPU 或 GPU 上并行执行多条 HMC 链.

```
z = tf.Variable(0.)

hmc = zs.HMC(step_size=1e-3, n_leapfrogs=10)

sample_op, hmc_info = hmc.sample(
    meta_bn, observed={'x': x}, latent={'z': z})

with tf.Session() as sess:
    for i in range(iters):
        _ = sess.run(sample_op)
```

(a)

```
z = tf.Variable(0.)

optimizer = tf.train.AdamOptimizer(learning_rate=1e-3)

optimize_op = optimizer.minimize(cost(z))

with tf.Session() as sess:
    for i in range(iters):
        _ = sess.run(optimize_op)
```

(b)

图 6 (网络版彩图) 对比 ZhuSuan 中的 (a) HMC 和 (b) Tensorflow 优化器

Figure 6 (Color online) Comparing ZhuSuan's (a) HMC and (b) Tensorflow optimizers

ZhuSuan 中的 HMC 算法与 Tensorflow 优化器有非常相似的用户界面, 如图 6 所示. 同时, 根据隐变量参数存储 (`tf.Variable()`) 的维度, 该优化器可以自动并行执行多条链. 它还提供自动调整参数的选项, 包括步长和质量矩阵. 不过自动确定 Leapfrog 积分器步数的 NUTS 算法 [70] 未包含在内, 因为它是递归算法, 每个单独的链可以有不同的步数, 因此很难在静态计算中并行执行.

4 珠算可微概率编程系统的实现原理

第 3 节介绍了可微概率编程系统典型的接口设计, 包括建模原语以及推断算法两部分. 本节继续以 ZhuSuan 为例, 讲解这些接口背后的实现原理. 我们首先探讨如何实现构建在可微编程框架上的建模原语, 以及这其中模型重用的关键处理.

4.1 概率程序的懒惰执行与模型重用

前文已经介绍, ZhuSuan 的概率程序通过 `MetaBayesianNet` 和 `BayesianNet` 的实例表示概率模型在观测前和观测后的状态. 模型中共有两类节点: 确定性节点和随机节点. 其中, 确定性节点可以由任意 Tensorflow 原语创建, 而随机节点则由 `BayesianNet` 实例的方法创建. 因此, 首先我们需要为 `BayesianNet` 类提供创建各类分布随机节点的成员方法. 当概率程序执行时, 随机节点可能有两种取值: 一种为观测值, 即该节点在数据中被观测到. 另一种为采样值, 即按照该节点的概率分布从中采样得到样本. 在建立模型时, 随机节点的状态无法确定, 因而这类节点的取值也无法立即确定, 因此依赖于该节点的后续计算无法立即执行. 传统的概率编程系统 [41, 43] 为了避免上述问题, 往往引入显式的 `sample` 和 `observe` 语句, 用于在编写模型的同时确定各个随机变量的状态. 然而, 这样的设计使得模型失去了可重用性, 即无法在不同数据上进行多次观测和推断.

要解决这一问题, 一个关键的方法是懒惰执行, 即, 将概率程序的执行推迟到所有的观测状态已经确定后. 因此, 一个方便的设计是将 `MetaBayesianNet` 实例作为这一中间状态. 而将 `BayesianNet` 的构建过程推迟到调用 `MetaBayesianNet` 的 `observe` 方法时. 这一方法的简化实现如下所示:

```
class MetaBayesianNet(object):
    def __init__(self, f, args=None, kwargs=None):
        self._f = f
        self._args = copy.copy(args)
        self._kwargs = copy.copy(kwargs)

    def observe(self, **observations):
        with Local() as local_cxt:
```

```
            local_cxt.observations = observations
            local_cxt.meta_bn = self
            return self._f(*self._args, **self._kwargs)
```

其中 self._f 是该实例中保存的原始的 BayesianNet 构建函数 (例如图 1 中的 build_blr 函数). 由于这一函数将由用户在设计模型时编写 (假设用户不了解模型重用的实现机制), 这里不能将观测量的信息作为参数传递给这一函数. 这一传递需要通过一个局部上下文环境 local_cxt 和 BayesianNet 配合实现. 我们首先引入一个全局栈以存储局部的上下文:

```
class Context(object):
    def __enter__(self):
        type(self).get_contexts().append(self)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        type(self).get_contexts().pop()

    @classmethod
    def get_contexts(cls):
        if not hasattr(cls, ''_contexts''):
            cls._contexts = []
        return cls._contexts

    @classmethod
    def get_context(cls):
        try:
            return cls.get_contexts()[-1]
        except IndexError:
            raise RuntimeError(''No contexts on the stack.'')
```

接着我们定义局部上下文环境 Local 用于存储观测值信息.

```
class Local(Context):
    def __getattr__(self, item):
        return self.__dict__.get(item, None)

    def __setattr__(self, key, value):
        self.__dict__[key] = value
```

最后, 当模型构建函数中的 bn=zs.BayesianNet() 语句调用时, 要确保这一 BayesianNet 实例会查看局部上下文环境中的观测值.

```
class BayesianNet(object):
    def __init__(self):
        self._nodes = {}
        try:
            self._local_cxt = Local.get_context()
        except RuntimeError:
            self._local_cxt = None
```

有了这些信息, 在调用 BayesianNet 实例的成员方法创建随机节点时, 就可以根据概率程序的执行状态 (局部的上下文) 确定随机节点的取值了. 最后, 为了使用户在调用模型构建函数时自动进行懒惰执行, 可以使用前面提到的 meta_bayesian_net 装饰器:

```
def meta_bayesian_net():
    def wrapper(f):
        @wraps(f)
        def _wrapped(*args, **kwargs):
            meta_bn = MetaBayesianNet(f, args=args, kwargs=kwargs)
            return meta_bn
        return _wrapped
    return wrapper
```

它会将模型构建函数及传入参数保存起来, 并返回一个对应的 `MetaBayesianNet` 实例, 等待后续通过调用 `observe` 成员函数真正执行.

这一懒惰执行机制可以妥善处理具有不确定状态的随机变量, 从而避免了在观测状态常发生变化的概率程序中维护梯度信息所导致的问题. 具体来说, 由于静态可微编程工具包 Theano [50] 和 Tensorflow [36] 需要通过构建计算图来保存梯度信息, 随机变量的观测状态发生改变时, 不采用此种设计的 PyMC3 [52] 和 Edward [53] 需要修改已创建好的计算图来实现模型的重用. PyMC3 (基于 Theano) 依赖于官方支持的 `theano.clone()` 函数复制和重新创建子图, 而 Edward (基于 Tensorflow) 则必须利用 Tensorflow 非公开的底层接口实现自己的 `copy()` 函数以复制计算子图. 该解决方案导致 Edward 不能很好的支持控制流. 上述懒惰执行的想法不依赖于更改创建的计算图, 因而不存在更改计算图导致的梯度信息丢失问题, 甚至不依赖于静态计算图, 完全可以迁移到动态可微编程框架上.

下面我们关注可微概率编程框架中变分推断算法的实现. 这其中最为重要的一步是梯度估计, 即如何计算变分目标函数的梯度.

4.2 变分目标, 梯度估计器和代理损失

前文已经介绍, 变分目标函数通常是由变分分布和真实后验分布之间的某种距离导出, 其中最常见的是 ELBO (式 (4)). 注意到 ELBO 由两项求和得到, 第 1 项为对数联合密度函数 ($\log p(\boldsymbol{x}, \boldsymbol{z})$) 在变分后验下的期望, 第 2 项为变分后验分布的熵, 即 $-\mathbb{E}_{q(\boldsymbol{z})}[\log q(\boldsymbol{z})]$. 两项均包含积分 (对 $q(\boldsymbol{z})$ 的期望). 因此, 实际中需要使用蒙特卡洛估计 ELBO 的值. 这一过程在前述的模型原语下不难实现. 首先, 我们从变分分布中采样得到隐变量 $\boldsymbol{z}$ 的样本, 接着调用模型 `MetaBayesianNet` 实例的 `observe()` 方法, 传入 $\boldsymbol{z}$ 的样本作为观测值, 在返回的 `BayesianNet` 实例中查询 $\boldsymbol{z}$ 节点和 $\boldsymbol{x}$ 节点们的局部对数概率值, 求和即可得到对数联合概率密度 $\log p(\boldsymbol{x}, \boldsymbol{z})$. 再将获得的对数联合概率密度对多个样本取平均, 即可得到 ELBO 的无偏估计值.

值得注意的是, 虽然自动微分框架支持对这一无偏估计值直接计算梯度, 这样计算得到的梯度却往往并不能用来优化变分目标. 这里问题的关键在于运用采样近似了期望所代表的积分. 注意到, 除了少数情况 [23], 从 $q(\boldsymbol{z})$ 中采样得到的样本 $\boldsymbol{z}$ 对 $q(\boldsymbol{z})$ 的参数并不可导. 这时直接优化 $\mathbb{E}_{q_\phi(\boldsymbol{z})}[f(\boldsymbol{z})]$ 的估计值将不会更新参数 ϕ. 而这一积分的结果对 ϕ 是存在导数的. 处理这一优化问题的正确方法是使用梯度估计器, 常见的梯度估计器包括重参数化估计器 (SGVB) [23]、REINFORCE [48] 等 (见表 1). 例如, REINFORCE 估计器利用了等式

$$\nabla_\phi \mathbb{E}_{q_\phi(\boldsymbol{z})}[f_\phi(\boldsymbol{z})] = \mathbb{E}_{q_\phi}[\nabla_\phi f(\boldsymbol{z}) + f(\boldsymbol{z}) \nabla_\phi \log q_\phi(\boldsymbol{z})]. \tag{8}$$

由于估计器的实现方法各不相同, 为了将它们以统一的接口提供给用户, 可以使用代理损失函数. 具体来说, 我们可以根据自动微分框架的特性, 构造一个不同与变分目标估计值的损失函数, 使得该函数的梯度恰好等于变分目标函数梯度估计器给出的值. 例如, REINFORCE 估计器可以由下述的代理

损失函数实现 (简化版):

```
def reinforce(f, x, log_q, sample_axis=0):
    fx = f(x)
    cost = fx + tf.stop_gradient(fx) * log_q(x)
    cost = tf.reduce_mean(cost, sample_axis)
    return cost
```

这里 `tf.stop_gradient` 是 Tensorflow 中自动微分系统的原语, 用于切断该张量的后向传播梯度.

4.3 多链并行的哈密顿蒙特卡洛

在实际使用中, 希望最大化 MCMC 算法的采样效率, 即在尽可能短的运行时间内, 得到足够数量的样本. 如 3.2.2 小节中所述, 给定一条马尔可夫链, 在经过预热阶段后, 每步迭代可以产生一个样本. 然而同一条马尔可夫链每步产生的样本是高度相关的, 只有间隔超过一定步数的两个样本才可以被认为是符合要求的, 彼此独立的样本. 独立样本的数量被称为有效样本数 (effective sample size, ESS). 为了得到尽可能多的样本, 可以并行地模拟多条马尔可夫链, 总的有效样本数即为每条链有效样本数的总和. 多链并行比单链模拟在实际中更有优势. 首先, 虽然理论上 MCMC 算法是渐进无偏的, 但在实际中模拟只能进行有限步, 因此初值会对采样产生影响. 多链并行减少了初值对采样的影响. 其次, 虽然理论上多链并行与单链时间复杂度相当, 但是实际中多链并行可以更好地利用 GPU 的并行计算能力, 因而比单链更加高效.

ZhuSuan 在 HMC 的实现中支持了多链并行. 为了利用多链并行, ZhuSuan 需要分别得到每条链的概率密度及梯度. 该功能可以通过将每条链对应的不同样本考虑进概率程序实现. 以 VAE 为例, 若希望采样变量 $\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{x})$, 且对每个数据点并行采样 10 条链, 则对应的概率程序如下所示:

```
@zs.meta_bayesian_net(''vae'', reuse_variables=True)
def build_vae(N, D):
    bn = zs.BayesianNet()
    z_mean = tf.zeros([N, D])
    z = bn.normal('z', z_mean, std=1., group_ndims=1, n_samples=10)
    h = tf.layers.dense(z, 500, activation=tf.nn.relu)
    x_logits = tf.layers.dense(h, 784)
    bn.bernoulli('x', x_logits, group_ndims=1)
    return bn
```

相比图 3(b) 中的概率程序, 此处在定义 $\boldsymbol{z}$ 的生成过程中加入了选项 `n_samples=10`, 表示采样 10 份独立的样本. 此时 $\boldsymbol{z}$ 的维度为 `[10,N,D]`, 其中第 1 个轴表示样本, 第 2 个轴表示数据点. 此概率程序会对每个数据点的每份样本分别计算概率密度. 其中概率密度的维度为 `[10,N]`, 而梯度的维度与 $\boldsymbol{z}$ 本身的维度相同, 为 `[10,N,D]`. 因此 HMC 将同时模拟这 $10N$ 条马尔可夫链. 以下我们以 HMC 中的一个关键步骤为例, 讨论多链实现与单链实现的不同. 这一步骤为 Metropolis-Hastings (MH) 算法, 即通过比较转移前和转移后的样本位置, 确定是否接受该次转移. 这是为了校准离散模拟哈密顿动力学所导致的近似误差.

```
# Leapfrog
new_q, new_p = self._leapfrog(q, p, new_step_size, get_gradient, mass)

# MH-Test
old_hamiltonian, new_hamiltonian, old_log_prob, new_log_prob, acceptance_rate = get_acceptance_rate(
    q, p, new_q, new_p, get_log_posterior, mass, self.data_axes)
```

```
u01 = tf.random_uniform(shape=tf.shape(acceptance_rate))
if_accept = tf.less(u01, acceptance_rate)

expanded_if_accept = if_accept
for i in range(len(self.data_axes)):
    expanded_if_accept = tf.expand_dims(expanded_if_accept, axis=-1)
new_q = tf.where(expanded_if_accept, new_q, q)

update_q = latent_v.assign(new_q)
```

代码片段中显示, 我们首先执行 Leapfrog 积分器, 获得下一步转移位置的信息, 包括新位置的哈密顿量以及新的对数密度值, 接着通过 MH 算法计算出接受概率 `acceptance_rate`. 在以上的 VAE 例子中, 由于存在 $10N$ 条马尔科夫链, 则该接受概率具有维度 `[10,N]`. 按照接受概率进行采样, 我们得到各条链上是否接受当前转移的布尔值 `if_accept`. 为了同时更新各条链的位置, 这里需要采用并行的条件控制语句 `tf.where`. 为此我们首先将布尔值 `if_accept` 扩展成和位置变量 `q` 具有相同维度的变量 `expanded_if_accept`, 在 VAE 的例子中即扩展成 `[10,N,1]`. 然后根据每条链对应的布尔值确定下一时间是转移到新位置 `new_q` 还是保留上一位置 `q`. 最后, 将更新后的样本位置赋值给存储采样值的变量 `latent_v`.

5 总结与展望

本文介绍了开源概率编程库 “珠算” (ZhuSuan), 讨论了可微概率编程系统的基本设计和实现原理. 我们介绍了可微概率编程系统的两大基本构成: 建模原语以及推断与学习算法, 并就懒惰执行, 模型重用, 梯度估计, 并行马尔可夫链采样等关键技术的实现原理进行了详细的分析.

随着机器学习的研究重点从具有大量标记数据的特定任务逐渐转向只具有稀疏学习信号且充满不确定性的开放世界, 兼具结构知识和拟合能力, 能够探索未知的概率学习系统将扮演越来越重要的角色. 未来, 我们将在更多的应用中看到概率模型的可微编程技术. 希望本文介绍的可微概率编程系统设计与实现方法能为更进一步探索以上问题提供新的思路.

最后, 深度神经网络以及可微概率编程充分利用大数据, 是数据驱动的人工智能技术的代表. 此外, 知识驱动的符号化模型 (如专家系统) 是人工智能的另一种重要方向, 相比神经网络, 这类方法具有可解释性高、鲁棒性强、适合推理、决策等任务. 近期的发展趋势是将知识驱动的符号化模型与数据驱动的神经网络相融合 [71], 发展神经符号模型. 未来将进一步扩展 “珠算”, 支持可微神经符号模型的编程.

致谢 感谢曾经参与珠算开发和维护的同学以及开源社区贡献者: Shengyang SUN, Yucen LUO, Yihong GU, Yuhao ZHOU, Haowen XU, Shuyu CHENG, Ziyu WANG, Alexander BOTEV, Huajun WU, Wenzhe LI, Jiayi YUAN. 感谢 Haoyu LIANG 为珠算编程库设计 Logo.

参考文献

1 LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444

2 Hinton G, Deng L, Yu D, et al. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process Mag, 2012, 29: 82–97

3 Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2012. 1097–1105
4 Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2014. 3104–3112
5 Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016, 529: 484–489
6 Dong Y P, Liao F Z, Pang T Y, et al. Boosting adversarial attacks with momentum. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018
7 Ghahramani Z. Probabilistic machine learning and artificial intelligence. Nature, 2015, 521: 452–459
8 Zhu J, Chen J F, Hu W B, et al. Big learning with Bayesian methods. Natl Sci Rev, 2017, 4: 627–651
9 Gal Y. Uncertainty in deep learning. Dissertation for Ph.D. Degree. Cambridge: University of Cambridge, 2016
10 Cosentino J, Zhu J. Generative well-intentioned networks. In: Proceedings of Advances in Neural Information Processing Systems, 2019
11 Lake B M, Salakhutdinov R, Tenenbaum J B. Human-level concept learning through probabilistic program induction. Science, 2015, 350: 1332–1338
12 Li C X, Xu K, Zhu J, et al. Triple generative adversarial nets. In: Proceedings of Advances in Neural Information Processing Systems, 2017. 4088–4098
13 Koller D, Friedman N. Probabilistic Graphical Models: Principles and Techniques. Cambridge: MIT Press, 2009
14 Wainwright M J, Jordan M I. Graphical Models, Exponential Families, and Variational Inference. Hanover: Foundations and Trends®in Machine Learning, 2008
15 Neal R M. MCMC using Hamiltonian dynamics. In: Handbook of Markov Chain Monte Carlo. Boca Raton: CRC Press, 2011
16 Koller D, McAllester D, Pfeffer A. Effective Bayesian inference for stochastic programs. In: Proceedings of Proceedings of the 14th National Conference on Artificial Intelligence, 1997. 740–747
17 van de Meent J W, Paige B, Yang H, et al. An introduction to probabilistic programming. 2018. ArXiv:1809.10756
18 Blei D M, Ng A Y, Jordan M I. Latent Dirichlet allocation. J Mach Learn Res, 2003, 3: 993–1022
19 Rainforth T W G. Automating inference, learning, and design using probabilistic programming. Dissertation for Ph.D. Degree. Oxford: University of Oxford, 2017
20 Grosse R B. Model selection in compositional spaces. Dissertation for Ph.D. Degree. Cambridge: Massachusetts Institute of Technology, 2014
21 Salakhutdinov R, Mnih A. Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In: Proceedings of International Conference on Machine Learning, 2008. 880–887
22 Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets. Neural Comput, 2006, 18: 1527–1554
23 Kingma D P, Welling M. Auto-encoding variational Bayes. 2013. ArXiv:1312.6114
24 Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets. In: Proceedings of Advances in Neural Information Processing Systems, 2014. 2672–2680
25 Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks. 2015. ArXiv:1511.06434
26 Jin W, Barzilay R, Jaakkola T. Junction tree variational autoencoder for molecular graph generation. In: Proceedings of International Conference on Machine Learning, 2018. 2323–2332
27 Kingma D P, Mohamed S, Rezende D J, et al. Semi-supervised learning with deep generative models. In: Proceedings of Advances in Neural Information Processing Systems, 2014. 3581–3589
28 Salimans T, Goodfellow I, Zaremba W, et al. Improved techniques for training GANs. In: Proceedings of Advances in Neural Information Processing Systems, 2016. 2226–2234
29 Xu H W, Chen W X, Zhao N W, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal KPIS in web applications. In: Proceedings of World Wide Web Conference, 2018. 187–196
30 Rezende D, Danihelka I, Gregor K, et al. One-shot generalization in deep generative models. In: Proceedings of International Conference on Machine Learning, 2016. 1521–1529
31 Neal R M. Bayesian learning for neural networks. Dissertation for Ph.D. Degree. Toronto: University of Toronto, 1995
32 MacKay D J. Bayesian non-linear modeling for the prediction competition. In: Proceedings of Maximum Entropy and

Bayesian Methods, 1996. 221–234
33 Wang Z Y, Ren T Z, Zhu J, et al. Function space particle optimization for Bayesian neural networks. In: Proceedings of International Conference on Learning Representations, 2019
34 Shi J X, Sun S Y, Zhu J. A spectral approach to gradient estimation for implicit distributions. In: Proceedings of International Conference on Machine Learning, 2018. 4644–4653
35 Johnson M, Duvenaud D K, Wiltschko A, et al. Composing graphical models with neural networks for structured representations and fast inference. In: Proceedings of Advances in Neural Information Processing Systems, 2016. 2946–2954
36 Abadi M, Agarwal A, Barham P, et al. Tensorflow: large-scale machine learning on heterogeneous distributed systems. 2016. ArXiv:1603.04467
37 Spiegelhalter D, Thomas A, Best N, et al. BUGS 0.6: Bayesian Inference Using Gibbs Sampling (Addendum To Manual). Cambridge: Medical Research Council Biostatistics Unit, Institute of Public Health, 1997
38 Carpenter B, Gelman A, Hoffman M, et al. Stan: a probabilistic programming language. J Stat Softw, 2017, 76: 1–32
39 Minka T, Winn J, Guiver J, et al. Infer.NET 0.3. Microsoft Research Cambridge, 2018. http://dotnet.github.io/infer
40 Milch B, Marthi B, Russell S, et al. BLOG: probabilistic models with unknown objects. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, 2005. 1352–1359
41 Goodman N D, Mansinghka V K, Roy D, et al. Church: a language for generative models. In: Proceedings of Conference on Uncertainty in Artificial Intelligence, 2008. 220–229
42 Mansinghka V, Selsam D, Perov Y. Venture: a higher-order probabilistic programming platform with programmable inference. 2014. ArXiv:1404.0099
43 Wood F, Meent J W, Mansinghka V. A new approach to probabilistic programming inference. In: Proceedings of International Conference on Artificial Intelligence and Statistics, 2014. 1024–1032
44 Goodman N D, Stuhlmüller A. The design and implementation of probabilistic programming languages. 2014. http://dippl.org
45 Kucukelbir A, Tran D, Ranganath R, et al. Automatic differentiation variational inference. J Mach Learn Res, 2017, 18: 1–45
46 Winn J, Bishop C M. Variational message passing. J Mach Learn Res, 2005, 6: 661–694
47 Minka T P. Expectation propagation for approximate Bayesian inference. In: Proceedings of Uncertainty in artificial intelligence, 2001. 362–369
48 Williams R J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn, 1992, 8: 229–256
49 Paisley J, Blei D M, Jordan M I. Variational Bayesian inference with stochastic search. In: Proceedings of International Conference on Machine Learning, 2012. 1363–1370
50 Al-Rfou R, Alain G, Almahairi A, et al. Theano: a Python framework for fast computation of mathematical expressions. 2016. ArXiv:1605.02688
51 Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library. In: Proceedings of Advances in Neural Information Processing Systems, 2019. 8024–8035
52 Salvatier J, Wiecki T V, Fonnesbeck C. Probabilistic programming in Python using PyMC3. PeerJ Comput Sci, 2016, 2: e55
53 Tran D, Kucukelbir A, Dieng A B, et al. Edward: a library for probabilistic modeling, inference, and criticism. 2016. ArXiv:1610.09787
54 Shi J X, Chen J F, Zhu J, et al. ZhuSuan: a library for Bayesian deep learning. 2017. ArXiv:1709.05870
55 Bingham E, Chen J P, Jankowiak M, et al. Pyro: deep universal probabilistic programming. J Mach Learn Res, 2019, 20: 973–978
56 Siddharth N, Paige B, van de Meent J W, et al. Learning disentangled representations with semi-supervised deep generative models. In: Proceedings of Advances in Neural Information Processing Systems, 2017. 5925–5935
57 Jang E, Gu S X, Poole B. Categorical reparameterization with Gumbel-softmax. 2016. ArXiv:1611.01144
58 Maddison C J, Mnih A, Teh Y W. The concrete distribution: a continuous relaxation of discrete random variables. 2016. ArXiv:1611.00712
59 Zhu J, Chen N, Xing E P. Bayesian inference with posterior regularization and applications to infinite latent SVMs. J

Mach Learn Res, 2014, 15: 1799
60 Blei D M, Kucukelbir A, McAuliffe J D. Variational inference: a review for statisticians. J Am Stat Assoc, 2017, 112: 859–877
61 Wei G C G, Tanner M A. A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. J Am Stat Assoc, 1990, 85: 699–704
62 Chen J F, Zhu J, Teh Y W, et al. Stochastic expectation maximization with variance reduction. In: Proceedings of Advances in Neural Information Processing Systems, 2018
63 Leimkuhler B, Reich S. Simulating Hamiltonian Dynamics. Cambridge: Cambridge University Press, 2004
64 Mnih A, Gregor K. Neural variational inference and learning in belief networks. In: Proceedings of International Conference on Machine Learning, 2014. 1791–1799
65 Burda Y, Grosse R, Salakhutdinov R. Importance weighted autoencoders. 2015. ArXiv:1509.00519
66 Mnih A, Rezende D. Variational inference for Monte Carlo objectives. In: Proceedings of International Conference on Machine Learning, 2016. 2188–2196
67 Bornschein J, Bengio Y. Reweighted wake-sleep. 2014. ArXiv:1406.2751
68 Rezende D, Mohamed S. Variational inference with normalizing flows. In: Proceedings of International Conference on Machine Learning, 2015. 1530–1538
69 Neal R M. Connectionist learning of belief networks. Artif Intell, 1992, 56: 71–113
70 Hoffman M D, Gelman A. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. J Mach Learn Res, 2014, 15: 1593–1623
71 Zhang B, Zhu J, Su H. Toward the third generation of artificial intelligence. Sci Sin Inform, 2020, 50: 1281–1302 [张钹, 朱军, 苏航, 等. 迈向第三代人工智能. 中国科学: 信息科学, 2020, 50: 1281–1302]

ZhuSuan: design and implementation of differentiable probabilistic programming libraries

Jiaxin SHI[1,2], Jianfei CHEN[1] & Jun ZHU[1*]

1. *Department of Computer Science and Technology, BNRist Center, Tsinghua-Bosch Joint Center for ML, Institute for AI, State Key Laboratory for Intelligence Technology & System, Tsinghua University, Beijing* 100084*, China*;
2. *Microsoft Research New England, Cambridge* MA 02142*, USA*
* Corresponding author. E-mail: dcszj@tsinghua.edu.cn

Abstract Probabilistic models provide a set of powerful tools to deal with uncertainty that is pervasive in machine learning applications. Probabilistic programming utilizes computer programs to represent probabilistic models, and it supports the functions of sampling as well as probabilistic inference conditioned on arbitrary observations. Traditionally, the dependency relationship in probabilistic programming is mainly linear or generalized linear, which serves as the basis of many successful models and inference algorithms. However, such linearity also limits the expressiveness and flexibility of probabilistic programs. Differentiable probabilistic programming allows probabilistic programs to have nonlinear dependency under a proper parameterization form (e.g., neural networks) and can learn unknown parameters from data via gradient-based methods. This programming paradigm is easy to extend, largely avoids the tedious model selection process, and makes the deployment of probabilistic models possible in an end-to-end manner. This article presents ZhuSuan, an open-source library for differentiable probabilistic programming. Taking ZhuSuan as an example, we discuss the design and implementation of differentiable probabilistic programming systems.

Keywords probabilistic models, probabilistic programming, Bayesian inference, variational inference, deep learning