编译大作业2报告

1600012956 黄凯

1700012918 叶文沁

1700012979 刘皓华

1700012885 王胤博

小组分工

所有成员共同讨论求导技术的思路

黄凯：负责AST语法树的构建与变化代码

叶文沁：主要负责buildTable.h和solution2.cc的修改，主要负责最终代码的debug

刘皓华：负责IRprinter的修改，共同负责solution2.cc的修改，共同负责最终代码的debug

王胤博：阅读论文，撰写报告。

自动求导技术设计

设json文件的输入为output=f(input)，则根据链式法则，有

$$\mathrm{d}input = \mathrm{d}output * f'(input)$$

已知doutput，要求dinput，只需求f'(input)。我们编写了函数get_grad()来求f'(input)。该函数深度优先地遍历语法分析阶段得到的AST，然后在每个节点处计算对input的导数，最终得到f对input的导数。例如对于一个表示乘法的节点，设其左右子节点分别为left与right，导数分别为left'与right'，则此乘法节点的导数为left'*right+left*right'。get_grad()遍历AST的过程也是建立新AST的过程，新AST对应的表达式就是求导式dinput=doutput*f'(input)。得到求导式的AST后，利用IR产生IR树即可生成代码。

另外对于下标的线性变换，我们认为不对下标的线性变换进行处理，直接让最终的等式左侧使用线性变换的下标与run2.cc中的解法等价，因为正反向过程中节点的影响是相互对应的。

对于case10中同一个求导变量，不同下标的问题，可以通过多个求导等式(+=)来解决，但是我们没有时间实现这一思路。

实现流程与结果

我们首先如第一次作业一样，调用read_json解析json文件，然后调用parser为kernel构建AST。此后，对于grad_to中每一个要求导的参数变量，我们调用change_tree()来得到新的AST。change_tree的实现如下：

```
TreeNode* change_tree(TreeNode *root, std::string &name) {
    init(name);
    return get_grad(root);
}

TreeNode* get_grad(TreeNode *node) {
    TreeNode *left, *right;
```

```
    switch (node->type) {
        case Add:
            left = get_grad(node->left);
            right = get_grad(node->right);
            return sim_add(left, right);
        case Sub:
            left = get_grad(node->left);
            right = get_grad(node->right);
            return sim_sub(left, right);
        case Mul:
            TreeNode *ret;
            left = sim_mul(get_grad(node->left), node->right);
            right = sim_mul(node->left, get_grad(node->right));
            ret = sim_add(left, right);
            if (isEXPR(ret)) {
                return new_node(Par, ret, NULL);
            }
            else {
                return ret;
            }
        case Fdiv:
        case Idiv:
            left = sim_mul(get_grad(node->left), node->right);
            right = sim_mul(node->left, get_grad(node->right));
            left = sim_sub(left, right);
            if (isEXPR(left)) {
                left = new_node(Par, left, NULL);
            }
            right = sim_mul(node->right, node->right);
            if (isEXPR(right)) {
                right = new_node(Par, right, NULL);
            }
            return sim_div(left, right);
        case Eq:
            TreeNode *Dout, *Din;
            Dout = rename_node(node->left);
            right = get_grad(node->right);
            Din = rename_node(grad_node);
            return new_node(Eq, Din, sim_mul(Dout, right));
        case Com:
            left = get_grad(node->left);
            right = get_grad(node->right);
            return new_node(Com, left, right);
        case Par:
            left = get_grad(node->left);
            return new_node(Par, left, NULL);
        case Tref:
            if (strcmp(node->val.String, grad_name.c_str()) == 0) {
                if (grad_node == NULL) {
                    grad_node = node;
                }
                return one_node;
            }
            return zero_node;
        case Mod:
        case Sref:
        case Index:
        case Intv:
```

```
        case Floatv:
        default:
            return zero_node;
    }
}
```

然后对新得到的AST，我们修改了buidTable.h使其适应线性下标变换中存在求余计算时的index范围计算。代码如下：

```
const int mod_begin = 0x80000000;
const int mod_end = 0x7fffffff;
void Index_Mod(TreeNode *node, int begin, int end) {
    if (node->right->type == Intv) {
        parse_expr(node->left, mod_begin, mod_end);
        return;
    }
    parse_expr(node->left, mod_begin, mod_end); // 设置为最大、最小值
    parse_expr(node->right, 1, end > 1 ? end : 1);
}
void Index_Add(TreeNode *node, int begin, int end) {
    if (node->right->type == Intv) {
        int i = node->right->val.Int;
        int _begin = (begin == mod_begin) ? begin : begin - i;
        int _end = (end == mod_end) ? end : end - i; //求余范围不参与计算
        parse_expr(node->left, _begin, _end);
        return;
    }
    parse_expr(node->left, begin, end);
    parse_expr(node->right, begin, end);
}
```

之后调整了solution2.cc，包括修改了函数标签和对于有多个grad_to时的处理（代码如下），使用Com节点将他们连接起来，另外对于临时变量的初始化问题也进行了调整。

```
if (caseInfo.grad_to.size() == 1) {
        TreeNode* root = change_tree(TreeRoot, caseInfo.grad_to[0]);
        TreeRoot = root;
        }
else {
    int num = caseInfo.grad_to.size();
    TreeNode* left = change_tree(TreeRoot, caseInfo.grad_to[num - 2]);
TreeNode* right = change_tree(TreeRoot, caseInfo.grad_to[num - 1]);
    TreeNode* root = new_node(Com, left, right);
    for(int i = num - 3; i > 0; --i) {
        TreeNode* right = root;
        TreeNode* left = change_tree(TreeRoot, caseInfo.grad_to[i]);
        root = new_node(Com, left, right);
    }
    TreeRoot = root;
    }
```

最后利用IR来生成十个grad_case.cc，存于kernel/下。

举例

以case1为例。

首先为 `C<4，16>[i，j] = A<4，16>[i，j] * B<4，16>[i，j] + 1.0` 生成常规的语法树，根节点是一个Eq类型的节点。读取json其他部分可知，grad_to为A。

在递归调用get_grad()的过程中，第一层返回一个类型为Eq的节点，左子节点为dA，右子节点为new_node(Mul, Dout, right)，Dout即为dc。然后调用new_node(Mul, Dout, right)，实质上是计算doutput*f'(input)。doutput已知，需要计算的是f'(input)。

right对应的表达式是A<4, 16>[i, j] * B<4, 16>[i, j] + 1.0，是一个Add类型的节点，于是递归调用get_grad()，求出加号左右两边对A的导数，然后sim_add()。

其中右子节点是const类型，落入default，递归调用的get_grad()返回zero_node；左子节点是mul类型，递归调用的get_grad()返回sim_add(sim_mul(get_grad(node->left), node->right), sim_mul(node->left, get_grad(node->right)))，对应的表达式显然是B<4, 16>[i, j]。

最终生成的新AST的表达式是dA<4, 16>[i, j]=dC<4, 16>[i, j]*B<4, 16>[i, j]，是正确的。

之后依次调用buildTable.h、solution2.cc和IRPrinter对于结果进行最终生成。

总结

我们结合求导法则，深度优先地遍历所给式子的AST，构造一个新的AST，对应于分析出来的求导表达式，然后根据新AST生成最终的结果。