编译大作业第二部分实现说明

贾云杉 雷博涵 贾瑞琪 林文心

2020.06.18

问题概述

本次project的问题为，根据json文件中给定的表达式信息，生成loss对某个输出的的导函数，并将导函数展开成逻辑正确的.cc函数，写入对应文件夹。json文件针对每一个样例的描述包含输入输出，数据类型，运算表达式和求导对象。

解决思路

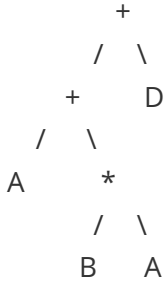
由于在project1中，我们已经能够实现：给定表达式，生成其对应的语法树，重新构建循环结构，推测循环变量的范围，并输出最终的c代码。在此次project中，我们依然可以沿用上述功能，并且加入新的改动实现求导功能。
具体实现逻辑如下：

kernel的构建

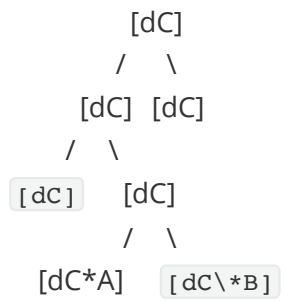
首先读入json文件，根据文件内的信息构建`kernel`结点，创建AST，并且记录需要被求导的变量。

计算梯度

在计算梯度时，我们采用按照AST的结构，自顶向下传递梯度，最后在叶子节点收集所有梯度形成求导表达式的方法。主要逻辑如下：
当遇到加法时，由于$d(A+B)=dA+dB$，需要把父亲节点的梯度复制两份分别传递给两个子节点。
当遇到乘法时，由Leibniz法则$d(AB)=dA*B+dB*A$，需要把父亲节点的梯度乘以子节点的兄弟节点作为子节点的梯度。
当以上梯度传播完毕后，假设原式对A求导，则需将原AST中A这一项对应的节点梯度相加，所得表达式即为A所对应的梯度表达式。
下面用一个例子说明实现的具体过程：
$C=A+B*A+D$ (其中D为与A无关的项)，设对A求导。
该函数对应的AST为：



根据上述求导法则，梯度传递形成的树为：



其中，阴影部分为梯度传递树中与原AST的A叶子节点对应的梯度，将其相加，即可得结果：
$dA[i,k] = \frac{\partial loss}{\partial A[i,k]} = \sum_j \frac{\partial loss}{\partial C[i,j]} \times \frac{\partial C[i,j]}{\partial A[i,k]}$

调整输入和输出

在计算梯度后，`gradcase`需要的输入和输出和原json文件不相同，因此需要对其进行调整。
输出修改为需要被求导的变量的导数形式。例如对A求导，输出应为dA。
输入修改为求导过程中需要用到的全部变量，包括原输入和求导过程中运用到的所有导数形式。

循环变量边界推断和IR输出

此部分基本都可复用project1的代码，只是在接口部分做了修改。
`BoundInfer`增加了直接对`Stmt`进行边界推断的接口函数`loop_bound_infer`。
`IRCCPrinter`只是将原来打印的`#include "../run.h"`变成打印`#include "../run2.h"`。

梯度计算

我们首先从`generateAST`的返回值得到求导的对象，接着在`generateAST`返回的`kernel`结点的基础上，保留AST的架构，自顶向下按求导的规则对每个分量求导，最后收集所有对应求导对象的叶子结点求出的梯度并相加，就可以得到求导的结果，并且根据求导的结果构造新的`kernel`结点并返回。

具体实现

构造AST

在project1的基础上，读入`grad_to`对应的内容，存储在`gvec`中，传递给后续计算。

计算梯度

下面具体说明实现求导过程的类和函数：
首先是一个`IRVisitor`的派生类`getGradVec`，用于上述自顶向下的求导过程。
该类重写了部分visit函数，首先对`MOVE`节点进行visit时，由链式法则可知，将根节点的梯度初始化为`d+MOVE`语句的`dst`。

```
 void visit(Ref<const Move> op) override {
        auto dst = op->dst.as<Var>();
        CHECK(dst, "inter");
        grad_inter[op->src] = Var::make(dst->type(),
                                        "d" + dst->name,
                                        dst->args,
                                        dst->shape);
        (op->src).visit_expr(this);
        return;
    }
```

对`Binary`节点进行visit时，根据运算类型的不同，做不同的处理，对于加法，将父节点的梯度分裂传给两个子节点，不考虑减法情况，对于乘法，每个子节点都接收父节点的梯度并乘兄弟节点对应值，对于除法，只考虑对被除数求导的情况，即对a/b，a的梯度是父节点的梯度除以b。

```
void visit(Ref<const Binary> op) override {
        switch (op->op_type)
        {
        case BinaryOpType::Add :
            grad_inter[op->a] = grad_inter[Expr(op)];
            grad_inter[op->b] = grad_inter[Expr(op)];
            break;

        case BinaryOpType::Sub :
            CHECK(false, "didn't impl sub operation");
            break;

        case BinaryOpType::Mul :
            grad_inter[op->a] = Binary::make(op->type(),
                                             BinaryOpType::Mul,
                                             grad_inter[Expr(op)],
                                             op->b);
            grad_inter[op->b] =  Binary::make(op->type(),
                                             BinaryOpType::Mul,
                                             grad_inter[Expr(op)],
                                             op->a);
            break;

        case BinaryOpType::Div :
            CHECK(op->b.as<IntImm>() || op->b.as<FloatImm>(), "can't
support this");
            grad_inter[op->a] = Binary::make(op->type(),
                                             BinaryOpType::Div,
                                             grad_inter[Expr(op)],
                                             op->b);
            break;

        default:
```

```
            CHECK(false, "didn't impl other operation");
        }
        op->a.visit_expr(this);
        op->b.visit_expr(this);
        return;
    }
```

接着为了收集所有对应求导对象的叶子节点，重写对于`Var`节点的visit函数，若遇到对应求导对象`gradStr`的叶子节点，则将`d+gradStr`加入`gradArg`，将该节点对应的梯度加入`gradVec`。

```
void visit(Ref<const Var> op) override {
        if(op->name == gradStr){
            gradArg.push_back(Var::make(op->type(),
                                        "d" + op->name,
                                        op->args,
                                        op->shape));
            gradVec.push_back(grad_inter[Expr(op)]);
        }
        return;
    }
```

一个接口函数`toGrad`，其参数为`generateAST`返回的包含求导对象名字的vector和生成的`kernel`节点，该函数对`kernel`节点中的每一句`for_stmt`，通过`getGradVec`类的遍历得到所有梯度，并生成`MOVE`语句累加每个叶子节点的梯度得到求导的结果，从而得到一个用于求导的语句列表`updated_stmt_list`,最后根据新的语句列表生成一个新的`kernel`节点返回。

```
Group toGrad(const vector<string> & gradient_vec, const Group &
origin_kernel){
    vector<Stmt> updated_stmt_list;
    auto kernel = origin_kernel.as<Kernel>();
    CHECK(kernel, "internal error");
    for(auto i : kernel->stmt_list){
        auto for_stmt = i.as<LoopNest>();
        CHECK(for_stmt, "internal error");
        CHECK(for_stmt->body_list.size() == 1, "internal error");
        for(string grad : gradient_vec){
            getGradVec gv(grad);
            for_stmt->body_list[0].visit_stmt(&gv);
            size_t size = gv.gradVec.size();
            for(size_t j = 0; j < size; ++j){
                updated_stmt_list.push_back(LoopNest::make(for_stmt-
>index_list,

{Move::make(gv.gradArg[j],

Binary::make(gv.gradArg[j].type(),

         BinaryOpType::Add,
```

```
            gv.gradArg[j],
            gv.gradVec[j]),
MoveType::LocalToLocal)}));
                }
            }
        }

        return Kernel::make(kernel->name, kernel->inputs, kernel->outputs,
                            updated_stmt_list, kernel->kernel_type);
}
```

调整输入输出

由于求导函数`grad_case`的输入输出和原始json文件中的输入输出不同，因此通过`IRIOMutator`调整输入和输出。
首先，确定新的输出，`MOVE`语句的`dst`参数就是`grad_case`的输出。
为了确定新的输入，对`MOVE`语句的`src->b`进行遍历，即遍历`Binary`节点中的右项。只遍历`Binary`节点右项的原因是生成`MOVE`节点累加梯度时的格式是`grad = grad+每个子节点的梯度`，因此所有计算导数需要的输入都只在`Binary`的右项里。收集所有在`src->b`中出现的`Var`加入`refvars`，`refvars`中为求导用到的所有变量。

```
Expr IRIOMutator::visit(Ref<const Var> op) {
    refvars.insert(op->name);
    return op;
}
```

接着将原有的`inputs`与`refvars`进行比较，若原有的`input`中的变量或其梯度在`refvars`，则该变量或梯度也会在新的输入`new_inputs`中：

```
for (auto expr : op->inputs) {
        Ref<const Var> arg = expr.as<Var>();
        CHECK(arg.defined(), "one input arg is not of Var type");

        if (refvars.find(arg->name) != refvars.end()) {
            new_inputs.push_back(expr);
        }
    }
    for (auto expr : op->inputs) {
        Ref<const Var> arg = expr.as<Var>();
        CHECK(arg.defined(), "internal error\n");

        if (refvars.find("d" + arg->name) != refvars.end()) {
            new_inputs.push_back(Var::make(arg->type(), "d" + arg->name,
arg->args, arg->shape));
```

```
        }
    }
```

将原来的`outputs`与`refvars`进行比较，若原来`outputs`中的变量的梯度在`refvars`中，也要将该梯度加入新的输入`new_inputs`中：

```
for (auto expr : op->outputs) {
        Ref<const Var> arg = expr.as<Var>();
        CHECK(arg.defined(), "internal error\n");

        if (refvars.find("d" + arg->name) != refvars.end()) {
            new_inputs.push_back(Var::make(arg->type(), "d" + arg->name,
arg->args, arg->shape));
        }
    }
```

从`run2.cc`可知，新的输入参数的顺序是`原有的input|原有的input的梯度|原来未知的梯度(原有的output的梯度)`，上面得到`new_inputs`的顺序与其一致。
最后根据新的输入输出返回新的`kernel`节点：

```
return Kernel::make(op->name, new_inputs, grad_vec, op->stmt_list, op-
>kernel_type);
```

成员分工

实现部分：贾瑞琪和林文心负责修改`generateAST`的接口，读入`grad_to`对应的内容。贾云杉负责`toGrad`函数，计算原表达式的梯度，并且修改了`BoundInfer`和`IRCCPrinter`的接口。雷博涵负责`IRIOMutator`，更新输入和输出。
报告部分：由贾瑞琪和林文心共同完成。