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(其中D为与A无关的项)：

> **例子** 若要求C=A + B * A + D关于A的导数。

该函数对应的AST为：

```
        +
      /   \
     +     D
   /   \
  A     *
      /   \
     B     A
```

根据上述求导法则，梯度传递形成的树为：

```
          [dC]
        /     \
     [dC]     [dC]
    /    \
 [dC]    [dC]
       /     \
   [dC*A]  [dC*B]
```

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：
$dA[i,k] = \frac{\partial loss}{\partial A[i,k]} = \sum_j \frac{\partial loss}{\partial C[i,j]} \times \frac{\partial C[i,j]}{\partial A[i,k]}$
总之，计算梯度的简要流程如下：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。

坐标变换

我们用两个例子说明坐标变换是如何进行的：

> **例子** 若我们上面的求导器生成了一条语句 `dA[(i+1)+1][j+1] = dB[i][j]`，要求对 `dA[(i+1)+1, j+1]` 进行坐标变换，得到符合要求的形式。

我们先访问 `dA` 的第一个下标，即表达式 `(i+1)+1`。在进行操作前，我们先新定义一个变量，名为 `x`，并认为它满足 `x = (i+1)+1`。整个下标变换的过程就是试图将 `i` 用 `x` 的表达式表示出来。表示过程如下述：

```
(i + 1) + 1 = x,
i + 1 = x - 1,
i = x - 1 - 1
```

我们通过遍历 `(i + 1) + 1` 的语法树，逐层作逆运算，这样把 `i` 表示成了 `x - 1 - 1`。我们不希望再引入一个新的变量，把 `x` 仍用 `i` 表示，并将整条语句中的 `i` 全部换成 `i - 1 - 1`。`j` 的情况与 `i` 类似。

注意到，我们上面只用了加法做例子。乘法、除法和取模的情况比较麻烦。先说除法和取模的情况。整数除法和取模这两种运算均没有确定的逆运算。如果我们有这样一条语句 `dA[i/16] = dB[i]`，令 `x = i / 16`，我们并不能把 `i` 用某个确定的 `x` 的表达式表示出来，因为 `x` 和 `i` 是一对多的关系。尽管这样，我们有这样的关系式：`i = x * 16 + i % 16`。表达式的右边会被替换成 `i * 16 + i % 16`。进一步地，与上面的情况一样，语句右边出现的所有 `i` 都会被替换为 `i * 16 + i % 16`。这个式子，如果对两个 `i` 采取相同的理解，自然是有问题的。但是，我们并不在上述阶段处理取模的运算，取模运算放到最后单独处理，因此两个 `i` 可以被分开。

我们最后会扫描一遍整个式子，找出所有的取模运算，把取模运算换成另一个临时变量。比如，上面产生的 `i % 16`，就会被替换为临时变量。这样处理可以得到正确的表达式，但是会丢失关于 `i % 16` 与 `i` 之间关系的信息，从而可能得出错误的结果。但在这次作业的 Case 8 中，由于 `/` 和 `%` 各自提供的限制，变换后的 `i` 只能取 0 和 1，不会出现问题。

事实上，我们的坐标变换实现并不能处理乘法的情况。如果有一条语句 `dA[i*2] = dB[i]`，那么 `x = i * 2` 是有限制条件的，它必须是 2 的倍数。这样一来，我们需要增加条件判断。不过，由于下标的乘法运算不见于全部样例，我们可以不考虑这种情况。

调整输入和输出

在计算梯度后，`grad_case`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`等。
`IRCCPrinter`只是将原来打印的`#include "../run.h"`变成打印`#include "../run2.h"`。

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;
}
```

坐标变换

实现了IRMutator的派生类`Replace`和`IRVisitor`的派生类`FindNeedExpr`用于对`LHS`的下标索引中有运算的情况进行坐标的变换。
其中最主要实现变换的函数为`getReplace`,该函数会返回一个`pair`，`pair`中的第一个元素是要被替换的变量，`pair`中的第二个元素是由新变量表出原变量的表达式，其中新变量仍使用原变量的`Index`。
该函数中对不同的运算类型做不同的处理，如对于需要被替换的式子中运算为加法或减法或乘法的时候，生成逆运算（即加对应减，减对应加，乘对应除）的节点，对于除法节点，考虑整数除法的情况，会将例如`x=i/16`的表达式转换成`i=x*16+i%16`的形式。

```
pair<Expr, Expr> getReplace(Expr cal){
    Expr first = cal;
    Expr item = Index::make(Type::int_scalar(32), "*", Dom::make(Type::int_scalar(32), 0, 1), IndexType::Spatial);
    Expr second = item;
    while(first.as<Index>() == nullptr){
        auto wrapper = first.as<Binary>();
        CHECK(wrapper != nullptr, "index should be Index or BinaryOp");
        if(wrapper->a.node_type() == IRNodeType::IntImm){
            return pair<Expr, Expr>(Expr(), Expr());
        }
        first = wrapper->a;
        switch (wrapper->op_type)
        {
        case BinaryOpType::Add:
            second = Binary::make(first->type(), BinaryOpType::Sub, second, wrapper->b);
```

```
            break;
        case BinaryOpType::Sub:
            second = Binary::make(first->type(), BinaryOpType::Add, second, wrapper->b);
            break;
        case BinaryOpType::Mul:
            second = Binary::make(first->type(), BinaryOpType::Div, second, wrapper->b);
            break;
        case BinaryOpType::Div:
            second = Binary::make(first->type(),
                                  BinaryOpType::Add,
                                  Binary::make(first->type(),
                                               BinaryOpType::Mul,
                                               second,
                                               wrapper->b),
                                  Binary::make(first->type(),
                                               BinaryOpType::Mod,
                                               second,
                                               wrapper->b));
            break;
        default:
            // leave % later
            return pair<Expr, Expr>(Expr(), Expr());
        }
        return pair<Expr, Expr>(first, replace(second, item, first));
    }
```

接口函数

接口函数`toGrad`的参数为`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，然后调用坐标变换的过程将赋值语句左侧下标索引的运算消除。最后得到一个用于求导的语句列表`updated_stmt_list`,再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){
                std::vector<Expr> needHandle;
                for(Expr index : gv.gradArg[j].as<Var>()->args){
                    if(index.as<Index>() == nullptr){
                        needHandle.push_back(index);
                    }
                }
                Expr dst = gv.gradArg[j];
                Expr src = gv.gradVec[j];
                for(Expr index : needHandle){
                    pair<Expr, Expr> replaceItem = getReplace(index);
                    if(replaceItem.first.defined()){
                        dst = replace(dst, index, replaceItem.first);
                        src = replace(src, replaceItem.first, replaceItem.second);
                    }
                }
                src = Binary::make(dst.type(), BinaryOpType::Add, dst, src);
                Stmt move = Move::make(dst, src, MoveType::LocalToLocal);
                vector<Expr> need_replace = getNeedExpr(move);
                vector<Expr> index_list(for_stmt->index_list);
                size_t temp_size = need_replace.size();
                for(size_t k = 0 ; k < temp_size; ++k){
                    Expr com_index = Index::make(Type::int_scalar(32),
                                                 "com_" + std::to_string(k),
                                                 Dom::make(Type::int_scalar(32), 0, 1),
                                                 IndexType::Spatial);
```

```
                    index_list.push_back(com_index);
                    move = replace(move, need_replace[k], com_index);
                }
                updated_stmt_list.push_back(LoopNest::make(index_list, {move}));
            }
        }
    }
    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);
```

实验结果

我们最后通过了所有10个测试样例。

```
Random distribution ready
Case 1 Success!
Case 2 Success!
Case 3 Success!
Case 4 Success!
Case 5 Success!
Case 6 Success!
Case 7 Success!
Case 8 Success!
Case 9 Success!
Case 10 Success!
Totally pass 10 out of 10 cases.
Score is 15.
```

使用到的编译知识

进行词法分析、语法分析，构造抽象语法树。
语法制导翻译，遍历并变换语法树，在求导时将梯度作为继承属性自顶向下传播。

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`中计算原表达式的梯度的部分，并且修改了`BoundInfer`和`IRCCPrinter`的接口。雷博涵负责`IRIOMutator`，更新输入和输出。`toGrad`中变量替换部分由贾云杉和雷博涵共同完成。
报告部分：由贾瑞琪和林文心共同完成。