概率编程 HackPPL 综述

摘 要：HackPPL 是在 Hack 语言的基础上拓展开发的概率编程语言，具有通用的预测模型，用户可以在 Hack 代码基础上进行概率模型的建模和预测。通过语言的拓展、开发者工具的结合，HackPPL 利用自身内嵌的概率编程语言特性和功能实现特定领域的目标。本文总结了 HackPPL 语言中的一些设计和实现。

关键词：概率编程；Hack；HackPPL；tensor

1 引言

概率分析在机器学习的应用领域正变得愈发重要，尤其是在信息量巨大但有效性不高的现代信息社会。诸如贝叶斯等概率模型提供了有效的结合模型参数和相关性来提取不确定性的方法，但是对用户而言，要求特定的概率专业知识和技能仍然是不友好的。概率编程旨在通过提供统一的表达生成模型的语法以及内置用于测试假设的推理机来减轻开发者的负担，提高人们在从事相关领域分析的效率。

本文旨在分析和归纳HackPPL的一些设计特性和用法。第二节通过示例介绍 HackPPL 建模和预测方法的使用。第三节概述 HackPPL 为了提供简洁的调用而对一些 Hack 语法做出的改进。第四节简要介绍了 HackPPL 中一些对数据和模型的处理。

2 HackPPL 基础

本节通过实际的样例来简要地对 HackPPL 进行一些介绍。

2.1 线性回归模型

HackPPL 模型实现了 PPLModel 的接口并且通过属性<<__PPL>>和其他Hack代码加以区分。从例 1 代码中可以看到，用户可以通过 sample 关键字来定义随机变量和组合生成模型，sample 可以看作是一种运算操作，既可以用来从分布中生成新的变量，还可以用来调节和观察生成的那些变量。每一个 sample 语句都必须关联一个特定的用于操作的标识符，可以看作是它的名字。

```
<<__PPL>>
class LinearRegressionModel implements PPLModel<void> {
  public function __construct(private Tensor $x) {}
  public function run(): void {
    $num_coeffs = $this->x->size()[1];
    $w = sample(new Normal(
      Tensor::zeros(vec[$num_coeffs]),
      Tensor::ones(vec[$num_coeffs])), 'w');
    $tau = sample(new Gamma(
      Tensor::scalar(1.),
      Tensor::scalar(1.)), 'tau')
        ->expand($this->x->size()[0]);
    sample(new Normal($x->matmul($w), $tau, true), 'y');
  }
}
```

例 1. 线性回归模型

2.2 执行预测

HackPPL 提供通用的建模方法，具体实现则采用的是基于路径的预测方法。路径是建模程序通过依次计算样本和参数建立的，计算的过程则取决于实际的预测模型，比如对样本取对数等等。

```
$obs = dict['y' => $y];
$model = new LinearRegressionModel($x);
$hist = PPLInfer::hmc($model, $obs)->history()->run($num_iter);
$hist->getSample('w');
```

例 2. 调节和预测

HackPPL 中预测的方法和建模的过程是分开的，用户可以根据样本选择最合适该模型的预测方法。例 2 中的代码展示了如何在之间建立的线性回归模型上进行预测。在建立了样本标识符到函数值的映射后，调用 PPLInfer 类执行 Hamiltonian Monte Carlo 预测，得到回归模型的预测结果的各个参数。

3 HackPPL 语言特性

3.1 协程

为了最优化预测结果，Monte Carlo 预测算法引入大量的样本并且采用了基于路径选择的方法，一个关键的提升在于过滤重

复执行的建模过程。在例 3 代码中我们可以看到，在生成大量的样本路径的过程中，可以通过分支的形式判断已有的路径结果从而不需要重复地访问第一个样本。这个策略可以为建模的过程节省相当可观的时间。

```
$flip = sample(new Bernoulli(Tensor::scalar(0.7))), 'flip');
if ($flip->first() === 0) {
  sample(new Categorical(Tensor::vector(vec[1, 2])), 'res');
} else {
  sample(new Categorical(Tensor::vector(vec[1, 3])), 'res');
}
```

例 3. 控制

在 HackPPL 语言中，模型都是以协程的方式建立并且在预测部分的代码中通过大量的 multi-shot 的方式实现，这种方式使得我们可以通过多状态来控制循环和异常的分支，更重要的，通过协程的方式能大幅度提高运算的效率。协程有以下两个基本特征：协程的本地变量在该协程从中断状态恢复时会被重新加载；当程序离开当前函数或功能单元时当前协程被中断，而主程序控制回到某一块时从上次中断的位置继续执行。

Multi-shot 的协程实现允许模型在中断后下次执行时从中断处恢复状态并继续建立，前面提到 sample 操作定义随机变量作为标识符，运算时的空间可以被充分利用。当控制程序从建模过程切换到 sample 运算的协程中，建模过程被中断，后续从 sample 过程返回时则可以从中断处被恢复。

3.2 框架

sample 运算使用了协程框架来组织各个子运算模块的执行，同样的，还可以用这样的框架做更多类似的组织。在协程框架的基础上，用户可以具体实现协程的具体内容，只要在中断的位置记录好恢复协程时继续执行的点即可。

例 4 的代码展示了一个定义协程的过程，对于 Hack 的代码，协程都用 coroutine 保留字表示，通过 suspend 操作实现协程之间的互相中断和恢复执行。需要注意的是，非协程的程序段无法直接调用协程。在例 4 中，coroutine 中断了 suspendMultiple，后者是标准库中的一个用于具体实现 multi-shot 方式的接口。

```
class MyClass {
  ?MultishotContinuation<string> $suspended_coroutine;
  public coroutine function myCoroutine(): string {
    print "Started the coroutine";
    $resumed_value = suspend suspendMultiple(
      coroutine ($my_suspended_coroutine) ==> {
        $this->suspended_coroutine = $my_suspended_coroutine;
    });
    print "Continued the coroutine";
    return $resumed_value;
  }
}
```

例 4. 协程的中断

例 5 的代码中展示了协程是如何调用的。myCoroutine 通过一个特定的协程库函数 StartCoroutine::start 被调用，由于是被非协程的程序段调用，需要定义一个回调函数来返回到调用协程时中断的位置继续执行。当样例中的协程开始运行时，首先打印内容“Started the coroutine”，然后中断进入协程 suspendMultiple。在代码例 4 中开发者存储中断时位置作为后续返回时程序恢复的起点，在调用者调用方法 resumeAsync 后恢复到该协程继续执行，在执行完成后通过调用 CoroutineBack 返回到调用它的地方，“First”和“Second”被依次打印出来。

```
class CoroutineCallback implements Continuation<string> {
  public function resume(string $coroutine_return): void {
    print $coroutine_return;
  }
}
StartCoroutine::start(
  coroutine () ==> suspend $my_class->myCoroutine(),
  new CoroutineCallback());
// Later on...
$my_class->suspended_coroutine->resumeAsync("First");
$my_class->suspended_coroutine->resumeAsync("Second");
```

例 5. 协程 resume 的示例

3.3 协程的实现

和 Kotlin 和 C#中的实现类似，协程是以状态机的方式在编译时实现的，其中函数（或程序段）的中断点经过转换作为额外的参数传递，每个协程都被编译成包含当前状态的域以及状态机实现方法的匿名类，保留字 suspend 则被翻译成 goto 来执行。当一个协程被中断，本地状态被存储起来，当 multi-shot 形式的协程被中断时，本地状态被拷贝存储，然后再通过 goto 指令跳转到进入到当前协程的前一个协程的中断点。

因为协程的回调是通过重新恢复上个协程的中断点而不是 return 返回，Hack 语

言由于缺少对尾递归的优化而可能导致程序执行时发生栈溢出（stack overflow）的异常。为了解决这个问题，HackPPL 实现了一个专门管理协程的框架 coroutine manager，这个框架能保证协程之间的匿名调用避免发生栈溢出的异常。

3.4 语法的调整

在前面已经提到过，为了能够在主程序控制多个模型的执行和预测，预测方法是通过实现 sample 操作来实现的。sample 操作必须通过 suspend 语句调用，并且实现时调用 sample 的函数必须使用标识符 coroutine。示例 6 的代码展示了对象 InferenceState 的使用，他能起到汇集信息的作用。有了前面的语法，实际上能推敲出模板的实现细节，进而导致用户其实拥有直接访问一些预测方法对象的权限。

```
coroutine function model(InferenceState $state): int {
  $flip = suspend $state->sample(new Bernoulli(
    Tensor::scalar(0.7))), 'flip');
  if ($flip->first() === 0) {
    return suspend $state->sample(new Categorical(
      Tensor::vector(vec[1,2])), 'res'));
  } else {
    return suspend $state->sample(new Categorical(
      Tensor::vector(vec[1,3])), 'res'));
  }
}
```

例 6. 底层模型控制的实现

为了解决这样潜在的隐患，HackPPL 引入新的语法来将这种实现的细节和建立模型的语法区分开。在例 3 代码中我们已经看到了属性<<__PPL>>的使用，它负责告诉编译器 sample 关键字是一个函数，当作保留字，这样 sample 就可以被每个预测方法函数分别实现。所以，尽管各个预测方法可能都用到某个库函数来实现 multi-shot 模型的建立过程，使用建模方法的用户并不需要理解每一个预测的实际过程。通过这样的方法向建立模型的过程隐藏了 InferenceState 状态，用户可以像调用 Hack 的原始函数一样直接调用像 sample 这样的方法。

4 数据和模型的处理

HackPPL 对随机数据和模型的处理引用了 PyTorch 的方法。其中，tensor 的广泛使用影响了很多建模和预测方法的效率。尤其在处理一些较大较复杂的模型时，张量（tensor）的引入起到了至关重要的作用。

4.1 连续分布的数据

Hack 虚拟机引入了 PyTorch 的 tensor 框架作为拓展，tensor 可以简易的理解为多维数组，HackPPL 引入 tensor 这种数据结构用于各种分布和样本的处理，同时提供了方便的对模型的并行计算的抽象。示例 1 和 3 的代码分别展示了标量和向量的构造，除此以外，还可以构造更高维的 tensor。

4.2 离散数据

尽管引入 PyTorch 后，tensor 数据结构使得连续分布的数据的计算变得很方便，但它却难以处理离散的数据。使用 sample 处理离散随机分布的变量是普遍且及其重要的，所以在 HackPPL 中引入了新的抽象框架——DTensor。一个 DTensor 可以被看作是一系列的数或字符串，它能够和一维编码的 tensor 相互转换，转换成 tensor 是为了方便进行数值运算。示例 7 的代码展示了 DTensor 的用法，构造方法需要用户提供名字标签以及一个从下标到标签的映射。

```
$labels = vec[1, 3, 2];
$vocab = vec['a', 'c', 'b'];
$dtensor = new DTensor($labels, $vocab);
$one_hot_tensor = $dtensor->toOneHotEncodedTensor();
```

例 7. DTensor 构造函数的示例

4.3 分布

HackPPL 提供很多种高效的基于 tensor 的分布，并且提供方便开发者实现的接口，例如 sample 和 score。正如下面的例 8 中看到的，进行 tensor 的运算时，HackPPL 同时也支持一次性进行批量的（batch）样本的运算。

```
$alphas = Tensor::matrix(vec[
  vec[0.25, 0.25, 0.25, 0.25],
  vec[0.3, 0.6, 0.2, 0.1]]);
$dirich = new Dirichlet($alphas);
$val = $dirich->sample(5); // $val has shape 5×2×4
$score = $dirich->score($val);
```

例 8. batch 运算

在例 8 中，构造了 2 维的迪利克雷分布，

每一个分布都有 4 维。在实际使用时，维度的数据通常使得用户能更好地了解和利用 tensor 在提升运算性能时的作用。

5 小结

本文介绍了 HackPPL 的一些功能和特性，通过几个示例展示了如何使用 HackPPL 实现概率编程相关的函数和功能。通过将概率编程的算法结合到一般用途的语言，HackPPL 希望得到更好的运行效率，更方便更友好的接口调用。在 HackPPL 的底层实现中，既继承了 Hack 的语法，也拓展了诸如 PyTorch 这样的工具包。

参考文献：

Jessica Ai, Nimar S. Arora, Ning Dong, Beliz Gokkaya, Thomas Jiang, Anitha Kubendran, Arun Kumar, Michael Tingley, and Narjes Torabi. 2019. HackPPL: A Universal Probabilistic Programming Language. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'19), June 22, 2019, Phoenix, AZ, USA. ACM, New York, NY, USA,9 pages. https://doi.org/10.1145/3315508.3329974