Julia 语言及其对新兴机器学习应用的支持探索

张昱[*1], 蔡文韬[†2], 戴路[‡3], 董恒[§4], 何纪言[¶5], 何理扬[‖6], 俞晨东[**7], and 张立夫[††8]

[1] 中国科学技术大学计算机科学与技术学院

January 11, 2019

[*]yuzhang@ustc.edu.cn
[†]PB15081576-蔡文韬：elisa@mail.ustc.edu.cn
[‡]PB16080210-戴路：dldaisy@mail.ustc.edu.cn
[§]PB16111545-董恒：drdh@mail.ustc.edu.cn
[¶]PB16111447-何纪言：hejiyan@mail.ustc.edu.cn
[‖]PB16110264-何理扬：heliyang@mail.ustc.edu.cn
[**]PB15000135-俞晨东：ycdxsb@mail.ustc.edu.cn
[††]PB15020718-张立夫：zlf123@mail.ustc.edu.cn

本文介绍 Julia 语言特征及其编译运行机制、扩展包及其构造机制、新兴机器学习应用需求以及现有解决对策（如 Ray [4]），探讨如何用 Julia 解决新兴机器学习应用带来的一些挑战。

Part I

Julia 语言篇

主要调研者: 张昱、蔡文韬、何理扬、俞晨东

1 Julia 概述

[2] (OOPSLA2018, §18.1) (details the design choices made by the creators of Julia and reflects on the implications of those choices for performance and usability) 详细介绍 Julia 的创作者所做出的设计选择，并反映这些选择对性能和可用性的影响。该文在 [1](SIAM2017) 的工作之上进一步细化了贯穿语言设计和编程风格之间的完整编译流水线的协同作用（by detailing the synergies at work through the entire compilation pipeline between the design and the programming style of the language）。在包含 10 个小应用的测试基准程序集PLBG上的测试结果表明，Julia 的性能是优化的 C 代码的 0.9 到 6.1 倍，Julia 在所有测试程序上都优于 JavaScript 和 Python (we give results obtained on a benchmark suite of 10 small applications where Julia performs between 0.9x and 6.1x from optimized C code. On our benchmarks, Julia outperforms JavaScript and Python in all cases) 。最后，[2] 还分析了 GitHub 上的 50 个流行项目以了解实际的库开发者使用哪些 Julia 语言特征和底层设计选择; 分析 (corpus analysis) 表明**多分派** (multiple dispatch) 和**类型标注** (type annotation) 被 Julia 程序员广泛使用, 并且被分析的库中绝大多数是**类型稳定** (type stable) 的代码 [1]。

[1] 一个方法/函数是**类型稳定**的, 是指当它被特化成一组具体类型时, 数据流分析可以将具体类型指派到该函数的所有变量

2 Julia 语言特性

Julia: A Fast Dynamic Language for Technical Computing详细介绍了 Julia 的语言特性和设计目的。不同于传统的高效编程与高性能的解决方案，Julia 的设计希望能够在一种语言上同时实现高效编程与良好的运行性能。它从底层设计，应用现代编译技术，获得了静态编译语言的性能和动态语言的高效编程。具有丰富的类型信息，为函数提供多重派发，在运行时对类型进行有效推断，并使用 LLVM 框架进行 JIT 即使编译运行。

2.1 Julia 类型

Julia 将类型作为值集合的描述，每个值都有唯一的不可变的运行时类型。对象带有类型的 tag，类型也是对象，可以在运行时创建和检查。

Julia 具有以下五种类型：

- 抽象类型：可以声明子类型和父类型（Sub <: Super）
- 复合类型：类似于 C 语言结构体，已有名字域，已声明父类型
- 比特类型：值是 bit 类型，已声明父类型
- 元组类型：不可变的有序值集合，常用于可变函数参数以及多返回值
- 类型共用体：抽象类型联合

2.2 Julia 类型参数

抽象类型、复合类型、bit 类型可以带有类型参数，用于表达类型的转换，比如 $Array\{Float64, 1\}$，同时类型参数也是有边界约束的，使用符号 <: 表示。

同时，Julia 也允许不带类型参数的类型使用，比如 $Array$ 代表可以是任何类型，$Array\{Float64\}$ 代表只类型是 Float64 的任意长数组，通过填写类型参数可以得到实例化的类型，而不是一个集合。

2.3 Julia 方法定义

在 Julia 语言中，方法的定义可以使用关键字 function 进行多行定义或者单行定义，举例如下：

```
function iszero(x::Number)
    return x==0
end

iszero(x)=(x==0)
```

- :: 用于类型断言，用于对类型进行调度约束，当类型声明缺省时，默认为 Any，:: 可以用于任何代码表达式，用于运行时类型推断。当:: 作用于变量名时，即限制了变量的类型，在赋值时会进行类型转换。注意：当没有明显类型上下文时，类型会在运行时进行推断出来。
- 对于函数，可以限制其返回值类型，例如：f(x)::Int
- 匿名方法可以这么写：x->x+1
- 全局变量声明需要使用 global 声明
- 运算符（+/-等）是简单的函数，比如 x+y 等价于 +(x,y)
- 当函数参数最后以... 结尾时，代表可变参数

2.4 Julia 参数化方法

在方法内部引用参数类型的参数通常是有用的，并指定对这些参数的约束。

例如：

```
    function assign{T<:Integer}(a::Array{T,1}, i, n::T)
```

以上表明一个元素个数为 1 的数组，元素类型是 Integer 的一种。参数化方法常用于编写应用于某集合类型的方法。

2.5 Julia 构造函数

构造函数是用于构造新复合类型对象的函数

例如以下定义的内部构造函数：

```
type Rational{T<:Integer} <: Real
  num::T
  den::T

    function Rational(num::T, den::T)
        if num == 0 && den == 0
            error("invalid␣rational:␣0//0")
        end
        g = gcd(den, num)
        new(div(num, g), div(den, g))
    end
end
```

2.6 Julia Singleton 类型

泛型函数的方法表实际上是一个字典，其中键值是类型。在调用时即需要对类型进行严谨的判断，所以引入 $SingletonKindTypeT$，类型 T 是其唯一的值。

```
    typemax(::Type{Int64}) = 9223372036854775807
```

类型在使用时是十分重要的，例如进行 I/O 操作时，我们使用 $read(file, Int32)$ 以 4 字节读取数据，并且返回 Int32 类型的值，这样的使用比使用 enum 或者其他常量更加优雅

2.7 Julia 方法排序和歧义处理

方法定义后经排序后进行调用，所以第一个匹配的方法必然是最正确的要调用的方法。所以在排序时，需要引入很多的推理逻辑，比较方法的签名是十分重要的，而参数的推理是关键。以下规则用于推断 A 是比 B 更加特殊的类型：

- 规则 1：A 是 B 的子类
- 规则 2：A 有集合 TP，B 有集合 SQ，对于一些参数值，T 是 S 的子类
- 规则 3：A 和 B 的交集是非空的，A 比 B 更具体，并且不等于 B，并且 B 并不比 A 更具体
- 规则 4：A 和 B 是元组，并且 A 以 ... 结尾
- 规则 5：A 和 B 具有兼容的参数和结构，并且 A 为 B 的参数提供一致的分配，但是反过来则不然

一些说明与解释：

- 规则 2 表明声明的子类型总是比它们声明的超类型更具体，而不考虑类型参数。
- 规则 3 对于 Union 很有效，例如当有 A 是 $Union(Int32, String)$，B 是 Number 时，A 比 B 更加特殊
- 规则 5 的一个明显表达即，$\forall T(T,T)$ 比 $\forall X,Y(X,Y)$ 更加特殊

Julia 使用多分派，所有参数都一样重要，所以歧义的存在时很有可能的，例如同时定义 $foo(x :: Number, y :: Int)$ 与 $foo(x :: Int, y :: Number)$，当调用时传入参数为 (Int, Int) 时，是不明确的。当添加一个方法时，通过寻找一对签名来检测歧义。如果存在有一个非空集合，则表明没有一个比另一个更具体。此时会抛出一个 warning

2.8 Julia 迭代循环

根据迭代的接口 $(start, done, andnext)$，一个 for 循环会转化为 while 循环，例如：

```
for i in range
    #body
end
```

会转化成

```
state = start(range)
while !done(range,state)
    (i,state) = next(range,state)
    #body
end
```

之所以选择迭代设计，是因为它不依赖于可变的堆分配状态

2.9 Julia 运算符

运算符在底层通过函数调用来实现：

a[i,j]	ref(a,i,j)
a[i,j]=x	assign(a,x,i,j)
a[i;j]	vcat(a,b)

2.10 Julia 调用 C/Fortion

提供关键字 ccall 调用本地的 C/Fortran 代码，看起来像调用函数一样简单

```
ccall(dlsym(libm,:sin),Float64,(Float64,),x)
```

2.11 Julia 并行

并行执行依赖于 Julia 标准库中的协程，语言设计通过提供对称的协同程序来支持这样的库的实现，这也可以被认为是协作调度线程，此特性允许异步通信隐藏在库中，而不要求用户设置回调。

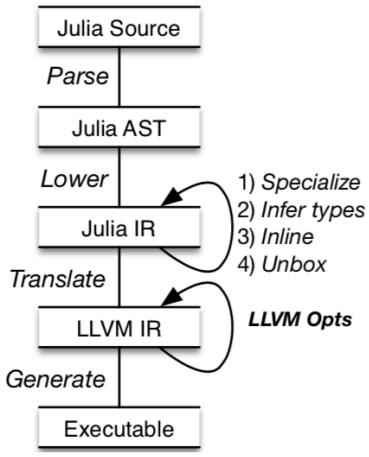
Part II

Julia 编译运行篇

主要调研者　　蔡文韬、何理扬、俞晨东

3 Julia 编译流程

Julia 的编译可以分为几个阶段，如下图所示:



(a) A given traffic flow set

图 1: 编译流程图

Julia 一个非常友好的地方在于提供了宏让我们查看各个阶段的编译结果，下面我们来看在每个阶段 Julia 编译器分别做了什么。来看下面这段代码:

```
function vsum(x)
    sum = 0
    for i = 1:length(x)
        @inbounds v = x[i]
        if !is_na(v)
            sum += v
        end
    end
    sum
end
```

首先使用 @code_lowered 宏来查看 Julia IR 中的一个阶段，它把源代码转化成类似 Python byte code 的形式，这种形式的一个特点是 single static assignment，意思是每个变量只赋值一次，并且每个变量都要先声明再使用，此外，循环和条件语句转化为 goto 和 label 的形式。

```
julia> @code_lowered vsum(1)  // like python byte code
CodeInfo(
2 1        sum = 0
3     %2  = (Main.length)(x)
      %3  = 1:%2
            #temp# = (Base.iterate)(%3)
      %5  = #temp# === nothing
      %6  = (Base.not_int)(%5)
            goto #6 if not %6
  2   %8  = #temp#
            i = (Core.getfield)(%8, 1)
      %10 = (Core.getfield)(%8, 2)
4           $(Expr(:inbounds, true))
      %12 = (Base.getindex)(x, i)
            v = %12
```

```
16            val = %12
17            $(Expr(:inbounds, :pop))
18            val
19  5     %17 = (Main.is_na)(v)
20        %18 = !%17
21              goto #4 if not %18
22  6 3         sum = sum + v
23    4         #temp# = (Base.iterate)(%3, %10)
24        %22 = #temp# === nothing
25        %23 = (Base.not_int)(%22)
26              goto #6 if not %23
27    5         goto #2
28  9 6         return sum
29  )
```

另一个是 @code_typed 宏，在这个阶段我们可以看到完成了内联和类型推断

```
julia> @code_typed vsum(1)
CodeInfo(
3 1         (Base.ifelse)(true, 1, 0)::Int64                    Colon
      %2  = (Base.slt_int)(1, 1)::Bool                           isempty
            goto #3 if not %2
  2         goto #4
  3         goto #4
  4   %6  =   (#2 => true, #3 => false)::Bool
      %7  =   (#3 => 1)::Int64
      %8  =   (#3 => 1)::Int64
      %9  = (Base.not_int)(%6)::Bool
            goto #22 if not %9
  5   %11 =   (#4 => 0, #21 => %36)::Int64
      %12 =   (#4 => %7, #21 => %42)::Int64
      %13 =   (#4 => %8, #21 => %43)::Int64
4           goto #9 if not false                                 getindex
  6   %15 = (%12 === 1)::Bool                                    ==
            goto #8 if not %15
  7         goto #9
  8   %18 = %new(Core.BoundsError)::BoundsError                  Type
            (Base.throw)(%18)::Union{}
            $(Expr(:unreachable))::Union{}
  9         goto #10
  5 10  %22 = (Main.is_na)(x)::Any
      %23 = (isa)(%22, Missing)::Bool
            goto #12 if not %23
  11        goto #15
  12  %26 = (isa)(%22, Bool)::Bool
            goto #14 if not %26
  13  %28 =   (%22, Bool)
      %29 = (Base.not_int)(%28)::Bool                            !
            goto #15
  14  %31 = !%22::Union{Missing, Bool, ##54#55{_1} where _1}
            goto #15
  15  %33 =   (#11 => $(QuoteNode(missing)), #13 => %29, #14 => %31)::Union{Missing, Bool, ##54#55{_1} where
        _1}
            goto #17 if not %33
6 16  %35 = (Base.add_int)(%11, x)::Int64                        +
  17  %36 =   (#16 => %35, #15 => %11)::Int64
      %37 = (%13 === 1)::Bool                                    ==
            goto #19 if not %37
  18        goto #20
  19  %40 = (Base.add_int)(%13, 1)::Int64                        +
            goto #20                                             iterate
  20  %42 =   (#19 => %40)::Int64
      %43 =   (#19 => %40)::Int64
```

```
      %44 =   (#18 => true, #19 => false)::Bool
      %45 = (Base.not_int)(%44)::Bool
            goto #22 if not %45
  21         goto #5
9 22  %48 =   (#20 => %36, #4 => 0)::Int64
            return %48
) => Int64
```

再使用 @code_llvm 宏查看上一层次的代码，即它的 llvm ir 代码:

```
@code_llvm vsum(1)
...... // 省略
```

最后使用 @code_native 宏查看在本地电脑中的生成的汇编结果

```
@code_native vsum(1)
...... // 省略
```

Part III

Julia 并行篇

主要调研者　　何纪言、张立夫

4 Parallel Computing

4.1 Overview

分为三个等级的并行状态：

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

4.2 Corountines

4.2.1 Tasks (aka Coroutines)

Tasks 是一个控制流特性，可以允许任务的推迟和恢复，也被称为 cooperative multitasking。在一个任务中可以随时进行暂停切换至另一个任务，看起来类似于函数调用，但是有以下两点关键区别：

1. 切换任务不需要占用空间，无论切换多少任务都不会消耗调用堆栈的空间
2. 可以以任何顺序进行任务的切换，无需在任务结束后返回至调用的任务处

4.2.2 Channel

提供了 Channel 结构进行多任务并行的实现，Channel 是一个先进先出的队列，类似于 socket 的缓冲区。最简单的例子即为生产者消费者问题

一些基本操作：

- *take!(c)*： 出队
- *put!(c, item)*： 入队
- *fetch(c)*： 读取队头数据但是不弹出
- *ChannelType(sz)*： 定义一个 Channel ，类型为 Type，缓冲区大小为 sz
- *@async*： 非阻塞宏

示例程序 *multitask.jl*:

```
const jobs = Channel{Int}(32);

const results = Channel{Tuple}(32);

function do_work()
    for job_id in jobs
        exec_time = rand()
        sleep(exec_time)                # simulates elapsed time doing actual work
                                        # typically performed externally.
        put!(results, (job_id, exec_time))
    end
end;

function make_jobs(n)
    for i in 1:n
        put!(jobs, i)
    end
end;

n = 12;

@async make_jobs(n); # feed the jobs channel with "n" jobs

@time begin
for i in 1:4 # start 4 tasks to process requests in parallel
    @async do_work()
```

```
end

while n > 0 # print out results
    job_id, exec_time = take!(results)
    println("$job_id finished in $(round(exec_time; digits=2)) seconds")
    global n = n - 1
end
end
```

4.2.3 Multi-Threading (Experimental)

仍在实验中，未来接口可能会有改变

Setup

通过环境变量规定启用线程数：

```
    export JULIA_NUM_THREADS=4
```

可在命令行窗口中确认总线程数与当前线程 id：

```
    julia> Threads.nthreads()
    4
    julia> Threads.threadid()
    1
```

@threads

通过 *Threads.@threads* 宏可以使 for 循环由多个线程进行操作：

```
    julia> Threads.@threads for i = 1:10
       a[i] = Threads.threadid()
       end
```

结果如下：

```
    julia> a
    10-element Array{Float64,1}:
     1.0
     1.0
     1.0
     2.0
     2.0
     2.0
     3.0
     3.0
     4.0
     4.0
```

Atomic Operation

Juia 提供原子操作避免出现 race condition，如 *Threads.atomic_add!()* 、*Threads.atomic_sub!()* 等，例如：

```
    julia> using Base.Threads

    julia> nthreads()
    4

    julia> acc = Ref(0)
    Base.RefValue{Int64}(0)

    julia> @threads for i in 1:1000
              acc[] += 1
           end
```

```
julia> acc[]
926

julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)

julia> @threads for i in 1:1000
           atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

@threadcall (Experimental)

4.3 Multi-Core or Distributed Processing

Julia 提供了多进程环境，可以使多个进程拥有独立分隔的内存。

Julia 分布式编程主要由以下两个原语构成：*remote references* 和 *remote calls* 。*remote reference* 是可以由任何进程使用的来引用存储在特定进程上的对象的对象。*remote call* 是由一个进程向另外一个进程发出特定的函数调用请求。

Remote references 有两种主要形式：*Future* 和 *RemoteChannel*

可以由 *julia -p N/auto* 启动多个进程，在主进程中指定 id 进行任务调度，如 *@spawnat id* 、*remotecall(f, id, args...)* ，其返回值为 *Future* 类型，通过 *fetch* 得到结果，若计算结果仍未得到则会等待至产生结果。例如：

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, nothing)

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

@spawn 也可以进行远程执行，并且会自动分配进程执行

```
julia> r = @spawn rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawn 1 .+ fetch(r)
Future(3, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

定义函数和调用模块前要加 *@everywhere* 使其全局有效，如 *@everywhere foo()* 、*@everywhere include("MyModule.jl")*

Lacks:

1. 多线程由 Base.Threads 模块进行支持，因为 Julia 仍不是完全的线程安全的，所以多线程仍在实验中。
2. 在 IØ 操作或任务切换过程中可能会出现特定的 segfaults。
3. 对于 Parallel Computing 仍有很多问题：issues

Packages:
和并行计算相关的 packages：*MPI.jl* 、*DistributedArrays.jl*

5 Concurrency

5.1 Channel

主要为 *channels.jl* 中内容
抽象类型：

```
abstract type AbstractChannel{T} end
```

Channel 结构体定义：

```
mutable struct Channel{T} <: AbstractChannel{T}
    cond_take::Condition                # waiting for data to become available
    cond_put::Condition                 # waiting for a writeable slot
    state::Symbol
    excp::Union{Exception, Nothing}         # exception to be thrown when state != :open

    data::Vector{T}
    sz_max::Int                         # maximum size of channel

    # Used when sz_max == 0, i.e., an unbuffered channel.
    waiters::Int
    takers::Vector{Task}
    putters::Vector{Task}

    function Channel{T}(sz::Float64) where T
        if sz == Inf
            Channel{T}(typemax(Int))
        else
            Channel{T}(convert(Int, sz))
        end
    end
    function Channel{T}(sz::Integer) where T
        if sz < 0
            throw(ArgumentError("Channel size must be either 0, a positive integer or Inf"))
        end
        ch = new(Condition(), Condition(), :open, nothing, Vector{T}(), sz, 0)
        if sz == 0
            ch.takers = Vector{Task}()
            ch.putters = Vector{Task}()
        end
        return ch
    end
end
```

5.1.1 成员变量

1. *cond_take::Condition* & *cond_put::Condition*： 对于从 Channel 中读写数据的等待信号；
2. *state::Symbol*： 标示当前 Channel 状态，*:open* & *:close*；
3. *excp::Union*： 标示异常；
4. *data::VectorT*： 数据内容；
5. *sz_max::Int*： Channel 大小；
6. *waiters::Int*： 等待操作的个数；
7. *takers::VectorTask*： 等待从 Channel 中获取数据的 Tasks；

8. *putters::VectorTask*：等待向 Channel 中写入数据的 Tasks。

5.1.2 特殊构造函数

```
function Channel(func::Function; ctype=Any, csize=0, taskref=nothing)
```

通过一个调用，直接将一个新的 Task 即 func 与一个新建的 Channel 进行关联，并调度 func。

```
function Channel(func::Function; ctype=Any, csize=0, taskref=nothing)
    chnl = Channel{ctype}(csize)
    task = Task(() -> func(chnl))
    bind(chnl, task)
    yield(task) # immediately start it

    isa(taskref, Ref{Task}) && (taskref[] = task)
    return chnl
end
```

5.1.3 其他相关函数

put!(c::Channel, v)

向 Channel 中放入一个数据 v，首先检查 Channel 是否为开启状态，之后判断该 Channel 是否具有 buffer：

```
isbuffered(c) ? put_buffered(c,v) : put_unbuffered(c,v)
```

对于 *put_buffered(c, v)*：

```
function put_buffered(c::Channel, v)
    while length(c.data) == c.sz_max
        wait(c.cond_put)  # wait for notify on a condition
    end
    push!(c.data, v)

    # notify all, since some of the waiters may be on a "fetch" call.
    notify(c.cond_take, nothing, true, false)
    v
end
```

其中：*notify(condition, val=nothing; all=true, error=false)*

对于 *put_unbuffered(c, v)*：

```
function put_unbuffered(c::Channel, v)
    if length(c.takers) == 0
        push!(c.putters, current_task())
        c.waiters > 0 && notify(c.cond_take, nothing, false, false)

        try
            wait()
        catch ex
            filter!(x->x!=current_task(), c.putters)
            rethrow(ex)
        end
    end
    taker = popfirst!(c.takers)
    yield(taker, v)

    return v
end
```

如果没有在等待的 *c.takers* 将当前任务放入 *c.putters* 并等待，捕获到 ex 会将当前任务弹出，更新 *c.putters*
。

take!(c::Channel)

和 put! 同样先进行 Channel 的类型判断，然后执行对应的 take 操作：

```
take!(c::Channel) = isbuffered(c) ? take_buffered(c) : take_unbuffered(c)
function take_buffered(c::Channel)
    wait(c)
    v = popfirst!(c.data)
    notify(c.cond_put, nothing, false, false) # notify only one, since only one slot has become
    available for a put!.
    v
end

popfirst!(c::Channel) = take!(c)

# 0-size channel
function take_unbuffered(c::Channel{T}) where T
    check_channel_state(c)
    push!(c.takers, current_task())
    try
        if length(c.putters) > 0
            let refputter = Ref(popfirst!(c.putters))
                return Base.try_yieldto(refputter) do putter
                    # if we fail to start putter, put it back in the queue
                    putter === current_task || pushfirst!(c.putters, putter)
                end::T
            end
        else
            return wait()::T
        end
    catch ex
        filter!(x->x!=current_task(), c.takers)
        rethrow(ex)
    end
end
```

fetch!(c::Channel)

fetch 只支持对具有 buffer 的 Channel 操作：

```
function fetch_buffered(c::Channel)
    wait(c)
    c.data[1]
end
```

close(c::Channel)

关闭一个 Channel。

isopen(c::Channel) = (c.state == :open)

判断 Channel 是否处于 open 状态。

bind(chnl::Channel, task::Task)

将一个 Channel 和一个 task 关联起来，当 task 完成时 Channel 会自动关闭。一个 task 可以关联多个 Channel，多个 task 也可以关联一个 Channel，当第一个 task 终结时会关闭 Channel。

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

```

```
julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

channeled_tasks(n::Int, funcs...; ctypes=fill(Any,n), csizes=fill(0,n))

一次调用创建 n 个 Channel，和 length(funcs) 个 task，并将每一个 task 和每一个 Channel 进行关联，之后进行调度。

```
foreach(t -> foreach(c -> bind(c, t), chnls), tasks)
foreach(schedule, tasks)
yield()
```

5.2 Julia Channel & Task 自顶向下分析

5.2.1 简介

本文是对 Julia 中 Channel 和 Task 实现的技术细节的一个自顶向下分析，并解释了 Julia 事件通知模型在不同操作系统的底层技术实现，下面大致分为几个层次分析，每一层的实现几乎都是依赖于下一层提供的模型：

- Channel (base/channel.jl)
- Event (base/event.jl)
- libuv.jl (base/libuv.jl)
- jl_uv.c (src/jl_uv.c)
- libuv (extern libraray)

此外附录中有对 libuv 的一些简单介绍和例子，最后附有参考资料。

5.2.2 Channel (base/channel.jl)

首先是 Julia 中 Channel 提供的主要接口有 *put!()* 和 *take!()*。

忽略各种 multi-dispatch 和一些特殊情况，这两个函数的主要逻辑是调用 *wait* 和 *notify* 两个接口实现的：

```
# 生产者调用
function put_buffered(c::Channel, v)
    # Chanel 满了，阻塞，直到有可写的位置
    while length(c.data) == c.sz_max
        wait(c.cond_put)
    end
    # 添加一个数据
    push!(c.data, v)

    # notify all, since some of the waiters may be on a "fetch" call.
    # 通知消费者，可以拿数据了
    notify(c.cond_take, nothing, true, false)
    v
end

# 消费者调用
function take_buffered(c::Channel)
    # 阻塞，直到有数据可读
    wait(c)

```

```
        # 注意! 这个 wait 不是按条件 wait

        # 取出一个数据
        v = popfirst!(c.data)
        # 通知想放置数据的生产者，有位置可写了
        notify(c.cond_put, nothing, false, false) # notify only one, since only one slot has become
        available for a put!.
        v
    end
```

值得一提的是，*take* 中使用的 *wait* 并不是 *event* 标准库中提供的 *wait*，而是：

```
    wait(c::Channel) = isbuffered(c) ? wait_impl(c) : wait_unbuffered(c)
    function wait_impl(c::Channel)
        while !isready(c)
            check_channel_state(c) # 检验 c.state 状态
            wait(c.cond_take) # 等生产者放数据
        end
        nothing
    end
```

总结：

这一层依靠 Event 提供的事件阻塞和通知模型，提供了一个多任务可读可写的 Channel。

5.2.3 Event (base/event.jl)

由源代码可以看出，wait 和 notify 两个接口为我们提供了一个阻塞和唤醒的模型。

其中：

notify 这个函数的作用是唤醒一些等待某些条件的 Task。

```
    notify(condition, val=nothing; all=true, error=false)

    function notify(c::Condition, arg, all, error)
        cnt = 0
        if all
            # 通知排队队列中的所有
            cnt = length(c.waitq)
            for t in c.waitq
                # 安排任务执行
                error ? schedule(t, arg, error=error) : schedule(t, arg)
            end
            # 清空排队队列
            empty!(c.waitq)
        elseif !isempty(c.waitq)
            # 只通知排队队首的那一个
            cnt = 1
            t = popfirst!(c.waitq)
            # 安排这个任务执行
            error ? schedule(t, arg, error=error) : schedule(t, arg)
        end
        cnt
    end
```

wait 这个函数的作用是阻塞某些 Task，直到某些事件发生。

```
    wait([x])

    function wait(c::Condition)
        ct = current_task()

        # 将当前任务加入排队队列
```

```
        push!(c.waitq, ct)

        try
            # 等待
            return wait() # 注意这里不是调用自身，是下面的函数
        catch
            filter!(x->x!==ct, c.waitq)
            rethrow()
        end
    end

    function wait()
        while true
            if isempty(Workqueue)
                # 工作队列是空的
                c = process_events(true)
                if c == 0 && eventloop() != C_NULL && isempty(Workqueue)
                    # 没有活跃的 handles，等待信号
                    pause()
                    # 注意：pause() = ccall(:pause, Cvoid, ())
                end
            else
                # 取得一个任务
                reftask = poptask()
                if reftask !== nothing
                    result = try_yieldto(ensure_rescheduled, reftask)
                    process_events(false)
                    # return when we come out of the queue
                    return result
                end
            end
        end
        # unreachable
    end
```

这里的 *schedule* 本质上是包装 *eventloop* 的一个函数：

```
    schedule(t::Task) = enq_work(t)
    function enq_work(t::Task)
        t.state == :runnable || error("schedule: Task not runnable")
        ccall(:uv_stop, Cvoid, (Ptr{Cvoid},), eventloop())
        # 将任务放入工作队列，并且更改更改任务的状态：正在排队
        push!(Workqueue, t)
        t.state = :queued
        return t
    end
```

schedule 实现的功能是：将任务添加到工作队列异步执行（排队），不阻塞，实现所谓的并发，即“计划执行”的意思，并不阻塞住立刻生效。

总结：这一层利用 *uv_stop* 、*process_events* 以及 *eventloop* 实现了任务的事件阻塞和通知模型。

（PS. 这里 *uv_stop* 跨层调用了，我的看法是应该在 *libuv.jl* 中添加一层封装更优雅。）

5.2.4 libuv.jl (base/libuv.jl)

eventloop 这个函数只是对外部函数调用的一个简单封装：

```
    eventloop() = uv_eventloop::Ptr{Cvoid}

    global uv_eventloop = ccall(:jl_global_event_loop, Ptr{Cvoid}, ())
```

process_events 同理：

```
function process_events(block::Bool)
    loop = eventloop()
    if block
        return ccall(:jl_run_once,Int32,(Ptr{Cvoid},),loop)
    else
        return ccall(:jl_process_events,Int32,(Ptr{Cvoid},),loop)
    end
end
```

总结：这一层作为 Julia 内部（语言层面的函数）和外部（libuv）的连接，对 libuv 的接口进行了封装。

5.2.5 jl_uv.c (src/jl_uv.c)

在这一层中我们关注 libuv 究竟为 Julia 提供了怎样的接口：
首先 *jl_global_event_loop* 是一个返回 *jl_io_loop* 的函数：

```
JL_DLLEXPORT uv_loop_t *jl_global_event_loop(void)
{
    return jl_io_loop;
}
```

这里的 *jl_io_loop* 其实就是 *uv_default_loop()* （见：链接 ），是 libuv 提供的默认 loop。
这个函数是 *process_events(block=false)* 时实际调用的函数，使用的 *uv_run* 参数是 *UV_RUN_NOWAIT*。

```
JL_DLLEXPORT int jl_process_events(uv_loop_t *loop)
{
    jl_ptls_t ptls = jl_get_ptls_states(); // 还没太搞懂这是什么，和 LLVM，GC 有关
    if (loop) {
        loop->stop_flag = 0;
        jl_gc_safepoint_(ptls);
        // 调用 uv_run
        return uv_run(loop,UV_RUN_NOWAIT); // UV_RUN_NOWAIT
    }
    else return 0;
}
```

这个函数是 *process_events(block=true)* 时实际调用的函数，使用的 *uv_run* 参数是 *UV_RUN_ONCE*。

```
JL_DLLEXPORT int jl_run_once(uv_loop_t *loop)
{
    jl_ptls_t ptls = jl_get_ptls_states();
    if (loop) {
        loop->stop_flag = 0;
        jl_gc_safepoint_(ptls);
        // 调用 uv_run
        return uv_run(loop,UV_RUN_ONCE); // UV_RUN_ONCE
    }
    else return 0;
}
```

总结：这一层主要是调用外部库 libuv 中的 *uv_run*，当然别忘了还有在 schedule 中调用的 *uv_stop* 。

5.2.6 libuv (extern libraray)

libuv 采用事件循环的方式来完成各种异步操作，例如发射一个 I/O 请求，然后等待返回数据这一行为，就很适合异步完成，在数据返回前可以去做别的任务，程序由阻塞变为非阻塞会大大提供效率。

libuv 在不同功能操作系统上使用不同的高并发异步模型：

- Linux: epoll
- FreeBSD: kqueue

- Windows: iocp

libuv 可以用于很多地方，Julia 使用的是 *uv_async_init* 这种 handle，实现线程间各种信号的通信。

5.3 Thread

Julia 中对于 thread 的定义主要在 *threading.h* 和 *threadgroup.h* 中，对于 thread 的操作主要在 *threading.c* 中。

首先 Julia 是根据当前操作系统去判断使用的 thread 操作的 API，对于 Windows，Julia 采用的主要是微软自家的线程操作，调用的是头文件 *Windows.h* ；对于非 Windows 系统，Julia 采用的是 pthreads 库进行线程的操作。

5.3.1 Thread-local storage (TLS)

在 threading.c 中定义了一系列的 thread 操作，如初始化 thread，执行分配到 thread 上的函数等操作，Julia 在对线程进行这些操作时会用到 tls_states buffer (Thread-local storage，Julia 的优化定义在 *src/tls.h* 中)，值得注意的是 Julia 对 tls 的使用也做了一些优化：

首先是因为 Mac 和 Windows 不是用 ELF，所以采用运行时的 API 来创建 TLS，据他们说这种方式要比直接使用 *___thread* 关键字的效率更高一些。

- 对于 Mac：因为没有静态 TLS 模型，所以采用的是调用 pthread 库中的 API 来进行创建；
- 对于 Windows：则是使用微软自家的 *TLSAlloc* 实现；
- 对于 Linux，FreeBSD：使用 *___thread* 关键字进行实现；

Julia 中具体的 TLS 结构在 *src/julia_threads.h* 中定义为 *_jl_tls_states_t* 。

5.3.2 Thread init

对于 master thread 的初始化，Julia 同样根据操作系统进行了区分，对于 Windows 采用了 *DuplicateHandle* 进行处理，而其他的操作系统，则由 *ti_initthread(0)* 进行。

对于其他线程，均是由 *static void ti_initthread(int16_t tid)* 进行初始化，在该函数里面对目标线程的 tls_states 进行了各项初始化：

```
static void ti_initthread(int16_t tid)
{
    jl_ptls_t ptls = jl_get_ptls_states();
#ifndef _OS_WINDOWS_
    ptls->system_id = pthread_self();
#endif
    assert(ptls->world_age == 0);
    ptls->world_age = 1; // OK to run Julia code on this thread
    ptls->tid = tid;
    ptls->pgcstack = NULL;
    ptls->gc_state = 0; // GC unsafe
    // Conditionally initialize the safepoint address. See comment in
    // safepoint.c
    if (tid == 0) {
        ptls->safepoint = (size_t*)(jl_safepoint_pages + jl_page_size);
    }
    else {
        ptls->safepoint = (size_t*)(jl_safepoint_pages + jl_page_size * 2 +
                                    sizeof(size_t));
    }
    ptls->defer_signal = 0;
    void *bt_data = malloc(sizeof(uintptr_t) * (JL_MAX_BT_SIZE + 1));
    if (bt_data == NULL) {
        jl_printf(JL_STDERR, "could␣not␣allocate␣backtrace␣buffer\n");
        gc_debug_critical_error();
        abort();
```

```
    }
    memset(bt_data, 0, sizeof(uintptr_t) * (JL_MAX_BT_SIZE + 1));
    ptls->bt_data = (uintptr_t*)bt_data;
    ptls->sig_exception = NULL;
    ptls->previous_exception = NULL;
#ifdef _OS_WINDOWS_
    ptls->needs_resetstkoflw = 0;
#endif
    jl_init_thread_heap(ptls);
    jl_install_thread_signal_handler(ptls);

    jl_all_tls_states[tid] = ptls;
}
```

5.3.3 Thread function

在 *threading.c* 中还定义了处理分配到 thread 上函数的操作，主要定义在函数 *void ti_threadfun(void *arg)* 中，这里会首先创建一个新的 thread，然后对该线程进行一个栈空间的初始化，这个栈空间的初始化同样根据不同的操作系统采用的不同的栈分配 API（非 Windows 采用的 pthread），之后则是在分配好的栈空间上创建一个根任务，然后将该 thread 添加到 threadgroup 中初始化，从 threadgroup 的结构体中可以看到采用 libuv 库中的一些接口：

```
typedef struct {
    int16_t *tid_map, num_threads, added_threads;
    uint8_t num_sockets, num_cores, num_threads_per_core;

    // fork/join/barrier
    uint8_t group_sense; // Written only by master thread
    ti_thread_sense_t **thread_sense;
    void              *envelope;

    // to let threads sleep
    uv_mutex_t alarm_lock;
    uv_cond_t  alarm;
    uint64_t   sleep_threshold;
} ti_threadgroup_t;
```

之后则会进入一个 *for(::)* 循环来执行调用的任务，是对分配给当前线程上的任务进行处理，判断结构体 *ti_threadwork_t work* 是否为空，根据 work 中的成员进行相关函数的的调用，以及根据成员状态判断该任务是否完成执行跳出循环。

```
typedef struct {
    uint8_t command;
    jl_method_instance_t *mfunc;
    jl_callptr_t fptr;
    jl_value_t **args;
    uint32_t nargs;
    jl_value_t *ret;
    size_t world_age;
} ti_threadwork_t;
```

Part IV

Julia Packages 篇

主要调研者　　戴路、何理扬

6 Math Library

作为一种科学计算语言，Julia 拥有丰富的数学库，比如 DataFrames.jl，在 Python 语言中，有一个专门用于处理行列式或表格式（tabular）数据的结构，名为 DataFrame，是科学计算库 Pandas 的重要组成部分，已成为很多 Python 第三方包支持的基本数据操作框架。类似地，Julia 语言的 DataFrames 库，正是意图建立这样一个基础的数据结构:

```
julia> df = DataFrame(A = 1:4, B = ["M", "F", "F", "M"])
4×2 DataFrames.DataFrame

 Row   A   B

 1     1   M
 2     2   F
 3     3   F
 4     4   M
```

又比如 StatsFuns.jl，一个统计相关的数学函数库，举一个使用二项分布概率密度函数的例子:

```
help?> binompdf(1,2,3)
No documentation found.

StatsFuns.RFunctions.binompdf is a Function.

# 2 methods for generic function "binompdf":
[1] binompdf(n::Real, p::Real, x::Union{Float64, Int64}) in StatsFuns.RFunctions at /Users/heliyang/.julia/
       packages/StatsFuns/0W2sM/src/rmath.jl:55
[2] binompdf(n::Real, p::Real, k::Real) in StatsFuns at /Users/heliyang/.julia/packages/StatsFuns/0W2sM/src/
       distrs/binom.jl:16

julia> binompdf(10,0.5,3)
0.11718750000000014
```

此外还有例如 Lora.jl：蒙特卡罗方法、PDMats.jl：正定矩阵各种结构统一接口、ConjugatePriors.jl：共轭先验分布的 Julia 支持包、GLM.jl：广义线性模型、Distances.jl：向量之间距离的评估包等丰富的数学库，更多的可以参考:Julia Math Libraries

7 JuliaGPU

7.1 多 GPU 编程 (Multi-GPU Programming)

CUDA 主要提供如下接口来实现对多 GPU 的编程

- cudaSetDevice(): command sets the context by which the commands are issues to a specific GPU
- cudaDeviceSynchronize(): waits until all preceding commands in all streams of all host threads have completed.
- cudaGetDeviceCount(): Command is designed to determinewhether multi-GPU usage is possible.

一个简单的例子：//设置设备号 cudaSetDevice(0) float *p0 //分配空间 cudaAlloc(&p0, size) //运行设备 0 mykernel«<grid, block»>(p0); //将内容从设备 0 复制到设备 1 cudaMemcpyPeer(p1,1,p0,0,0,size); //运行设备 1 mykernel«<grid, block»>(p1);

从而实现了多个设备中的共享内存

7.2 julia 对 GPU 编程的支持

CUDAnative, CudaDrv 等可以初始化一个或多个 GPU，用于计算，并释放用于其他用途。由于与 Julia 的垃圾收集交互，在此过程中存在一些可能的问题 - 在一个“会话”中分配的 CUDA 数组对象不应该如果您关

闭设备然后打开一个新的“会话”，则可以使用。

julia 调用 GPU 的基本代码思路如下：result = devices(dev->true) do devlist # Code that does GPU computations end

这些支持主要通过 CUDAdrv 和 CUDAnative 等库实现

CUDAdrv

该库封装了 CUDA 驱动程序 API。它提供与驱动程序 API 相同的粒度级别，但因为供 julia 高级语言调用，程序员的代码效率得以提高。使用 Julia 对 GPU 编程时将调用此库。在多个设备的协调方面，其实现的功能为设备的管理，context 的管理，事件的管理等：

- Device Management CUDAdrv.CuDevice CUDAdrv.devices CUDAdrv.name(::CuDevice) CUDAdrv.totalmem(::CuDev CUDAdrv.attribute CUDAdrv.capability(::CuDevice) CUDAdrv.warpsize(::CuDevice)
- Context Management CUDAdrv.CuContext CUDAdrv.destroy!(::CuContext) CUDAdrv.CuCurrentContext CUDAdrv.activate(::CuContext) CUDAdrv.synchronize(::CuContext) CUDAdrv.device(::CuContext)
- Primary Context Management CUDAdrv.CuPrimaryContext CUDAdrv.CuContext(::CuPrimaryContext) CUDAdrv.isactive(::CuPrimaryContext) CUDAdrv.flags(::CuPrimaryContext) CUDAdrv.setflags!(::CuPrimaryContext, ::CUDAdrv.CUctxflags) CUDAdrv.unsafereset!(::CuPrimaryContext, ::Bool)
- Device Management CUDAdrv.CuEvent CUDAdrv.record CUDAdrv.synchronize(::CuEvent) CUDAdrv.elapsed CUDAdrv.@elapsed

CUDAnative.jl 其实现了 juliaCompiler 和 CUDA 的交互

实现原理图如下：

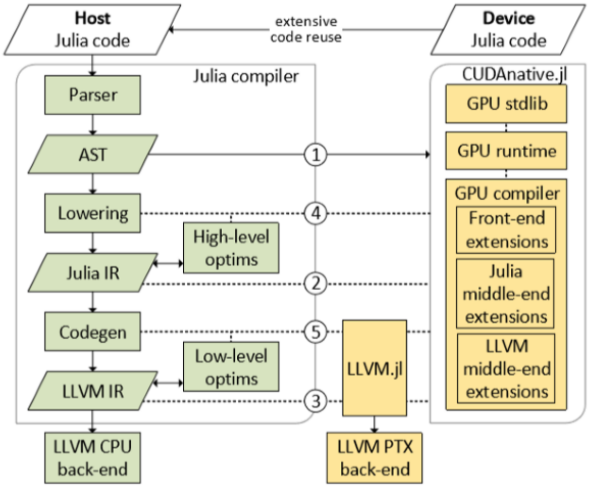


图 2: julia 和 GPU 编译器的交互

其支持的 julia 语言的部分如下：

- Indexing: threadIdx().x,y,z, blockDim(), blockIdx(), gridDim(), warpsize()
- Shared memory: @cuStaticSharedMemory, @cuDynamicSharedMemory
- Array type: CuDeviceArray (converted from input CuArrays, or shared memory)
- I/O: @cuprintf
- Synchronization: syncthreads
- Communication: voteall,any,ballot
- Data movement: shflup,down,bfly,idx

不过，一些 julia 的语言特性目前还不支持被编译到 GPU 上（会报错），比如 julia 运行时，垃圾收集等

8 Flux

机器学习对于语言提出的要求

- 数值计算
- 微分计算（自动微分，Automatic Differentiation）
- 并行计算（比如在语言层面，定义新的类型系统，支持 GPU 上阵列，矩阵维度的数据类型）
- 概率计算

julia 语言适用于数值计算，并行计算等，且对微分计算提供了编译层面的思路（见后文 zygote). 为了在机器学习中充分发挥 julia 的高级语言特性，flux 对机器学习方法进行了更进一步的抽象。其设计思想主要为：

- Doing the obvious thing: 用概念封装函数, 掩盖数学细节例如：求梯度 (gradient), 正则化 (regularization) 等基本操作，均被封装为函数
- You could have written Flux: Flux 的代码十分简明直接。因此有问题的时候完全可以查看 julia 源码
- Play nicely with others: Flux 与 julia 其他的科学计算库（dataframe, differential equation solvers）兼容性良好。因此在整个数据处理过程中，可以灵活调用 flux 的模块

一个例子：使用 flux 进行线性回归

1. **loss function**.

```
function loss(x, y)
   ŷ = predict(x)
    sum($(y - y)^2$)
   end

```

2. **计算梯度**

```
   using Flux.Tracker
   W = param(W)//告诉 flux W 是参数而不是变量
   b = param(b)
   gs = Tracker.gradient(() -> loss(x, y), Params([W, b]))

```

3. **训练**

```
   using Flux.Tracker: update!
   Δ = gs[W] \# Update the parameter and reset the gradient
   update!(W, -0.1Δ)//其实是应用了 W = W + Δ

```

flux 其他实现的模块还有：CNN（接口：CNN 的层，激活函数等等），RNN ，优化器 (optimizer) ，独热码编码（Ont-Hot Encoding）。

总结：Flux 的模块化比较灵活，类似于 matlab, 继承了 julia 对与数值计算的精神。同时，由于 Julia 的语言特性，其速度也快于基于 python 的 sklearn 等库然而，由于该库较新，常常有问题暴露，性能也不稳定。此外，其不支持多 GPU 编程。该库还有待未来发展

9 Zygote

9.1 Julia 的 SSA IR

从 Julia 0.7 开始，编译器的一部分使用 SSA 中间表示。以往，编译器直接从较低形式的 Julia AST 生成 LLVM IR，其删除了大多数语法抽象，但仍然看起来很像 AST。之后，为了便于优化，将 SSA 引入该 IR 并且 IR 被线性化（即函数参数可以仅是 SSA 值或常数的形式）。然而，由于 IR 中缺少 Phi 节点，非 ssa 值（时隙）保留在 IR 中，结果 SSA 表示的大部分功能没有得到充分发挥。Julia SSA IR 则引入了新的 Phi 节点的表示方法。

SSA 的主要设计为：数据结构的核心是一个 statement 向量。每个 statement 基于其在向量中的位置被隐式地分配 SSA 值（即，可以使用 SSAValue（1）等来访问 1 处的语句的结果）。对于每个 SSA 值，还记录其类型。由于 SSA 值仅在定义时分配一次，该类型也是相应索引处表达式的结果类型。然而，虽然这种表示相当有效（因为分配不需要明确地编码），但是重新排序和插入改变了语句号，因此可能在某些场景有缺点。另外，Julia SSA IR 不保留使用列表。相对的，Julia SSA 保留一个单独的节点缓冲区来插入（包括插入它们的位置，相应值的类型和节点本身）。这些节点按它们在插入缓冲区中的出现进行编号，允许它们的值立即在 IR 中使用。该方案的巧妙之处在于，这种压缩可以作为后续传递的一部分。大多数优化过程需要遍历整个语句列表，并在此过程中执行分析或修改。JULIA SSA IR 提供了一个 IncrementalCompact 迭代器，可用于迭代语句列表。它将执行任何必要的压缩，并返回节点的新索引以及节点本身。这种安排的动机是：由于优化过程无论如何都需要用到相应的存储器，并且导致相应的存储器访问损失，执行额外的内存处理应该具有相对较小的开销（并且节省在 IR 修改期间维护这些数据结构的开销）。

9.2 zygote 的优势

观察如下代码，其功能为对 f(x) 求导：

```
julia> using Zygote

julia> f(x) = 5x + 3

julia> f(10), f'(10)
(53,5)

julia> @code_llvm f'(10)
define i64 @"julia#625_38792"(i64) {
top:
  ret i64 5
}
```

通过查看其 LLVM 代码，可以发现其直接返回了导数 5，可见 zygote 在编译的时候就能够算出微分。以往的自动微分常常用计算图实现，但 zygote 利用 SSA 和计算图的相似性，在编译器 IR 的层面中进行处理，使得程序能直接通过 SSA IR 计算微分，避免构建计算图

Part V

Ray 及强化学习篇

主要调研者　　董恒

10 Ray Overview

10.1 总述

ray 的出现是为了解决 AI 应用创建过程中对框架灵活性和高效性的新要求。
为了达到这个要求，Ray 采用了分布式的调度器和分布式的可容错的储存去管理系统的控制状态。

10.2 目的

10.2.1 RL 的出现

构建深度神经网络的复杂性，导致需要专门的框架来减小难度。这部分已经有了很多的系统，比如 PyTorch, TensorFlow 等等。但是更加广阔的应用需要我们与环境进行交互，而不单单是监督学习。这就需要强化学习。但是针对 RL 的系统目前达不到要求。

10.2.2 RL 的基本组成

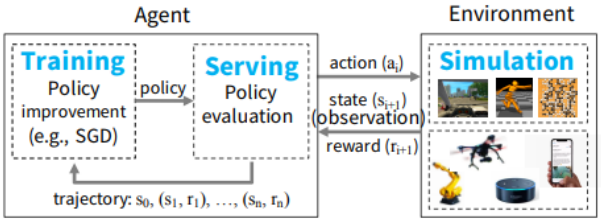


图 3: RL 的基本组成

1. **Simulation**
   探索不同的 action 之下的结果，由此对 policy 进行评估
2. **training**
   由上面的得到的类似路径的东西来训练，提高 policy 的能力
3. **serving**
   由 policy 以及当前的环境状态，判断采取何种 action

10.2.3 对系统的需求

1. **fine-grained computations**
   比如，action 需要在很短的时间内做出，需要同时做大量的 simulation
2. **heterogeneity both in time and in source usage**
   比如一个 simulation 可能需要几毫秒，也有可能几个小时
   或者 traing 需要 GPU 而 simulation 需要 CPU 资源等等
3. **dynamic execution**
   这是由于 simulation 或者与环境的交互可能会改变未来的状态

10.2.4 ray 的出现

现在存在的系统没有能完全达到上述要求的

10.3 Ray 的设计

10.3.1 总述

通用性 为了实现通用集群架构，同时满足 simulation, traing, serving 要求。

1. task-parallel
2. actor-based

性能 将以前中心化的东西分布化

1. task scheduler
2. metadata

以及容错机制

10.3.2 模型

Name	Description
$futures = \textbf{f.remote}(args)$	Execute function f remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
$objects = \textbf{ray.get}(futures)$	Return the values associated with one or more futures. This is blocking.
$ready\_futures = \textbf{ray.wait}(futures, k, timeout)$	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
$actor = \textbf{Class.remote}(args)$ $futures = actor.\textbf{method.remote}(args)$	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

图 4: Ray 编程模型

编程模型 **task**：stateless

actor：stateful. 需要串行执行

计算模型 使用计算图，表达依赖关系

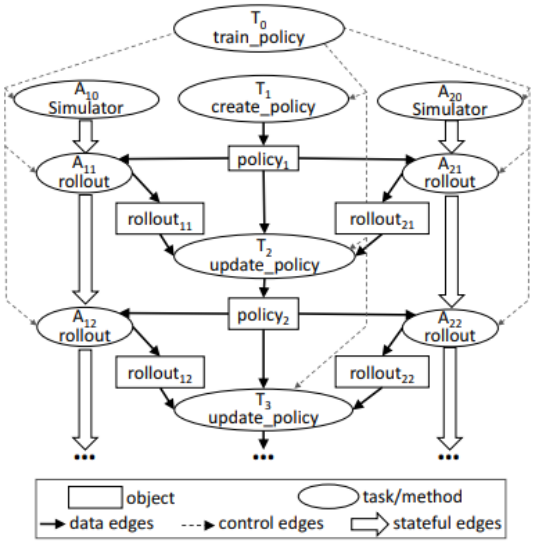


图 5: 计算图示例

比如数据依赖，串行执行的次序

10.3.3 架构

应用层

1. driver 执行用户程序的进程

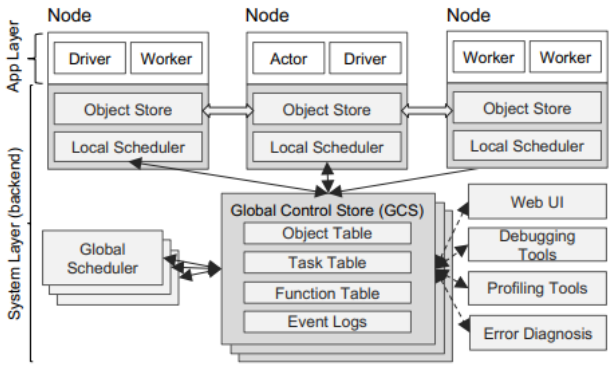


图 6: Ray 架构

2. worker 无状态的进程，执行 tasks
3. actor 有状态的进程，依次 invoke method

系统层

Global Control Store(GCS) 目的是容错性与低延迟。同时也可以将 task 与 task scheduling 分离
概括来说，是有 GCS 的存在，系统的每个部分都可以是无状态的。这样就会简化系统的设计

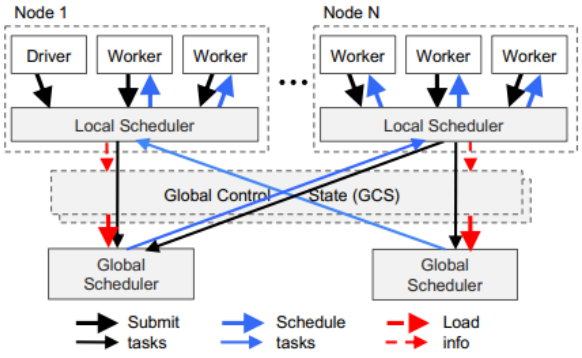


图 7: Ray 调度方式

Bottom-Up distributed Scheduler 有两个层次的调度器，global 与 local，在某个节点上创建的任务，先交给本地的调度器，如果分配不了，比如延迟太大或者资源不够，就会传达给全局的调度器。
这样不仅减小的全局调度器的负载，也减小了工作时延

In-Memory Distributed Object Store 同一个节点可能共享内存。
如果不在同一个节点，就会复制到执行的那个节点

实现 已经可以通过 pip 来安装
大概有 4 万行代码，72% 的 C++ 实现系统层，28% 的 python 代码实现应用层

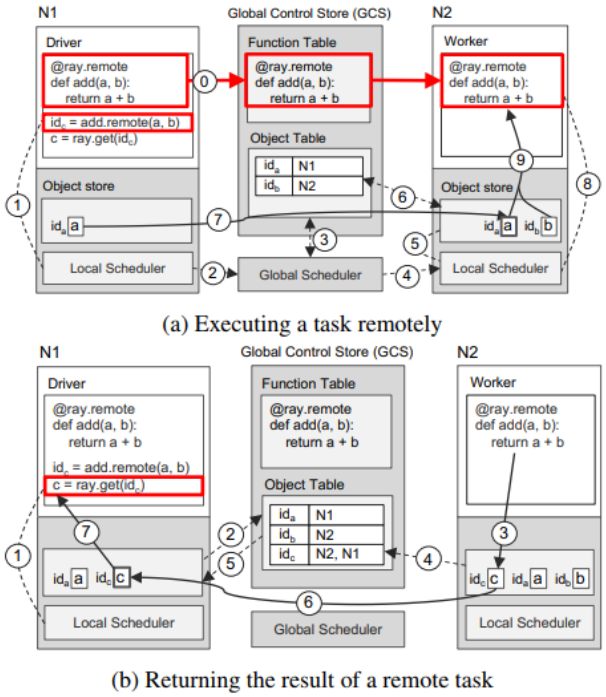


图 8: 执行流程示意图

10.3.4 执行流程

11 RLlib Overview

11.1 Reference

Markov Decision Process
Q Learning
Playing Atari with Deep Reinforcement Learning
Trust Region Policy Optimization
Proximal Policy Optimization Algorithms

11.2 RLlib 设计

11.2.1 目的

现代的 RL 算法高度 irregular

- 每个 task 的时长与资源需求
- 交流的模式变化多端
- 计算是有可能嵌套在一起的
- 需要维护和更新很多的数据量

现在的计算架构很多都做不到这一点。

11.2.2 RL 的控制组块

Logically centralized control for distributed RL

(c) 中

- D: driver program

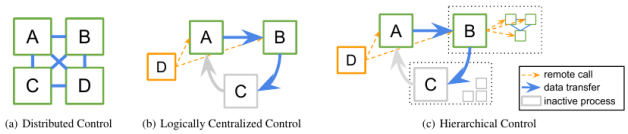


图 9:

- A,B: Worker(Active)
- C: Worker(Inacive)
- B 方框：nested computation --> sub-workers 由 B 来指定的子任务

这种框架的优势是

1. 相应的算法更加容易设计（相对于完全分布式的）
2. 子任务的分离增加代码的可重用性
3. 任务之间可以彼此嵌套

11.2.3 使用 ray 来实现原因

1. Ray: Distributed Scheduler --> 天然地适合 hierarchical control model. 在 ray 里面实现嵌套的计算不会出现中心化的任务分配瓶颈
2. Ray: Actor -->hierarchical delegation. 可以创建新进程和任务，也可以调用其他的 actor

11.2.4 概要

- 实现的内容很复杂，并没有完全弄清楚。有些最近出现的 RL 算法我还没有了解
- 参考内容 RLlib Document & RLlib Paper

综合来说：

1. 其实现依赖于 ray 的功能的实现

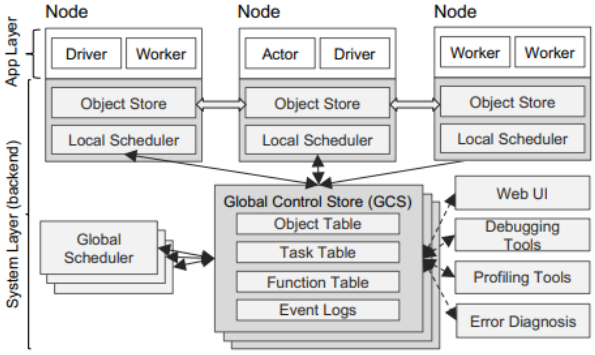


图 10:

即需要上面的 `task` 与 `actor` 抽象实现以及 `Local Scheduler` 与 `Global Scheduler` 的实现。

1. 需要 `tune` 库的支持作为深度学习的超参调整。所有的 RLlib agent 都与 Tune API 兼容。

11.2.5 High Level

Agent 包含一系列著名的算法和能够帮助计算新东西的扩展原语

在一个比较高的层次上，RLlib 提供了一个 agent 类，内部存有 policy, 便于与环境交互。

通过 agent 接口，policy 能够被训练，暂存，或者能够计算下一步动作。

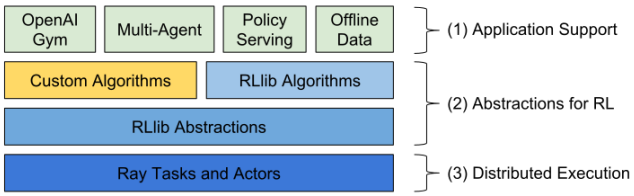


图 11:

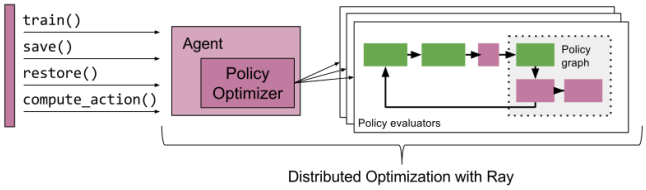


图 12:

训练一个 agent 需要指定环境 (任何 OpenAI gym 环境以及用户自定义的)，指定特定的算法 (`PPO`, `PG`, `A2C`, `A3C`, `IMPALA`, `ES`, `DDPG`, `DQN`, `APEX`, and `APEX_DDPG`). 也可以改变默认设定 (比如算法参数，资源参数等等)

算法

- Gradient-based
  - Advantage Actor-Critic(A2C, A3C)
    [paper] [implementation]
  - Deep Deterministic Policy Gradients (DDPG)
    [paper] [implementation]
  - Deep Q Networks (DQN, Rainbow)
    [paper] [implementation]
  - Policy Gradients
    [paper] [implementation]
  - Proximal Policy Optimization (PPO)
    [paper] [implementation]
- Derivative-free
  - Augmented Random Search (ARS)
    [paper] [implementation]
  - Evolution Strategies
    [paper] [implementation]

11.2.6 RLlib Abstraction

RLlib 主要概念

Policy Graph

概念 Policy graph 类封装了 RL 算法的核心数据组成。通常包括

- Policy model: 决定 action
- Trajectory postprocessor: 处理路径以获取经验
- Loss function: 右上面给的 postprocessed experiences 来改进 policy

为了支持多种多样的框架，绝大部分与深度学习框架交互的部分被独立于PolicyGraph interface. 另外，为了简化 policy graph 的定义，还包含了两个模板Tensorflow and PyTorch-specific

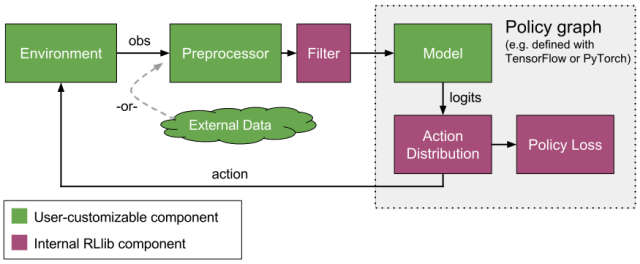


图 13:

上图是不同组件之间的数据流动概念图。

- Environment: 给一个 action 产生一个 observation
- Preprocessor & Filter: 预处理 observation
- Model: 神经网络，内部提供的网络包括：vision network(图像), fully connected network, LSTM
- ActionDistribution: 决定下一个动作

具体 用户指定一个 policy mode π, 将当前的 observation o_t 和 (optional) RNN 隐藏状态 h_t 映射到一个 action a_t 以及下一个 RNN 状态 h_{t+1} 也可以是用户定义的值 y_t^i 比如说 value predictions, TD error

$$\pi(o_t, h_t) \Rightarrow (a_t, h_{t+1}, y_t^1 ... y_t^N)$$

大部分算法都会定义一个 trajectory postprocessor ρ, 目的是转换一系列的 K 元组 $X_{t,K}$ $(o_t, h_t, a_t, h_{t+1}, y_t^1 ... y_t^N, r_t, o_{t+1})$, 其中，$r_t$ 为 reward, o_{t+1} 为在采取 action 之后新的 observation. 另外，为了支持 multi-agent 环境，从其他的 P 个 agents 的 experience batches $X_{t,K}^p$ 也可以获得

$$\rho(X_{t,K}, X_{t,K}^1 ... X_{t,K}^P) \Rightarrow X_{post}$$

Gradient-based 算法也会定义一个 loss function L, 减小它就可以改进 policy 以及附加的网络

$$L(\theta; X) \Rightarrow loss$$

最后，开发者也可以定义一系列的效用函数 u^i , 在训练的时候调用，比如返回训练数据 s 更新网络

$$u^1 ... u^M(\theta) \Rightarrow (s, \theta_{update})$$

举例来说

```
abstract class rllib.PolicyGraph:
  def act(self, obs, h): action, h, y* #policy model
  def postprocess(self, batch, b*): batch #trajectory postprocessor
  def gradients(self, batch): grads #计算梯度
  def get_weights; def set_weights;
  def u*(self, args*) #效用函数
```

Policy Evaluation

概念 给定 environment 和 policy graph, policy evaluation 会产生一系列的 experiences(batches). 就是所谓的"Environment interaction loop". 通常来说，这一项很繁重，特别是利用了向量化、RNN 或者 multi-agent 环境。管理所有这一切的是一个类 PolicyEvaluator.

也可以单独使用 policy evaluation 来生成一系列 experience. `ev.sample()` 或者并行地 `ev.sample.remote()`

具体 为了收集 experience, 提供了 `PolicyEvaluator` 类，将 policy graph 和环境包装起来，然后添加一个 method `sample()`. Policy evaluator 可以作为 remote actors 被创建，为了并行性，也可以在不同的集群直接复制。

考虑一个具体的案例，使用 TensorFlow policy gradient. 可以使用 RLlib 实现的模板

```
class PolicyGradient(TFPolicyGraph):
  def __init__(self, obs_space, act_space):
    self.obs, self.advantages = ...
    pi = FullyConnectedNetwork(self.obs)  # 全连接网络处理observation
    dist = rllib.action_dist(act_space, pi) #计算action 分布
    self.act = dist.sample() #采样，得到action
    self.loss = -tf.reduce_mean(
      dist.logp(self.act) * self.advantages)  #计算loss
  def postprocess(self, batch):
    return rllib.compute_advantages(batch)
```

开发者可以创建一系列 policy evaluator `ev`，然后调用 `ev.sample.remote()` 来从环境中收集 experience, 可以并行。

```
evaluators = [rllib.PolicyEvaluator.remote(env=SomeEnv, graph=PolicyGradient)
for _ in range(10)]
    print(ray.get([ev.sample.remote() for ev in evaluators]))
```

环境可以是OpenAI Gym或者用户自定义的或者 multi-agent 或者 batched environments.

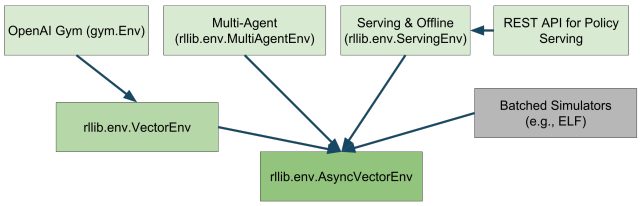


图 14:

Policy Optimization

概念 与 tensorflow 的gradient-descent optimizer 来改进一个模型一样, policy optimizers也实现了一系列不同的策略来改进 policy graph.

比如AsyncGradientsOptimizer实现了 A3C, 在不同的 worker 上异步地计算梯度。

比如 SyncSamplesOptimizer 并行且同步地收集 experiences, 然后中心化地优化模型。

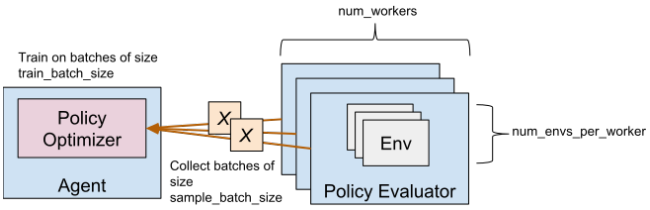


图 15:

具体 RLlib 将算法实现分为两个部分，一个是 policy graph(随算法的不同而不同)，一个是 policy optimizer(基本是相同的).

policy optimizer 主要做的是：分布式采样，参数更新，管理 replay buffers. 为了将计算分布开，optimizer 在一系列的 policy evaluator 复制品上操作。

如下，每个 `optimizer.step()` 都会运行一些 remote tasks 来改进 model. 另外，在每个 `step` 之间，policy graph 的复制品也可以之间访问，比如打印一些东西。

```
optimizer = rllib.AsyncPolicyOptimizer(graph=PolicyGradient, workers=evaluators) #定义在policy graph与
       evaluators上
while True:
  optimizer.step()  #改进算法
  print(optimizer.foreach_policy(
    lambda p: p.get_train_stats()))
```

实现的内容是典型的梯度下降优化

$$step(L(\theta), X, \theta) \Rightarrow \theta_{opt}$$

相应的，RLlib 的 policy optimizer 在 local policy graph G 和一系列 remote evaluator replicas 上面操作。比如

$$step(G, ev_1...ev_n, \theta) \Rightarrow \theta_{opt}$$

此处可以调用 policy evaluator 上面的 `sample()` 来产生新的 simulation 数据

这种 policy optimizer 抽象的优势为有

- 将 execution strategy 和 policy/loss 定义分开，可以让优化策略充分利用硬件调节，而无需修改其他算法
- policy graph 类封装起来与深度学习框架交互，让算法设计者避免将分布式系统与数值计算混合
- optimizer 可以不断被改进或在不同深度学习框架下重用

优化策略

- **Allreduce**

```
grads = [ev.grad(ev.sample())
  for ev in evaluators: #取每个evaluator的采样
       avg_grad = aggregate(grads) #综合
local_graph.apply(avg_grad)  #更新本地graph
weights = broadcast(local_graph.weights())  #
for ev in evaluators:  #
  ev.set_weights(weights)
```

- **Local Mutil-GPU**

```
samples = concat([ev.sample()
  for ev in evaluators])
pin_in_local_gpu_memory(samples)
for _ in range(NUM_SGD_EPOCHS):
  local_g.apply(local_g.grad(samples)
weights = broadcast(local_g.weights())
for ev in evaluators:
  ev.set_weights(weights)
```

- **Asynchronous**

```
grads = [ev.grad(ev.sample())
         for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
  grad, ev, grads = wait(grads)
  local_graph.apply(grad)
  ev.set_weights(
      local_graph.get_weights())
    grads.append(ev.grad(ev.sample()))
```

- **Sharded Param-server**

```
grads = [ev.grad(ev.sample())
  for ev in evaluators]
for _ in range(NUM_ASYNC_GRADS):
  grad, ev, grads = wait(grads)
  for ps, g in split(grad, ps_shards):
    ps.push(g)
  ev.set_weights(concat(
    [ps.pull() for ps in ps_shards])
  grads.append(ev.grad(ev.sample()))
```

Compatibility matrix:

Algorithm	Discrete Actions	Continuous Actions	Multi-Agent	Recurrent Policies
A2C, A3C	**Yes**	**Yes**	**Yes**	**Yes**
PPO	**Yes**	**Yes**	**Yes**	**Yes**
PG	**Yes**	**Yes**	**Yes**	**Yes**
IMPALA	**Yes**	No	**Yes**	**Yes**
DQN, Rainbow	**Yes**	No	**Yes**	No
DDPG	No	**Yes**	**Yes**	No
APEX-DQN	**Yes**	No	**Yes**	No
APEX-DDPG	No	**Yes**	**Yes**	No
ES	**Yes**	**Yes**	No	No
ARS	**Yes**	**Yes**	No	No

12 RLlib 深入

继续深入 RLlib 的代码部分

12.1 Overview

- 没有严格的接口。几乎所有的东西都可以自定义，包括算法、model、优化策略等等
- 提供一些方便的类 (actor). 可扩充的地方很多，比如更多的算法实现，嵌入更多的 DL 模型
- 提供方便的参数调整 (主要以 config 字典形式设定), 便于在现有模型的基础上，微调。同时提供已经优化好的参数案例 `tuned_examples`。

12.2 工作流程

从一个运行的案例来说明其构造原理是合适的。

```
python ray/python/ray/rllib/train.py --run DQN --env CartPole-v0
```

1. `train.py`

- 基本的分解参数
- 调用相应算法和 env
- 后续的运行交给 `tune` 库来控制. 在运行时候，自然会调用相应的 `RLlib` 内容

  这个库也有很多东西。暂时没有深入

2. `agents/dqn/dqn.py`

- agent 作为最顶层的实现

  DQN(Deep Q Network) 是一种 RL 算法，基于这种算法封装成为一个 agent, 便于 train/store/restore 等等

- 基本参数的设定 (在通用 agent 设定的基础上改写)

  包括是否为 double Q-learning, 设定 hidden layers 的大小, n-step 的设定.

  replay buffer 的设定, adam 算法的设定, 并行性的设定 (workers 数量)

- 定义了 `DQNAgent` 类，继承自 `Agent` 类，而 `Agent` 类继承自 `tune` 库中一个类 `Trainable`. 这就是为什么 `tune` 库可以方便地控制运行 agent
- 上述 `DQNAgent` 类使用 `dqn_policy_graph` 来构造 `evaluator`

  上次提到 evaluator 是利用 policy 与环境交互，获得 trajectory, 然后利用它来更新 policy(在 deep Q-learning 中是更新两个 Q network 的参数). 这部分由 policy graph 来做，返回一定的度量 metrics

  同时 evaluator 可以使本地的也可以是 remote. 相当于可以并行

- 最后的结果由 `collect_metrics` 来获取, 然后返回到 `tune` 内的控制单元，看是否需要继续训练

关于 deep Q-learning

基本算法CS 294-112 at UC Berkeley

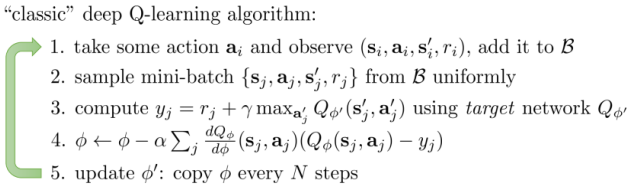


图 16:

double Q-Learning

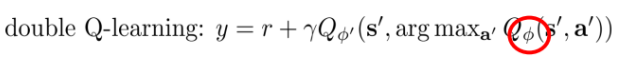


图 17:

同步异步的并行方式

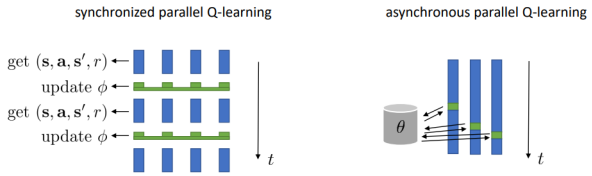


图 18:

3. `dqn_policy_graph.py`

- 继承自 `TFPolicyGraph`, 后者继承自 `PolicyGraph`

  前面提到过, PolicyGraph 就是将 policy 封装起来，然后实现一些方法，比如计算下一步 action, 处理 trajectory, 计算 gradients, 更新参数等等

`TFPolicyGraph` 是实现的一个比较方便的基于 tensorflow 的模板,另外实现的还有 `TorchPolicyGraph` 以及 `KerasPolicyGraph` 但是 build-in 算法里面大部分是使用 tensorflow 实现的

- 实现 deep q-learning

4. `policy_evaluator.py`

- 将 `policy graph` 与 `env` 包装起来，实现来收集 experience

  可以有很多的复制品，在 Ray 中作为 actor 来实现。

5. `policy_optimizer.py`

- control plane

  需要将 evaluator 传入

- 继承此类的优化算法

  比如异步计算 gradients, 异步/同步利用 replay buffer, 异步/同步采样, 多 GPU 等等

12.3 其他重要组件

1. `models`

功能: 输入 obeservations 输出 logits 的神经网络

- 提供 `tensorflow` 和 `pytorch` 两种实现，但是后者实现不完全，所以基本上默认是使用 `tensorflow`
- `Preprocessor`
  - image 预处理
  - One hot 预处理
  - 等等
- 多种网络
  - fully connected
  - CNN
  - LSTM
- `ActionDistribution`

  由上面得到的 logits 来决定下一步的 action

  - 离散的 (Categorical)
  - 连续的 (Gaussian)
  - 确定性的 (Deterministic)
  - 多个动作等等

2. `tuned_examples`

已经优化好的算法的设定。比如 dqn 算法使用多少 step, 比如使用多少 gpu, 比如 cnn 如何设定卷积核等等

12.4 总结

从代码上来看 RLlib 与 tune lib 密不可分，而这两个又与 Ray 底层以及 tensorflow 密不可分
如果抛开这些依赖，实现顺序可能是:

1. model 各种神经网络
2. policy graph 在 model 的基础上实现 action 的选择
3. evaluator 将 policy graph 以及 env 包装起来，以实现交互

4. optimizer 复制 evaluator 到不同机器上，优化参数
5. agent 对整个训练的封装

13 RLlib 性能

13.1 papers

没有找到直接使用 `RLlib` 并且公开源码的学术论文，也没有找到相关的评测论文。
事实上在 google scholar 上能找到的相关论文如下

- 介绍 ray 或者 rllib 的原作者论文
- 基于 `RLlib` 或 `RLlab` 构建的另一个系统
  Flow: Deep Reinforcement Learning for Control in SUMO
  其中关于 `RLlib` 与 `RLlab` 的一个简短评测，很难作为完全可靠的依据。并且 RLlab 已经不是一个活跃的开发项目了，上一次的更新也是 2 个月前了

Nodes	CPUs	rllab rollout (s)	RLlib rollout(s)	Speedup	Cost ($/hr)
1	72	79s	74s	2.58x	1.08
1	16	205s	191s	1.00x	0.24
2	32	*N/A*	98s	1.95x	0.48
4	64	*N/A*	62s	3.08x	0.96
8	128	*N/A*	42s	4.55x	1.93

图 19:

- 基于上述 ***Flow*** 做的 benchmarks
- 针对当前的 RL 框架的不足构建新的框架
  RLGRAPH: FLEXIBLE COMPUTATION GRAPHS FOR DEEP REINFORCEMENT LEARNING

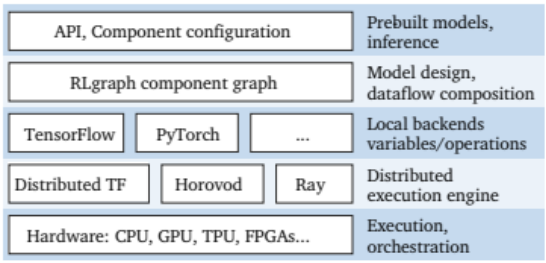


Figure 1. RLgraph stack for using and designing RL algorithms.

图 20:

13.2 others

在rl-experiments找到了他们自己做的测试，并且与公开的结果进行对比。
一部分是不同的算法比较 scores, 一部分是相同的算法也比较 scores, 并没有直接比较时间性能的。
比如：

env	RLlib Basic DQN	RLlib Dueling DDQN	RLlib Distributional DQN	Hessel et al. DQN	Hessel et al
BeamRider	2869	1910	4447	~2000	~13000
Breakout	287	312	410	~150	~300
QBert	3921	7968	15780	~4000	~20000

env	RLlib Basic DQN	RLlib Dueling DDQN	RLlib Distributional DQN	Hessel et al. DQN	Hessel et al
SpaceInvaders	650	1001	1025	~500	~2000

14 Tune

介绍 ray 中至关重要的 tune 库

14.1 前言

首先回顾一下关于整个 ray 系列的结构。

- ***Ray: Flexible, High-Performance Distributed Execution Framework***
  ray 作为底层，实现分布式计算的底层框架
- ***Tune: Scalable Hyperparameter Search***
  实现机器学习中的超参搜索
- ***RLlib: Scalable Reinforcement Learning***
  实现可拓展的强化学习框架

14.2 Features

- 支持任何深度学习框架，包括 PyTorch, TensorFlow, Keras
- 可从一些可拓展的 hyperparameter 与 model 搜索技术中选择，比如：
  - Population Based Training (PBT)
  - Median Stopping Rule
  - HyperBand
- 混合不同的 hyperparameter 优化方式，比如HyperOpt with HyperBand
- 可以使用多种方式可视化结果，比如：TensorBoard, parallel coordinates (Plot.ly), rllab's VisKit.
- 无需修改代码就可以拓展到在大分布式集群上运行
- 并行化，比如使用 GPU 或者算法本身就支持并行化和分布式，使用 Tune's resource-aware scheduling,

14.3 Introduction

14.3.1 Overview

Tune 在集群中调度一系列的 trials. 每个 trials 执行一个用户自定义的 Python function 或者 class, 并且每个 trial 的参数要么由 `config` 变体 (Tune's Variant Generator) 定义, 要么由用户自定义的搜索算法定义。最后，所有的 trials 由 trial scheduler 定义

14.3.2 Experiment 流程

构建 model 如下使用 keras 构建一个识别 mnist 数据中数字的 CNN
是一个很平常的神经网络，返回的 model 就是输出 image 然后输出分类 0-9

```
def make_model(parameters):
    config = DEFAULT_ARGS.copy()  # This is obtained via the global scope
    config.update(parameters)
    num_classes = 10

    model = Sequential()
    model.add(Conv2D(32, kernel_size=(config["kernel1"], config["kernel1"]),
                     activation='relu', input_shape=(28, 28, 1)))
    model.add(Conv2D(64, (config["kernel2"], config["kernel2"]), activation='relu'))
```

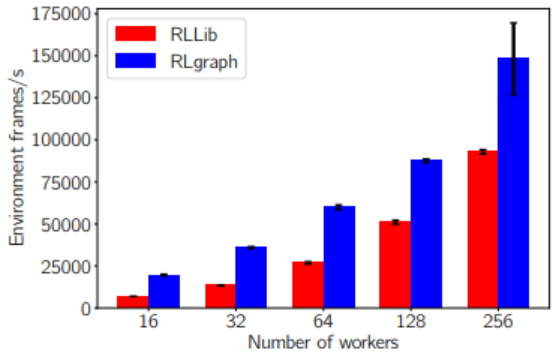


Figure 5. Distributed sample throughput on *Pong*.

图 21:

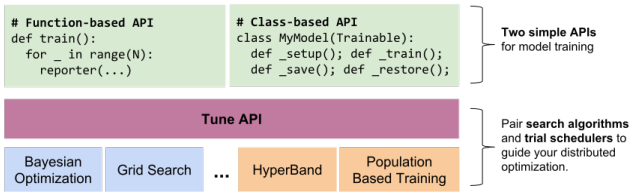


图 22:

```
    model.add(MaxPooling2D(pool_size=(config["poolsize"], config["poolsize"])))
    model.add(Dropout(config["dropout1"]))
    model.add(Flatten())
    model.add(Dense(config["hidden"], activation='relu'))
    model.add(Dropout(config["dropout2"]))
    model.add(Dense(num_classes, activation='softmax'))

    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.SGD(
                      lr=config["lr"], momentum=config["momentum"]),
                  metrics=['accuracy'])
    return model
```

使用 tune 来训练模型 主要是寻找超参

两种方式实现 training 的代码构建

- Function

  这里定义的函数需要传入两个对象，config 和 reporter，其中 config 是用于超参搜索的 dict 内容，reporter 用于返回 metrics 给 Tune 来控制什么时候停止

```
def train_mnist_tune(config, reporter):
    data_generator = load_data()
    model = make_model(config)
    for i, (x_batch, y_batch) in enumerate(data_generator):
        model.fit(x_batch, y_batch, verbose=0)
        if i % 3 == 0:
            last_checkpoint = "weights_tune_{}.h5".format(i)
            model.save_weights(last_checkpoint)
        mean_accuracy = model.evaluate(x_batch, y_batch)[1]
        reporter(mean_accuracy=result[1], timesteps_total=i, checkpoint=last_checkpoint)
```

- Class

作为 `ray.tune.Trainable` 的子类。RLlib 中的优化相关类就是作为 `Trainable` 的子类

设定超参的搜索空间以及其他 两种方式设置运行的 config

- Python
- JSON

传入的参数重要的有

`stop(dict)`: 确什么时候停止，比如设定准确率为 0.95

`config`: 设定超参的搜索空间，比如 Learning Rate 设为一系列小数

下面是使用 Python 的一个案例，使用 Json 很类似。

```
configuration = tune.Experiment(
    "experiment_name",
    run=train_mnist_tune,
    trial_resources={"cpu": 4},
    stop={"mean_accuracy": 0.95},
    config={"lr": lambda spec: np.random.uniform(0.001, 0.1),
            "momentum": tune.grid_search([0.2, 0.4, 0.6])}
)
```

执行 传入的重要参数有

上面的 tune.Experiment 一系列设定

search_alg 优化的搜索算法

scheduler 执行的调度器。下面会设定这一个

```
trials = tune.run_experiments(configuration, verbose=False)
```

使用调度器 最大的不同之处就在于，使用了 `AsyncHyperBandScheduler` 调度器。如果不指定调度器，就会直接使用 `FIFOScheduler`

主要目的是更好地利用分布的资源

```
configuration2 = tune.Experiment(
    "experiment2",
    run=train_mnist_tune,
    num_samples=5,
    trial_resources={"cpu": 4},
    stop={"mean_accuracy": 0.95},
    config={
        "lr": lambda spec: np.random.uniform(0.001, 0.1),
        "momentum": tune.grid_search([0.2, 0.4, 0.6]),
        "hidden": lambda spec: np.random.randint(16, high=513),
    }
)

hyperband = AsyncHyperBandScheduler(
    time_attr='timesteps_total',
    reward_attr='mean_accuracy')

trials = tune.run_experiments(configuration2, scheduler=hyperband, verbose=False)
```

15 Ray 系列算法

讨论在整个 *ray* 系列中出现的算法

15.1 Tune

注意 Trial Schedulers 与 Search Algorithms 是不同的。前者安排一系列 Trials 如何执行执行顺序，后者确定每次的 Hyperparameter Configuration.

15.1.1 Tune Trial Schedulers

Popilation Based Training (PBT) DeepMind Blog-PBT

Population Based Training of Neural Networks

本论文首先介绍当前的 hyperparameter 调整的方式分为如下

- Parallel Search

  并行多个进程，每个进程不同的 hyperparameter. 最后将结果比较，选择最好的那个
  - Grid Search
  - Random Search
- Sequential Optimisation

  每次运行一个，然后根据这个的结果，重新选择 hyperparameter, 循环往复，直到比较好的结果

用流程图来表示就是：

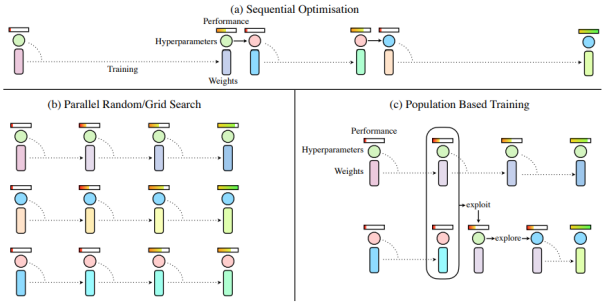


图 23:

而 PBT 就是，并行一系列 hyperparameter 不同的模型，然后在中间将那些结果不好的模型的 parameter 和 hyperparameter 换成较好的那个 (exploit), 并且加上一些随机的噪声 (explore).

Algorithm 1 Population Based Training (PBT)

procedure TRAIN($\mathcal{P}$) ▷ initial population $\mathcal{P}$
  for $(\theta, h, p, t) \in \mathcal{P}$ (asynchronously in parallel) **do**
    while not end of training **do**
      $\theta \leftarrow \texttt{step}(\theta|h)$ ▷ one step of optimisation using hyperparameters h
      $p \leftarrow \texttt{eval}(\theta)$ ▷ current model evaluation
      if $\texttt{ready}(p, t, \mathcal{P})$ **then**
        $h', \theta' \leftarrow \texttt{exploit}(h, \theta, p, \mathcal{P})$ ▷ use the rest of population to find better solution
        if $\theta \neq \theta'$ **then**
          $h, \theta \leftarrow \texttt{explore}(h', \theta', \mathcal{P})$ ▷ produce new hyperparameters h
          $p \leftarrow \texttt{eval}(\theta)$ ▷ new model evaluation
        end if
      end if
      update $\mathcal{P}$ with new $(\theta, h, p, t+1)$ ▷ update population
    end while
  end for
  return θ with the highest p in $\mathcal{P}$
end procedure

图 24:

Asynchronous HyperBand Massively Parallel Hyperparameter Tuning

这个算法是下面的 HyperBand 异步推广的结果。它的算法为

Algorithm 1: Asynchronous Successive Halving Algorithm.

Input: r, η (default $\eta = 3$), s
Algorithm `async_SHA()`
  repeat
    for free worker **do**
      $(\theta, k) =$ `get_job()`
      worker performs `run_then_return_val_loss`$(\theta, r\eta^{s+k})$
    end
    for completed job (θ, k) with loss l **do**
      Update configuration θ in rung k with loss l.
    end

Procedure `get_job()`
  `// A configuration in a given rung is "promotable" if its validation loss places it in the top` $1/\eta$ `fraction of completed configurations in its rung and it has not already been promoted.`
  Let θ_k be the furthest trained *promotable* configuration θ, with k indicating its rung.
  if θ_k exists **then**
    Promote θ_k to rung $k + 1$.
    return $\theta_k, k + 1$
  else
    Add a new configuration θ_0 to bottom rung.
    return $\theta_0, 0$
  end

图 25:

论文中只给出了 Successive Halving 算法 (SHA), 由于下面的 HyperBand 使用了 SHA 的子过程，所以很容易补充成为 Asynchronous HyperBand.

输入的参数为 r 最小资源，$\eta > 0$ 表示 reduction factor, s 表示最小的 early-stop rate.

注意到 rung 越大，表示这个超参的设定越有前景，则分配到的资源越大。

同时由于 get_job() 函数的存在，使得算法是异步的，当发现存在 promotable 设定的时候，就返回这个设定，并且将 rung 加一，如果不存在，也不用等待其他的结束，而是直接生成一个新的设定，且 rung=0

HyperBand standard version of HyperBand

https://people.eecs.berkeley.edu/~kjamieson/hyperband.html

这个的实际目标是，资源 B 是有限的，尝试的次数 n 可以选择，每次尝试分配的资源是 B/n, 那么如何分配资源，n 可大可小。

提出的算法如下

Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.

input : R, η (default $\eta = 3$)
initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$
for $s \in \{s_{\max}, s_{\max} - 1, \ldots, 0\}$ **do**
  $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil$, $r = R\eta^{-s}$
  `// begin SuccessiveHalving with` (n, r) `inner loop`
  $T =$ `get_hyperparameter_configuration`(n)
  for $i \in \{0, \ldots, s\}$ **do**
    $n_i = \lfloor n\eta^{-i} \rfloor$
    $r_i = r\eta^i$
    $L = \{$`run_then_return_val_loss`$(t, r_i) : t \in T\}$
    $T =$ `top_k`$(T, L, \lfloor n_i/\eta \rfloor)$
  end
end
return *Configuration with the smallest intermediate loss seen so far.*

图 26:

设定一个案例如下：

	$s=4$		$s=3$		$s=2$		$s=1$		$s=0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

Table 1: The values of n_i and r_i for the brackets of HYPERBAND corresponding to various values of s, when $R = 81$ and $\eta = 3$.

图 27:

将 trial 分成很多部分，每次 trial 最大资源为 R, 分成 $(s_{max} = \lfloor \log_\eta(R) \rfloor) + 1$ 个阶段，每个阶段总资源为 $B = (s_{max} + 1)R$ 每个阶段又分成多个子部分，其中当尝试的个数 n_i 越大，分配的资源 r_i 越小，越会提早结束探索。每个 SHA 子过程都使用不同的 early-stop rate.(由于现代的超参调试问题都有高维的搜索空间，并且模型有很大的训练代价，所以提前终止是很有必要的)

其中 get*hyperparameter*configuration(n) 表示从超参的设定集合中采样 n 个独立同分布的样本。

run*then*return*val*loss(t,ri) 表示超参为 t, 资源为 ri 时的 validation loss

python 代码如下

```
# you need to write the following hooks for your custom problem
from problem import get_random_hyperparameter_configuration,run_then_return_val_loss

max_iter = 81  # maximum iterations/epochs per configuration
eta = 3 # defines downsampling rate (default=3)
logeta = lambda x: log(x)/log(eta)
s_max = int(logeta(max_iter))  # number of unique executions of Successive Halving (minus one)
B = (s_max+1)*max_iter  # total number of iterations (without reuse) per execution of Succesive Halving (n,r
        )

#### Begin Finite Horizon Hyperband outlerloop. Repeat indefinetely.
for s in reversed(range(s_max+1)):
    n = int(ceil(int(B/max_iter/(s+1))*eta**s)) # initial number of configurations
    r = max_iter*eta**(-s) # initial number of iterations to run configurations for

    #### Begin Finite Horizon Successive Halving with (n,r)
    T = [ get_random_hyperparameter_configuration() for i in range(n) ]
    for i in range(s+1):
        # Run each of the n_i configs for r_i iterations and keep best n_i/eta
        n_i = n*eta**(-i)
        r_i = r*eta**(i)
        val_losses = [ run_then_return_val_loss(num_iters=r_i,hyperparameters=t) for t in T ]
        T = [ T[i] for i in argsort(val_losses)[0:int( n_i/eta )] ]
    #### End Finite Horizon Successive Halving with (n,r)
```

Median Stopping Rule Google Vizier: A Service for Black-Box Optimization

这篇文章介绍的是 google 研发的 Black0Box Optimization 系统。主要介绍了系统的组成。略。

15.1.2 Tune Search Algorithms

Variant Generation (Grid Search/Random Search) 不必过多介绍。

HyperOpt Search (Tree-structured Parzen Estimators) Hyperopt Distributed Asynchronous Hyper-parameter Optimization in Python

这其实是一个 Python 库：

> `hyperopt` is a Python library for optimizing over awkward search spaces with real-valued, discrete, and conditional dimensions.

目前实现的算法有

- Random Search
- Tree of Parzen Estimators (TPE)

使用案例

安装 `pip install hyperopt`

```
from hyperopt import hp

```

```
# define an objective function
def objective(args):
    case, val = args
    if case == 'case 1':
        return val
    else:
        return val ** 2

# define a search space
from hyperopt import hp
space = hp.choice('a',
    [
        ('case 1', 1 + hp.lognormal('c1', 0, 1)),
        ('case 2', hp.uniform('c2', -10, 10))
    ])

# minimize the objective over the space
from hyperopt import fmin, tpe, space_eval
best = fmin(objective, space, algo=tpe.suggest, max_evals=100)

print(best)
print(space_eval(space,best))
```

输出为

```
{'a': 1, 'c2': -0.08088083656564893}
('case 2', -0.08088083656564893)
```

注意到用法其实很简单，定义 objective, 设定 search space, 选择 search aalgorithms, 设定 evaluations 数目。

ray 里面其实是调用这个库来实现的。

15.2 RLlib

15.2.1 RLlib Algorithms

High-throughput architectures

Distributed Prioritized Experience Replay (Ape-X) Distributed Prioritized Experience Replay 是 DQN 与 DDPG 关于 Apx-X 的变体。使用一个 GPU learner 与多个 CPU workers, 用于 experience collection.

> Experience collection can scale to hundreds of CPU workers due to the distributed prioritization of experience prior to storage in replay buffers

Importance Weighted Actor-Learner Architecture (IMPALA) IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures

在 IMPALA 中，一个中心的 learner 在一个很紧凑的循环里执行 SGD, 同时异步地从许多 actor processes 里面拉取样本 batches

Gradient-based

Advantage Actor-Critic (A2C, A3C) Asynchronous Methods for Deep Reinforcement Learning

Deep Deterministic Policy Gradients (DDPG, TD3) Continuous control with deep reinforcement learning

DDPG 实现与 DQN 实现很像。

DDPG 同时学习 Q-Function 和 policy. 使用 off-policy 数据和 Bellman 等式来学习 Q-function, 然后使用 Q-function 来学习 policy.

这种方式与 Q-learning 联系非常紧密，也是有同样的动机：如果知道了 optimal action-value 函数 $Q^*(s,a)$, 然后如果给定 state, 就可以计算出 optimal action:

$$a^*(s) = \arg\max_a Q^*(s,a)$$

DDPG 交错学习一个 $Q^*(s,a)$ approximator 和一个 $a^*(s)$ 的 approximator,

对于离散的 action 还可以直接计算最大值，但是对于连续的，就不能遍历计算。但是由于 action space 是连续的，$Q^*(s,a)$ 就假定对于 a 可微，policy $\mu(s)$ 就利用了这个事实。所以，现在不需要每次都耗费资源计算 $\max_a Q(s,a)$, 而是用 $\max_a Q(s,a) \approx Q(s,\mu(s))$ 来近似。

注意到 DDPG 是 off-policy 的，而且仅适用于连续的 action spaces. 可以看为是连续版本的 DQN.

DDPG 包含两个部分：学习 Q-function 和学习 policy

The Q-Learning Side of DDPG

首先是关于 optimal action-value 函数的 Bellman 等式

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

其中 $s' \sim P$ 表示下一个 state s' 是依据分布 $P(\cdot|s,a)$ 来从环境中采样的。

The Policy Learning Side of DDPG

Deep Q Networks (DQN, Rainbow, Parametric DQN) Playing Atari with Deep Reinforcement Learning

Rainbow: Combining Improvements in Deep Reinforcement Learning

DQN

DQN 是第一个使用 DL 直接从高维数据中学习 control policy 作为 RL 输入的成功的模型。

该论文中提到了 DL 应用于 RL 面临的困境：

- 大部分成功的 DL 应用都需要大量标注了的数据。另一方面，RL 必须要从标量 reward 中学习，而这种 reward 通常是稀疏的、充满噪声的、延迟的。
- 大部分 DL 要求采样的数据必须是独立的。但是 RL 通常是连续的高度相关的状态。
- DL 通常假设数据的分布是固定的，但是 RL 的数据分布却通常随着 behaviours 的不同而不同

本论文为了解决数据相关问题以及不稳定的分布问题，使用了 Experience Replay 机制，也就是每次的转移都存起来，训练的时候随机从存储里面抽取。

文章使用了 Q Learning, 所以介绍了 Q Learning

Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a')\middle| s,a\right]$$

可以使用 *value iteration* 来收敛到该 optimal action-value funtion

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a')\middle| s,a\right]$$

$Q_i \to Q^*$ as $i \to \infty$. 但是实际上这种方式是完全不可行的，因为对于每个 sequence, action-value 函数的估计是分开的，这样就失去了 generalisation. 所以通常会使用 function approximator 来估计 action-value 函数，即 $Q(s,a;\theta) \approx Q^*(s,a)$. 通常会使用 linear function, 偶尔使用 non-linear 函数近似，比如 NN.

这篇论文中使用的他们称为 Q-network, 可以通过如下的 loss function 来优化 ($L_i(\theta_i)$ 表示每个循环 i 都会改变)

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\Big[(y_i - Q(s,a;\theta_i))^2\Big]$$

其中 y_i 为 target value. 它会随着循环而改变，这不同于监督学习。

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\Big[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})\Big|s,a\Big]$$

ρ 表示 behaviour distribution.

Loss function 的微分为

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\Big[\Big(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\nabla_{\theta_i} Q(s,a;\theta_i)\Big)\Big]$$

注意上述算法有如下的特点

- model-free: 并没有构建关于 estimator $\mathcal{E}$ 的模型
- off-policy: 学习的是 greedy strategy $a = \max_a Q(s,a;\theta)$, 但是使用的 behavior 通常是 ϵ-greedy 的

接下来就是详细介绍了 DQN 算法

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

图 28:

需要注意的是，由于原始的像素数据太大，并不适合作为模型输入，所以添加了一个 Φ 函数来预处理 image

Rainbow

在 DQN 提出来之后，有许多的论文提出了改进意见，这篇文章是抽取 6 种改进，然后将它们组合进行评测。使用的 extensions 如下:

1. Double Q-Learning

   首先注意到 DQN 是在减小如下 loss

   $$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2$$

   注意到其中的 max 部分，选择 action 使用的是 target value, 对选用的 action 来评估也是使用 target value, 这样有可能过高地估计某个 action 的效力，导致整个 q 都偏高。改进方式就是使用不同的 value. 考虑到计算量，DDQN 使用的是，q_θ 来选择 action, 由 $q_{\bar{\theta}}$ 来评估。式子如下:

   $$(R_{t+1} + \gamma_{t+1} q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2$$

2. Prioritized replay

   DQN 从 replay buffer 中取样是均匀的，但是有一些数据更加有价值，应该更多地去学。所以该方法使用了某种概率 p_t 来选择，该概率与 last encountered absolute TD error:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t) \right|^\omega$$

其中 ω 是决定分布形状的参数。新加进来的数据会有最大的可能性。

3. Dueling Network
4. Multi-step Learning

注意 DQN 使用了第一个 reward 然后后面就是估计值，其实还可以使用多个

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

相应的，loss 改为

$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2$$

5. Distributional RL
6. Noisy Nets

这里面好几个我自己没有接触过，更加详细的说明留在以后。

Policy Gradients Policy Gradient Methods for Reinforcement Learning with Function Approximation

vabilla policy gradients. 下面的 PPO 表现更好

算法的核心就是将那些能够获得高 return 的 action 的概率提高，而将地 return 的 action 概率降低。

首先这是一个 on-policy 的算法，适用的 env 的 action 可分离可连续。

用 π_θ 表示含参数 θ 的 policy, 用 $J(\pi_\theta)$ 表示有终结的没有 discount 的 return. 相应的 gradient 为

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right]$$

其中 τ 表示 trajectory, A^{π_θ} 表示 advantage function, 也就是选择 a_t 与所以 action 的 return 的均值的差。

PG 算法就是通过 stochastic gradient ascent 来更新 policy 的参数的

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

PG 是一个 on-policy 的，也就是说它的 exploration 是依据最新的 policy 采样而来的。所以 action 的随机性取决于初始的设置与训练的过程。随着训练的进行，policy 通常会慢慢减小随机性，这样就容易导致得到一个局部最优解，而且没有办法出来。

其算法如下

Proximal Policy Optimization (PPO) Proximal Policy Optimization Algorithms

首先需要介绍 Trust Region Policy Optimization(TRPO)

TRPO

TRPO 是建立在 PG 基础上的。TRPO 使用尽可能大的 step 来提高 performance, 并且要满足新的 policy 要接近旧的 policy. 由于 policy 也是一种概率，所以这种限制是使用 KL-Divergence 来实现的。

与普通的 PG 不同的是新旧的 policy 尽可能接近，这是因为一个小小的坏的 step 都可能使整个 policy 效果不好。TRPO 解决了这个，能够很快并且单调地提升 performance.

与 PG 类似，TRPO 也是一个 on-policy 的，并且 env 的 action 空间可离散可连续。

理论上 TRPO 更新的是

$$\begin{aligned} \theta_{k+1} &= \arg\max_\theta \mathcal{L}(\theta_k, \theta) \\ &\text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \le \delta \end{aligned} \tag{1}$$

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

图 29:

其中 $\mathcal{L}(\theta_k, \theta)$ 是 surrogate advantage, 就是在度量 policy π_θ 相比与旧的 policy π_{θ_k} 的性能

$$\mathcal{L}(\theta_k, \theta) = \underset{s,a \sim \pi_{\theta_k}}{E} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

然后 $\bar{D}_{KL}(\theta||\theta_k)$ 是一种平均的 KL-divergence, 关于旧的 policy 遍历的所有的 states

$$\bar{D}_{KL}(\theta||\theta_k) = \underset{s \sim \pi_{\theta_k}}{E} \left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right) \right]$$

但是理论上的 update 并不是很容易计算，所以 TRPO 做了一些近似，使用泰勒展开，同时对 $\mathcal{L}$ 与 D_{KL}

$$\begin{aligned} \mathcal{L}(\theta_k, \theta) &\approx g^T(\theta - \theta_k) \\ \bar{D}_{KL}(\theta||\theta_k) &\approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \end{aligned} \tag{2}$$

所以最终的近似的问题就变成

$$\begin{aligned} \theta_{k+1} = \arg\max_\theta \; & g^T(\theta - \theta_k) \\ \text{s.t. } & \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \le \delta \end{aligned} \tag{3}$$

这个问题可以通过 Lagrangian duality 方式得到分析解

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

如果到此为止，使用上面的结果，那么算法就是在计算 Natural Policy Gradient. 问题在于，由于泰勒展开的近似性，这个更新可能不满足 KL 的约束，或者实际上在更新 surrogate advantage.

TRPO 加上了一个小改动: a backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

其中 $\alpha \in (0,1)$ 是 backtracking 系数，而 j 是使 θ_{k+1} 满足 KL 约束的最小非负整数。

TRPO 的 exploration 与 exploitation 同 PG

PPO

PPO 与 TRPO 有同样的动机：如何采用最大的 step 来更新 policy 而又不更新太多，导致崩坏。TRPO 使用了二阶的方式，很复杂，而 PPO 使用一阶的方式，并且使用一些技巧是新的 policy 尽可能接近旧的 policy.

Algorithm 1 Trust Region Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, \ldots K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

图 30:

有两个基本的 PPO 变体: PPO-Penalty 和 PPO-Clip

PPO-Penalty: 解决一个近似的问题，就像 TRPO 那样，但是是将 KL 限制加到 objective 中，而不是让它成为一个比较难的约束。同时加上一个系数，并且会随着训练的进行变化。

PPO-Clip: 在 objective 中没有 KL, 根本没有约束。而是在 objective 中加上一个特殊的 clipping, 修建那些可能使新的 policy 远离旧 policy 的因素。

与 TRPO 类似，PPO 也是 on-policy，同时适用于 env 的 action 为离散或者连续的。

PPO-Clip 更新 policy 的方式为

$$\theta_{k+1} = \arg\max_\theta \mathop{\mathrm{E}}_{s,a \sim \pi_{\theta_k}} [L(s, a, \theta_k, \theta)]$$

通常是使用使用几步的 minibatch SGD(SGA) 来最大化 objective. 其中 L 是:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s, a) \right)$$

其中 ϵ 是一个很小的超参，就是在限制新旧 policy 的最大距离。

下面有一个简单的版本

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right)$$

其中

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

Derivative-free

Augmented Random Search (ARS) Simple random search provides a competitive approach to reinforcement learning

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

图 31:

Evolution Strategies Evolution Strategies as a Scalable Alternative to Reinforcement Learning

16 对 RLlib 的改进

介绍 Ray 系列的缺点并提出改进意见

RLgraph: Flexible Computation Graphs for Deep Reinforcement Learning

Ray 本身并不是完美的，对于某些特性，有研究人员提出了别的改进的策略。也就是 RLgraph, 其提出的基本问题就是，各个组件之间相互交错，对于测试、执行以及代码的重用都很不友好，于是提出了自己的解决方案，并且做了部分开源实现RLgraph以及相关文档RLgraph docs.

部分内容需要深入代码才能看明白。

由于算法的不稳定性、超参的敏感性以及特征不同的交流方式，RL 任务的实现、执行、测试都很具有挑战性。

16.1 Introduction

针对不同的方面，已经有很多的 RL 库实现了。比如 OpenAI, TensorForce, Ray RLlib. 虽然这些库都有各自不同的目的，但是都面临一些相似的问题，并且导致测试、分布式执行、扩展的困难。其根源就在于 **a lack of separation of concerns**. 定义在 RL 算法中的逻辑组块部分与特定深度学习框架相关的代码紧紧结合在一起，比如说 Ray RLlib 里面就是这样，没有完整地分离，使用 TensorFlow 的部分就是完全依赖的。这就导致了 API 的不良定义，同时也让各个组块的重用与测试变得困难。相似地，RL 复杂的 dataflow 与 control flow 也纠缠在一起。

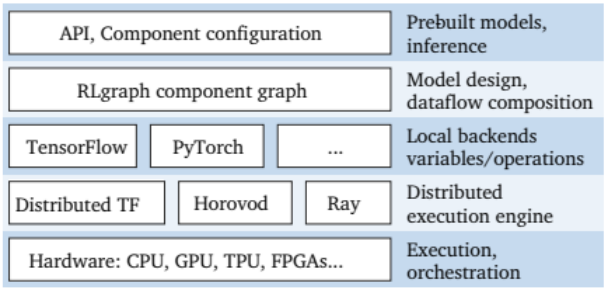


图 32:

这篇文章的核心贡献就是 RLgraph. 解决上述麻烦的办法就在于将 logical component composition, creation of operations, variables, placeholders, local and distributed execution of component graph 分离。如上图所示。

设计的核心就在于 meta graph 结构，这个 graph 主要作用是，集成、连接各个组块，比如 buffers 或 NN, 并且将这些功能用统一的 API 表示。很重要的一点是，这个 graph 与特定的表示实现（比如 TF 变量）不依赖。这就意味着，meta graph 可以同时建立 static graphs 和 define-by-run execution. 也同时支持 TensorFlow 和 PyTorch.

sth omit there

这种设计有如下几个优势：

1. **Distributed execution**. 将设计与执行分开，意味着 agent 可以实现在任何分布执行的框架里，比如 distributed TensorFlow, Ray, Uber's Horovod.
2. **Static and define-by-run backends**. meta graph 的设计并不强加限制在只能它自己的执行方式，这就意味着，不仅能支持 end-to-end static graphs, 包括 control flow, 也能支持 define-by-run semantics 比如 PyTorch, 这一点通过统一的接口实现
3. **Fast development cycles and testing**. 这是由于各个组件的分离。

16.2 Motivation

16.2.1 RL workloads

执行 RL 最中心的困难就在于需要频繁地与环境交互，贯穿于 training, evaluatin 以及 update. 其特征如下：

- **State management**. 样本 trajectories 一般是分布式采样，workers 同时与 env 的复制品进行交互。在一个或者多个 learner 与样本搜集者之间，算法必须保持 model weights 的一致性，这就需要同步和异步的策略了。另外，需要将样本高效地传递给 learner, 有时候需要共享内存。
- **Resource requirements and scale**. 最近成功的 RL 算法需要成百上千的 CPU 以及几百个 GPU. 与此相反，有的算法不易并行，却可能只能用一个 CPU.
- **Models and optimization strategies**. 模型可大可小，将硬件的作用用到极致很困难。

16.2.2 Existing abstractions

- **Reference implementations**. 很多的库仅仅作为一个实现的参考，帮助重新产生研究结果。比如 OpenAI baselines 和 Google's Dopamine 提供了一系列以及优化好的 benchmarks 比如 OPenAI gym 以及 ALE. Nervana Coach 包含相似的东西，但也加上了一些工具，比如可视化、Hierarchical learning、分布式训练。这种实现在很多组块之间共享了算法，比如 NN 结构并且通常叶忽略了实际应用的考虑。所以重新利用他们到不同的模式就很困难
- **Centralized control.** Ray RLlib 定义了一系列抽象，它依赖 Ray 的 actor 模型来执行算法，正如前面讲到的，RLlib 实现中很重要的就是 optimizer, 每个 optimization 都有一个 *step()* 函数，这个函数分发采样任务给 remote actor, 管理 buffers 以及更新 weights. RLlib 自己最宣传的一点就是，在 optimizer 的执行与 RL 算法的定义（由 policy graph）分开。然而每个 optimizer 同时封装了 local 和 distributed 机器的执行，这意味着，比如说，只有专用的多 GPU optimizer 能够同步地在不同 GPU 上分离 input. 使用 optimizer 驱动 control flow 的另一个坏处就是 RLlib 混合 Python control flow, Ray call 以及 TensorFlow call 进它的实现里。所以使用 RLlib 实现的算法就不容易移植，只能在 Ray 上执行。相反地 RLgraph 就不一样了，它支持端到端的计算图，包括 in-graph control-flow, 然后就可以将它们分布在 Ray, distributed TF 等等任何其他的。
- **Fixed end-to-end graphs** 主要是 TensorForce 的问题，略。

16.3 Framework Design

16.3.1 Design principles

没有占优势的单一模式，设计框架就必须解决灵活的原型、可重用的组块以及易拓展机制之间的矛盾。RLgraph 的设计是基于如下几个观点的：

- **Separating algorithms and execution** RL 算法需要复杂的控制流，来协调分布式的采样与内部训练。分割这些组块很困难但是非常有必要。RLgraph 通过 graph executors 来管理 local execution. 分布式的就被指派给专用的 distributed executors, 比如在 Ray 上，或者作为 graph 的一部分，组建到 executor 内部，比如 distributed TF.
- **Shared components with strict interfaces**
- **Sub-graph testing**

16.3.2 Components and meta graph

Components. 然后来讨论 RLgraph 中 component graph 的设计，为了简化，使用 TF 作为基本的后端，其他后端的实现，比如 PyTorch 留在后面。RLgraph 的核心抽象就是 Component 类，这个类通过 graph function 来封装任意的计算。考虑一个 replay buffer component, 这个 component 对外的功能是插入 experiences 和根据优先权重来批量采样。实现这样的 buffer 在命令式的语言，比如 python 是非常直接的，但是将它作为 TF Graph 的一部分却需要通过控制流操作创建和管理很多的变量。将很多种这样的组件按可重用的方式组建很困难。但是使用 define-by-run 的框架比如 PyTorch 却很简单，然而在大容量分布式执行与程序导出层面上却存在困难。

现有的高层次的 NN API 比如 Sonnet, Keras, Gluon 或者 TF.Learn 都专注于构建于训练 NN, 将 RL 实现在这样的框架里面通常需要将命令式的 python 与 DL graph objects 混合起来，这会导致上面提到的设计问题。

当构建在 static graph 后端上时，RLgraph 的 component API 能够快速构建端到端的可微分的 dataflow graph, 利用 in-graph control flow. 并且，graph builder 和 executor 会自动管理 burdensome tasks, 比如变量和 placeholder 的创建, scopes, input spaces 以及 device assignments.

Example component 下图是一个简化的 prioritized replay buffer componenet.

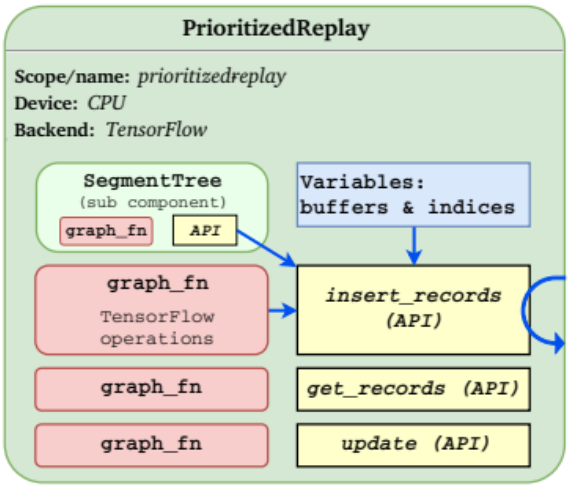


图 33:

所有的 componenetd 都继承自一个通用的 component 类，然后使用它们自己的 sub-components 来构建逻辑。这个 buffer 有一个 segment tree sub-component 来管理优先级顺序。它暴露的 API methods 有 insert, sample 和 update, 这些又关联一些 graph function. 简单的对象 mothod 与 RLgraph API method 的不同点在于，registered API methods 是 identified 然后被管理在 build 中。Input shape 可以被自动推断，从 inputs 到 root component.

开发者能够声明一个 method 为 API method ，通过调用 register function. 从技术上说，并不是一个 component 所有的功能都需要注册为 API method. 用户也可以实现 helper function 或 utilities, 比如说使用 TF 操作但是不将他们包含进 API methos, 如果并不需要从外部 components 调用他们的话。

16.3.3 Building component graphs

RLgraph 用三个不同的阶段来组装。

1. **Component composition phase** 这个阶段中，定义、结合 component objects, 也包括 sub-components 的任意嵌套
2. **Assembly phase** 创建一个很少类型、维度的 dataflow graph. 这是通过调用 root componet 的 API 实现的。
3. **Graph compilation/building phase**

案例如下：

1. **Component composition and nesting** 所有的 components 都被定义为 Python objects. Components 很有逻辑地组织进 root container component, 作为它的 sub-components. 而 root component 则对外暴露 API. 注意到一个 Agent 能够定义多个 root component, 这样就能并行地 act 和 learn 不同的 policy

```
def init(self, *args, **kwargs):
  self.register_api("update", self.update)

def update(self, batch_size):
  sample = self.call(memory.sample, batch_size)
  s, a, r, next_s, t = self.call(splitter.split, sample)
  loss = self.call(dqn_loss.get_loss, s, a, r, next_s, t)
  variables = self.call(policy.get_variables)
  update = self.call(optimizer.step, loss, variables)
  return update, loss

# Envisioned API: Implicit ' call'  via decorators.
# Use space-hints to auto-split, merge nested spaces.
@rlgraph.api_method(split=True)
def observe(records)
  self.memory.insert(records)
```

2. **Assembling the meta-graph** 然后用户可以定义流过 model 的 dataflow, 这是通过 components 之间的连接实现的。在这个阶段 data types 和 shape info 都是不需要的。每个 component 都有一系列的 API-methods. 如上面的程序所示。data 在这些 method 中解释为抽象 meta-graph operator objects, 它们的 shapes 和 types 会在 build 的时候推断出来。在 prototype 实现里面，显式地使用 component call method, 对于所有的 API 唤醒。之后会实现一个更加友好的。

   API 调用的 return 值现在就可以传递给其他的 API-methods 或 sub-component 的 API-method 或者数值计算的 graph function,

   一个简单的 meta graph 算法如下
3. **Building computation graphs**

16.3.4 Agent API

```
abstract class rlgraph.agent:
  # Build with default devices, variable sharing, ..
  def build(options)
  def get_actions(states, explore=True, preprocess=True)
    # Update from internal buffer or external data.
  def update(batch=None)
  # Observe samples for named environments.
  def observe(state, action, reward, terminal, env_id)
  def get_weights, def set_weights
  def import_model, def export_model
```

Algorithm 1 Meta graph build procedure

Input: component $root$, input_spaces $spaces$
$api = dict()$
// Call all api methods once, generate op columns.
for $method, record$ **in** $root.api$ **do**
  $in\_ops\_records = list()$
  // Create one input record per API input param.
  for $param$ **in** $record.input\_args$ **do**
    $in\_ops\_records.append(Op(param.space))$
  end for
  // Traverse graph from root for this method.
  $out\_ops\_records = method(in\_ops\_records)$
  // Register method with graph inputs and output ops.
  $api[method] = [in\_ops\_records, out\_ops\_records]$
end for
return MetaGraph(root, api)

图 34:

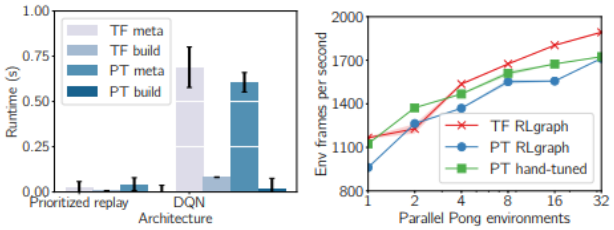
16.4 Executing Graphs

16.4.1 Graph executors.

16.4.2 Backend support and code generation

16.5 Evaluation

16.5.1 Results



(a) Build overheads. (b) Worker act performance.

图 35:

16.5.2 Discussion

16.6 Doc

- Space 类
  - What is a Space?
    在 RLGraph 中，Space 被用来定义数据类型、形状。
    比如，一个 RGB(0-255) 的类型就是 `int8`, 而形状为 `[width*height*3]`
  - 两个主要的 space 类型：BoxSpaces & ContainerSpaces
    BoxSpaces: 就是类似普通的张量
    ContainerSpaces: 包括两类：Tuple, Dict
- The Environment Classes
  - What is an environment ?

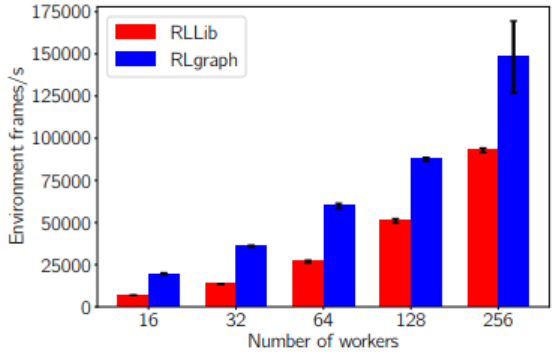
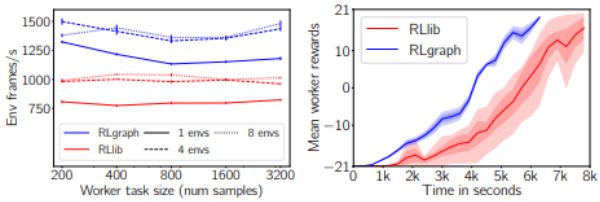


Figure 5. Distributed sample throughput on *Pong*.

图 36:



(a) Single worker throughput. (b) Training times for *Pong*.

Figure 6. Single task throughput and learning comparison.

图 37:

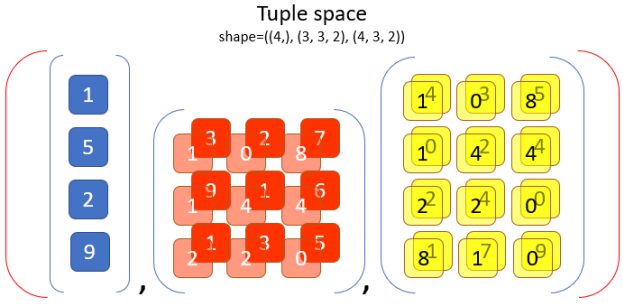


图 38:

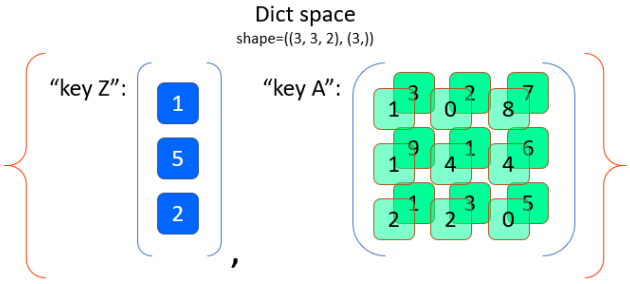


图 39:

  - RLgraph's environment adapters.
    OpenAI, Deepmind Lab, Simple Grid Worlds
- What is an RLgraph Component?
  可嵌套的最小单元，范围很广，比如：一层或者一个 NN，复杂的 policy networks, memories, optimizers, mathematical components (比如 loss functions)

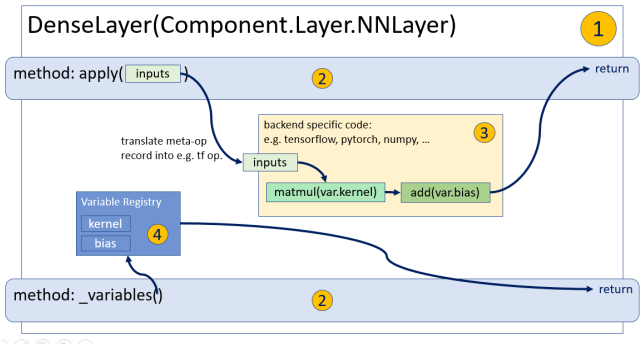


图 40:

上图就是一个 component 包含

1. 两个 API-methods
2. 一个 graph function
3. 两个变量 (kernel, bias)

- The Component Base Class
  包含最核心的部分，每个 Components 都必须继承自它。一些核心的 methods 如下：
  1. `add_components`: 添加任意数目的 `sub-componets`
  2. `check_input_spaces`: 检查 spaces
  3. `create_variables`: 会自动调用，创建一系列的变量，交由 componet 的 computation function 也就是 graph function 使用
  4. `copy`: 复制出一个完全一样的 componet. 比如 target network 和原来的 policy network 一模一样。

  `API_Methods` 相当于交给外界的 handle. 本质上是普通的 class method 加上装饰器。
- How to Write Your Own Custom Component
  - A Simple Single-Value Memory Component
- The Complete Code for Our Custom Component
- How to Test Your Components
  - Writing a New Test Case with Python's Unittest Module
- RLgraph API Reference Documentation
  - 1. RLgraph Core API
  - 1. Space Classes and Space Utilities
  - 1. Agent Classes
  - 1. Components Reference
  - 1. Environment Classes

17 Julia 中的 RL 库

Julia 强化学习库

目前 Julia 对这部分的支持不是很完善，诸多库还该构建当中。

在Julia 官网中有关于的介绍

Julia provides powerful tools for deep learning (Flux.jl and Knet.jl), machine learning and AI. Julia's mathematical syntax makes it an ideal way to express algorithms just as they are written in papers, build trainable models with automatic differentiation, GPU acceleration and support for terabytes of data with JuliaDB.

Julia's rich machine learning and statistics ecosystem includes capabilities for generalized linear models, decision trees, and clustering. You can also find packages for Bayesian Networks andMarkov Chain Monte Carlo.

17.1 Intro

这里介绍的是JuliaML提供的强化学习的包，根据目前的调研，Julia 对强化学习的包很少，而且大部分都不完善。即便是这个在 Julia 官方主页上提到的包，实现的功能也很少。

JuliaML 提供一站式的学习模型，提供为 model 和 optimization 通用的抽象和算法，实现了一些常用的模型，一些处理数据的工具。

它还提供了一个比较方便的安装方式Learn

17.2 RL 相关的包

17.2.1 Overview

除了核心的一些包，比如LearnBase, LossFunctions, ObjectiveFunctions, PenaltyFunctions等等，以及一些算法 (LearningStrategies, StochasticOptimization等)、一些工具（比如画图、数据处理等等），JuliaML 还提供三个关于强化学习的包。

- Reinforce
  使用 julia 写的关于强化学习的抽象、算法、一些 utilities. 是 JuliaML 关于 RL 的主要功能实现，下面会详细说明。
- OpenAIGym
  将 `OpenAIGym` 包装，能够作为 `Reinforce.jl` 的环境。
  深入源码其实可以发现，这个包相关文件只有一个 `OpenAIGym.jl`, 所做的工作也是与 `Python` 语言接合，相当于用 `Julia` 封装了 `Python`。仅此而已。
- AtariAlgos
  包装 `Atari` 作为 `Reinforce.jl` 环境。
  基本同上，只不过这里封装的是ArcadeLearningEnvironment, 后者也是一个 `julia` 库，而且封装的是ArcadeLearningEnvironment

可以看到的是，关于 RL 中的环境，这个包系列实现的很少，基本上都是封装其他语言的。下面详细说明。

17.2.2 Reinforce.jl

主要实现了 RL 的接口，目的是为模块化的 env、policy、solver 提供一个简单的接口。这个包相关的文档很少，下面简单说明一下。

Env 接口 新的环境需要作为 `AbstractEnvironment` 的子类型来实现，这个抽象类型是在 `Reinforce.jl` 中定义的。然后需要实现几个方法：

- `reset!(env) -> env`
- `actions(env, s) -> A`
- `step!(env, s, a) -> (r, s')`
- `finished(env, s') -> Bool`

以及可选的几个覆盖：

- `state(env) -> s`

- `reward(env) -> r`

另外还有几个方法，比如 `ismdp(env) -> Bool` 用于判断是否为完全观察的 MDP(Markov Decision Process).

最后还有两个判断是否结束的 `maxsteps() || finished()` 方法。

官方给出的实现里面只有 4 个,在 `envs` 目录下,`cartpole.jl,mountain_car.jl,multi-armed-bandit.jl,pendulum.j` 全部是用上面给出的子类型方法实现的。代码量不多。

Policy 接口 Agent/Policy 的实现需要作为 `AbstractPolicy` 的子类型，然后实现 `action` 方法。内置的一个随机 policy 是一个案例:

```
struct RandomPolicy <: AbstractPolicy end
action( ::RandomPolicy, r, s, A) = rand(A)
```

另外还需要实现 `reset!( ::AbstractPolicy) ->`

官方主要实现了两个 `actor_critic, online_gae`

Episode Iterator 这个迭代器用于遍历 epsodes. 每一步都会返回一个四元组 `(s,a,r,s')`

```
ep = Episode(env, )
for (s, a, r, s ) in ep
    # do some custom processing of the sars-tuple
end
R = ep.total_reward
T = ep.niter
```

另外还有一个比较方便的方法，下面与上面等价

```
R = run_episode(env, ) do
    # anything you want... this section is called after each step
end
```

Part VI

论文篇

主要调研者　　何理扬、俞晨东

18 2018 Paper

18.1 OOPSLA2018: Julia

The Julia programming language aims to decrease the gap between productivity and performance languages. On one hand, it provides productivity features like dynamic typing, garbage collection, and multiple dispatch. On the other, it has a type-specializing just-in-time compiler and lets programmers control the layout of data structure in memory. Julia, therefore, promises scientific programmers the ease of a productivity language at the speed of a performance language. Julia 语言旨在缩小性能语言（C、C++、Fortran）和生产力语言（如 Python、MATLAB、R）之间的差距。一方面，它提供像动态定型、垃圾收集、多分派等提高生产力的特性；另一方面它有类型特化的即时编译器，并让程序员控制数据结构在内存中的布局。因此，Julia 向科学计算程序员承诺以高性能语言的速度获得高生产力语言的简易性。

The key to performance in Julia lies in the synergy between language design, implementation techniques and programming style. Julia 的性能关键在于语言设计、实现技术和编程风格之间的协同作用。

18.1.1 Julia 的高效性

The key to performance in Julia lies in the synergy between language design, implementation techniques and programming style. Julia 的高效性取决于几个方面的协调配合，分别是语言设计（Language design）、实现的技术（implementation techniques）以及编程的方法（programming style）：

- Language design: 正如所提到的，Julia 在语言设计上吸收了其他语言的很多特点，比如动态类型、可选的类型注释（optional type annotations）、反射、垃圾回收机制、多重分派、抽象类型等
- Performance does not arise from great feats of compiler engineering: Julia’ s implementation is simpler than that of many dynamic languages. The Julia compiler has three main optimizations that are performed on a high-level intermediate representation; native code generation is then delegated to the LLVM compiler infrastructure. The optimizations performed in Julia are (1) method inlining which devirtualizes multi-dispatched calls and inline the call target; (2) object unboxing to avoid heap allocation; and (3) method specialization where code is special-cased to its actual argument types. Language implementation:Julia 的编译器相比其他动态语言的编译器来说要简单很多，它并没有为 Julia 的高效性作出很大的贡献，但它也做了一些优化，主要有三点，分别是方法内联、对象拆箱以避免堆分配以及方法特化。The synergy between language design and implementation is in evidence in the interaction between the three optimizations. Each call to a function that has, as arguments, a combination of concrete types not observed before triggers specialization. A data-flow analysis algorithm uses the type of the arguments (and if these are user-de ned types, the declared type of their elds) to approximate the types of all variables in the specialized function. This enables both unboxing and inlining. The specialized method is added to the function’ s dispatch table so that future calls with the same combination of argument types can reuse the generated code. 语言设计和实现技术上的协调配合就体现在这三种优化中，每次调用一个函数作为参数时，如果它具有之前未观察到的具体类型的组合，就会触发特化 (specialization)，一个数据流分析算法使用参数的类型来接近专用函数中所有变量的类型，这样就能实现拆箱和内联，然后特化的函数被加入到一个函数分派表中，当之后继续调用这个函数且参数类型相同时，就可以复用之前生成的代码。
- Programmers are keenly aware of the optimizations that the compiler performs and shape their code accordingly. For instance, adding type annotations to elds of datatypes is viewed as good practice as it provides information to the compiler to estimate the size of instances and may allow unboxing. Another good practice is to write methods that are type stable. A method is type stable if, when it is specialized to a set of concrete types, data- ow analysis can assign concrete types to all variables in the function. Programming style: 为了写出高效的代码，编写代码的人也需要掌握一定的技巧来配合 Julia 的编译器，比如加入类型注释来为编译器提供更多的信息，也比如编写类型稳定 (type stable) 的方法，在 Julia 文档中类型稳定的解释是: 输出的类型是可以由输入类型预测出来

18.1.2 性能

Fig.41给出了几种语言实现所花费的人年。

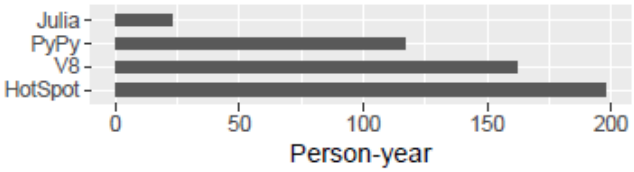


图 41: Time spent on implementations

为评测语言的相对性能，选择PLBG中的用 C、JavaScript、Python 分别实现的 10 个小程序，另外由 Julia 开发者编写相应的 Julia 版本。使用 PLBG 的方法，测量去除注释和重复的空白符后的代码的最小 Gzip 压缩包的大小，Julia 的有 6KB，JavaScript 的有 7.4KB, Python 有 8.3KB, C 有 14.2KB。Fig.42比较了 4 种语言相比 C 语言版本的规范化运行时间。测试使用的相关编译器版本是 Julia v0.6.2、CPython 3.5.3、V8/Node.js v8.11.1 和 GCC 6.3.0 -O2，评测的机器是 Intel i7-950 3.07GHz, 10GB RAM，操作系统是 Debian 9.4。结果表明，Julia 的性能胜过 Python 和 JavaScript(spectralnorm 除外)；多数程序的 Julia 版本的运行时间是 C 版本的 2 倍以内，**性能下降的原因可能是内存操作**（Julia 靠垃圾收集器管理内存）。Julia 禁止像 C 那样显式管理内存，它在堆中分配结构，栈分配仅用在有限的情况下，不允许指针算术运算。

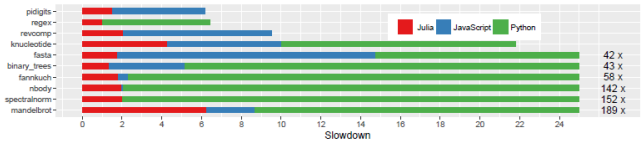


图 42: Slowdown of Julia, JavaScript, and Python relative to C

18.1.3 Julia 语言

Values Type Declarations Type Annotations Subtyping. Dynamically-checked Type Assertions. Multiple Dispatch. Metaprogramming: Macros, Reflection, Epochs.

Object oriented programming. Julia 不支持基于类的面向对象编程风格，缺少封装机制。

Functional programming. Julia 支持高阶函数、immutable-by-default values，但是没有 arrow types.

Gradual typing.

18.1.4 Julia 的实现

方法特化 (Method Specialization). Julia' s compilation strategy is built on runtime type information. Every time a method is called with a new tuple of argument types, it is specialized to these types. Optimizing methods at invocation time, rather than ahead of time, provides the JIT with key pieces of information: the memory layout of all arguments is known, allowing for unboxing and direct eld access. Specialization, in turn, allows for devirtualization. Devirtualization replaces method dispatch with direct calls to a specialized method. This reduces dispatch overhead and enables inlining. As the compilation process is rather slow, results are cached, thus methods are only compiled the rst time they are called with a new type. This process converges as long as functions are only called with a limited number of types. If a function gets called with many di erent argument types, then invocations will repeatedly incur the cost of specialization. Julia cannot avoid this pathology, as programs that generate a large number of call signatures are easy to write. To alleviate this problem, Julia allows tuple types to contain a Vararg component, which is treated as having type Any. Likewise, each function value has its own type, but Julia only specializes on function types if the argument is called in

the method body. Other heuristics are used for type Type. Julia has one recourse against type unstable code, programmers can use the @nospecialize annotation to prevent specialization on a speci c argument. Julia 的编译策略基于运行时的类型信息，每次使用新的参数类型去使用方法时，就特化这些参数的类型，而且特化使得可以实现虚化 (devirtualization)，虚化将方法调度替换为特化方法的直接调用，这能减少调度开销并实现内联。因为编译过程相当慢，所以结果会被缓存，因此只有第一次使用新的类型去调用方法时才会编译方法。这意味着只有使用有限数量的类型去调用函数这个过程才会收敛，如果使用许多不同类型的参数去调用函数，就会反复产生特化成本。而且 Julia 没办法彻底解决这个问题（这种函数很容易编写），为了缓解这个问题 Julia 也做了允许每组类型包含一个 Vararg 组件，该组件被视为具有 Any 类型。

方法内联 (Method Inlining). In Julia, it can be realized in a very e cient way because of its synergy with specialization and type inference. Indeed, if the body of a method is type stable, then the internal calls can be inlined. Conversely, inlining can help type inference because it gives additional context. For instance, inlined code can avoid branches that can be eliminated as dead code, which allows in turn to propagate more precise type information. Yet, the memory cost incurred by inlining can be sometimes prohibitive; moreover it requires additional compilation time. As a consequence, inlining is bounded by a number of pragmatic heuristics. 在 Julia 中，特化和类型推断的特点使得方法内联能以一种十分有效的方式进行。因为如果方法主体是类型稳定的，那么内部的调用就可以内联，另一方面内联又可以帮助类型推断因为它提供了额外的上下文信息。例如内联可以避免被作为死代码消除的分支，这反过来就能传播更精确的类型信息。当然内联会产生一定的内存开销，而且它会需要额外的编译时间，因此使用启发式的方法来决定在哪些地方限制内联。

对象拆箱 (Object Unboxing) Since Julia is dynamic, a variable may hold values of many types. As a consequence, in the general case, values are allocated on the heap with a tag that speci es their type. Unboxing allows to manipulate values directly. This optimization is helped by a combination of design choices. First, since concrete types are nal, a concrete type speci es both the size of a value and its layout. This would not be the case in Java or TypeScript due to subtyping. In addition, Julia does not have a null value; if it did, there would be need for an extra tag for primitive values. As a consequence, values such as integers and oats can always be stored unboxed. Repeated boxing and unboxing can be expensive, and unboxing can also be impossible to realize although the type information is present, in particular for recursive data structures. As with inlining, heuristics are thus used to determine when to perform this optimization. 因为 Julia 是动态类型语言，所以一个变量可以有多种类型，因此在通常情况下，被分配在堆上的值会有一个标记来指定它的类型。拆箱允许直接操作值，这源于 Julia 的设计，因为具体类型是明确的，所以具体类型会指定值的大小和布局。此外 Julia 没有 null 值，因为如果有的话就需要为原始值添加额外标记。因此像 integer 和 float 之类的值可以以未装箱 (unboxed) 的方式存储。重复装箱和拆箱可能是昂贵的，并且有时候即使存在类型信息也无法实现拆箱，特别是对于递归数据结构，因此和内联一样，使用启发式的方法来决定何时来进行优化。

18.2 arXiv1810: Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs

INTRODUCTION

Google has now made TPUs available for general use on their cloud platform and as of very recently has opened them up further to allow use by non-TensorFlow frontends. We describe a method and implementation for offloading suitable sections of Julia programs to TPUs via this new API and the Google XLA compiler. Google 提供了 API 接口，使得我们可以使用非 Tensorflow 的前端并利用其 API 接口使用它的 TPU 资源。在本文中，提出了**如何通过使用 API 和 Google 的 XLA 编译器将 Julia 代码部署到 TPU 上的方法**。

In particular, our approach allows users to take advantage of the full expressiveness of the Julia programming language in writing their models. This includes higher-level features such as multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers and generic linear algebra routines. 该方法**允许用户编写模型时充分利用 Julia 的语言特性**，包括上层的特性比如多分派、高阶函数、已有的库（微分求解库和通用线性代数库）

Since it operates on pure Julia code, it is also compatible with the Zygote.jl automatic differentiation tool, which performs automatic differentiation as a high-level compiler pass. Putting these together, we are able to compile full machine learning models written using the Flux machine learning framework, fusing the forward and backwards model passes as well as the training loop into a single executable that is offloaded to the TPU
结合 Zygote.jl 自动微分工具，我们用 Flux 框架编写的机器学习模型的前向和后向参数传播以及训练循环部署到单个的 TPU 上

18.2.1 TPU 系统架构

与大部分硬件加速不同，TPU 云计算平台不能直接提供 PCIe 给用户，所以 Google 通过提供名为 XLA 的中间代码表示，使得非 Tensorflow 的用户可以通过 API 构建 XLA IR 来使用 TPU 资源。

XLA(“Accelerated Linear Algebra”) 是 Google 的开源编译项目，具有丰富的输入 IR，用于线性代数运算，并为它生成后端的 CPU、GPU 和 TPU 代码

XLA 的输入 IR 称为 HLO(High-Level Optimization IR)，能够对基本数据类型和数组进行操作运算。HLO 操作包括基本算术运算，特殊函数，广义线性代数运算，高级数组运算以及分布式计算的基元。HLO 操作数分为静态操作数和动态操作数两种

- **Static Operands** whose values need to be available at compile time and that configure the operation (e.g. specifying the window of a convolution operation or the summation order of a tensor contraction). Additionally some of these static operands may reference other computations that are part of the same HLO module
- **Dynamic Operands** consisting of the aforementioned tensors (as output by other HLO operations). While the array entries of the tensor need not be available at compile time, the shape and layout of any tensor does need to be defined at compile time (i.e. there is no facility for dynamically sized dimensions).

18.2.2 Julia 编译

将 Julia 代码编译到 XLA IR 表示，需要对 Julia 的编译有一定了解。Julia 的后端是 LLVM，所以 Julia 的编译器需要在语言的动态语义和 LLVM 的静态语义间建立桥梁，主要有四部分（The dynamic semantics, the embedding of the static compiler intrinsics, interprocedural type inference and the extraction of static sub graphs）

- **Dynamic semantics**
- **Embedding of the static compiler intrinsics**
- **Interprocedural type**
- **The extraction of static sub graphs**

18.2.3 嵌入 XLA

To compile to XLA instead of LLVM, we apply the exact strategy as outlined in the previous section. In fact, we can re-use most of the compiler itself (in particular all of type inference and all mid-level optimization passes). 在使用 XLA 替换底层 LLVM 的时候，可以重用很多 Julia 编译器的内容（比如所有的类型推断以及中间层表示）

1. **Tensor representation:**
   Owing to its heritage as a teaching and research language for linear algebra, Julia has a very rich hierarchy of array abstractions.We thus embed XLA values by defining a runtime structure corresponding to immutable, shaped, N-dimensional tensors backed by handles to remote, XRT-managed memory(Fig1).
   为了与 Julia 的不可变 N 维数组对应，构造一个能处理不可变 N 维数组对应的结构
   Once we have successfully defined this representation, we are done in terms of semantics (on top of the base julia semantics, but largely replacing the LLVM-derived semantics).

```
const AA{T, N} = AbstractArray{T, N}
struct XRTArray{T, Shp, N} <: AA{T, N}
  storage::XRTAllocation
  # XRTArrays are constructable by
  # conversion from regular arrays
  function XRTArray(
        a::Array{T, N}) where {T, N}
    new{T, size(A), N}(transfer(a))
  end
  # XRTArrays are constructable from a
  # remote memory allocation if
  # (T, Dims, N) are specified
  function XRTArray{T, Dims, N}(
      a::XRTAllocation) where {T, Dims, N}
    new{T, Dims, N}(a)
  end
end
```

Listing 1: The definition of an XRTArray[3]. T is the element type of the array, `Shp` is a tuple of integers describing the shape of the tensor, `N` is always equal to `length(Shp)`, which is enforced by the constructor and is required because Julia does not currently allow computation in the subtype relation. The `AA` alias is defined for brevity of notation only.

图 43: Struct defined for julia's immutable array

2. **Operation representation:** HLO operands are partitioned into static and dynamic operands. Suppose we have an example XLA operation ‘Foo ‘taking one static operand (e.g. a single integer) and two dynamic operands. We would declare this embedding as follows: 定义自己的蕴含静态和动态操作数的结构，包含静态操作数和作用于之前定义的 XRTArray 上的动态操作数

```
struct HloFoo <: HloOp{:foo}
  static_operand::Int
end

function (hlo::HloFoo)(dop1::XRTArray,
                       dop2::XRTArray)
    execute(hlo, dynamic_op1, dynamic_op2)
end
```

图 44: An example about HLO

接着将其直接映射为对应的 AST。In the example in Fig.2, we have spliced the HLO operands (including the static operands) right into the AST.

If we can convince the Julia compiler to generate IR of this form, generating equivalent XLA IR becomes trivial (a simple rewriting from one representation to another).

函数在 XLA 中通常是比较容易表示的。In practice the called functions tend to be very simple and generally representable in XLA, so we may obtain HLO for the called function simply by recursively invoking the compiler

3. **Shape Transfer Functions** One additional consideration is the need for type inference to be able to statically understand the shape of each HLO operation (i.e. how the output shape depends on the input shapes). 为了类型推断，我们需要理解在 HLO 中的输入输入关系。In essence, we need the equivalent of

```
# An HLO operand that generates a random
# uniform random number of the specificed
# shape and element type:
struct HloRng <: HloOp{:rng}
    Type
    Shape
end

"""A function that adds random numbers to
   each entry of a 1000x1000 matrix"""
@eval function add_rand_1000x1000(
    A::XRTArray{Float32, (1000, 1000), 2}
    random = $(HloRng(Float32,
                (1000, 1000)))()
    result = $(HloAdd())(random, A)
    return result
end
```

Listing 2: A manually constructed XLA embedding. `HloRng` is slightly simplified from the actual operation for clarity of presentation. The boilerplate definition calling execute is omitted.

图 45: Mapping HLO example to AST

the type transfer functions we had for LLVM IR. Luckily, since all HLO operations are precisely inferable over the type lattice (as opposed to the inference lattice) there is no need to add these transfer functions to the compiler.

```
execute(op::HloOp, args::XRTArray...) =
  _execute(op, args...)::shape_infer(op,
    map(typeof, args)...)
```

图 46: HLO TypeCheck

18.2.4 将 Julia 转换为 XLA

只要 Julia 程序是按照 XLA 基元来编写的，就能将其编译到 XLA。然而，Julia 程序不是根据晦涩难懂的 HLO 操作来编写的，而是根据由 Julia 基本库提供的函数和抽象来编写的。幸运的是，Julia 使用了多重派发，使得根据 HLO 操作来表达标准库的抽象变得容易。

```
# Matrix-Matrix and Matrix-Vector product
function Base.:*(A::XRTMatrix,
    B::Union{XRTMatrix, XRTArray})
  ddots = DimNums((1,), (0,), (), ())
  HloDot(ddots)(A, B)
end
Base.transpose(A::XRTArray) =
    HloTranspose((1,0))(A)
# Scalar addition
Base.:+(A::XRTArray{T, (), 0},
        B::XRTArray{T, (), 0})
    where {T<:XLAScalar} =
 GenericHloOp{:add}(T, ())(A, B)
```

除了这些简单的操作以外，还提供了高级数组抽象的实现，尤其是 mapreduce 和 broadcast。

```
# Matrix-Matrix and Matrix-Vector product
function Base.:*(A::XRTMatrix,
    B::Union{XRTMatrix, XRTArray})
  ddots = DimNums((1,), (0,), (), ())
  HloDot(ddots)(A, B)
end
Base.transpose(A::XRTArray) =
    HloTranspose((1,0))(A)
# Scalar addition
Base.:+(A::XRTArray{T, (), 0},
        B::XRTArray{T, (), 0})
    where {T<:XLAScalar} =
 GenericHloOp{:add}(T, ())(A, B)
```

从上图可以看到将任意 Julia 函数作为静态计算运算的效果。由于 Julia 对泛型抽象的依赖，它只需指定极少数定义，就能覆盖大量 API。具体来说，从 mapreduce 的定义中，我们可以自动得到在 base 中所定义运算（如 sum 和 prod）的降维。事实上，获取足够的 API 覆盖来编译 VGG19 模型的前向传播和反向传播需要不到 200 行定义。

主要的映射可以分为两种：结构映射和控制流映射。在控制流映射中需要解决 Julia 控制流和 XLA 提供的控制流语义不匹配的问题。One additional complication we have not yet discussed is the semantic mismatch between the imperative control flow offered by Julia and the functional control flow offered by XLA.

18.2.5 结果

对 VCG19 的前向传播进行评测，结果如下:

N=	1	10	100
Flux CPU	0.79s	6.67s	52.4s
PyTorch CPU	1.16s	9.55s	93.0s
FluXLA CPU	12.06s	64.8s	> 600s
FluXLA TPU (total)	0.86s	0.74s	0.93s
FluXLA TPU (compute)	0.02s	0.04s	0.23s

图 47: VCG19 Forword Pass

不同批大小对应的 VGG19 前向传播时长。Flux CPU 是 Flux master/Julia master，但不使用 XLA 编译器。PyTorch CPU 是同一 CPU 上的相同 PyTorch 模型。FluXLA CPU 是我们的研究在 CPU 上的 xrt 实现；FluXLA TPU (total) 是端到端时间，和客户端报告的时间一致（包括 kernel launch 开销和从谷歌云把数据迁移回来，注意由于额外的网络迁移，该测量结果会出现极大的变动）；FluXLA TPU (compute) 是 TPU 上的总计算时间，和云分析器报告的时间一致（与 FluXLA TPU (total) 不同，该测量很稳定）。所有 CPU 测量基于支持 AVX512 的 Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz CPU。可获取高达 20 个内核，且 CPU 基准不限于单个内核（即使在实践中，也不是所有 CPU 基准都使用并行化）。TPU 基准仅限单个 TPU 内核。所有时间至少经过 4 次运行（除了 FluXLA CPU for N=100，因为它无法在 10 分钟内完成一次运行）。

[3](arXiv1810)

HLO Instruction Kind	Instruction count							
	Forwards				Backwards			
	Unopt		Opt		Unopt		Opt	
	E	T	E	T	E	T	E	T
parameter	2	56	2	52	2	508	2	183
constant	6	24	2	20	31	583	3	93
get-tuple-element	58	98	58	58	58	181	58	58
add	0	20	1	20	0	304	1	114
reshape	33	33	4	4	122	122	58	58
map	22	22	19	19	119	119	38	38
multiply	0	0	0	0	0	256	18	56
tuple	0	20	0	0	21	189	21	21
convolution	16	16	16	16	48	48	47	47
transpose	17	17	1	1	59	59	43	43
broadcast	20	20	17	17	62	62	19	19
less-than	0	0	0	0	0	108	0	36
maximum	0	23	0	23	0	59	0	23
conditional	0	0	0	0	0	108	0	0
reduce	1	1	1	1	20	20	18	18
select	0	0	0	0	0	0	0	72
reverse	0	0	0	0	16	16	16	16
dot	3	3	3	3	9	9	9	9
reduce-window	5	5	5	5	5	5	5	5
select-and-scatter	0	0	0	0	5	5	5	5
exponential	0	2	1	2	0	5	1	3
less-than-or-equal-to	0	0	0	0	0	5	0	5
divide	0	1	0	1	0	3	0	2
subtract	0	0	0	0	0	1	0	1
Total	183	361	130	242	577	2775	362	925

参考

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[2] Jeff Bezanson, Jiahao Chen, and other. Julia: Dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA):120:1–120:23, October 2018.

[3] Keno Fischer and Elliot Saba. Automatic full compilation of julia programs and ML models to cloud tpus. *CoRR*, abs/1810.09868, 2018.

[4] Philipp Moritz, Robert Nishihara, et al. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577, Carlsbad, CA, 2018. USENIX Association.