

编译技术 Project-2 实践报告

组员

1700012759 周厚健
1700012934 刘添翼
1700012771 张旭睿
1700016629 常可

我们组在本次 Project 上的工作主要集中在 `/src/generate.cc` 和 `/project2/solution/solution2.cc` 两个文件上, 其中 `solution2.cc` 的大部分内容继承自 Project1 中完成的 `codeBuilder.cc`。求导过程的主要思路为在给定的 `case.json` 中的 `kernel` 字符串上直接修改为其导数形式, 重新解析输入输出的张量并输出到新文件中, 然后再调用 `solution2.cc` 中进一步完善的代码生成器来生成代码。

小组分工:

周厚健: 数组下标中的运算符及下标范围处理
刘添翼: 数组下标中的运算符及下标范围处理
张旭睿: 为数组下标中的运算符处理提供思路及框架
常可: `kernel` 串求导处理

自动求导技术设计:

这部分主要内容位于 `/src/generate.cc` 中。

在本次作业中, 关于求导的内容实际上只用到了导数的四则运算及链式法则。这些知识应该是各位同学在高中就熟练掌握的。

具体而言, 以 Case1 为例, 我们给出的 dC 的值实质上就是 $\frac{\partial loss}{\partial C}$, 我们要求的 dA 就是 $\frac{\partial loss}{\partial A}$, 根据链式法则, 我们有

$$\frac{\partial loss}{\partial A} = \frac{\partial loss}{\partial C} \times \frac{\partial C}{\partial A}$$

因此我们实际上只需要求出 $\frac{\partial C}{\partial A}$ 。

所以, 整个求导的流程为:

1. 对每一个需要被求导的自变量考虑;
2. 对这个自变量的每一个单元, 把所有可能用到这一单元的式子提取出来, 分别求导, 乘上对应位置的 dA 值, 最后相加。

在实现时, 我们是直接 `kernel` 表达式求导得到新的 `kernel` 表达式, 最后再丢给 Project1 中的代码生成器完成 C++ 代码的生成。

对于给定的 `kernel`, 等式右边每出现一个自变量的位置, 就意味着用到它的情况多一种。对于每一种情况, 我们把其它自变量和位置不同的同一自变量均视为常数 (导数为 0), 而恰好为我们现在选定位置的自变量求导值为 1。

然后, 我们把选定位置的自变量加上字符 `d` 后移动到左边, 等式的右边先乘上对应的 dC 值, 然后写出 $\frac{\partial C}{\partial A}$ 的值:

```
return "d" + now_grad + nowindex + "=d" + now_grad + nowindex + "+d" + ker.substr(0,
i) + "*" + getgrad(i + 1, ker.size() - 2) + "));
```

(代码中 now_grad 表示当前求导的自变量名称, nowindex 表示下标列表)

其中的 getgrad 函数就是核心的求导部分, 它的输入参数是 kernel 字符串的首尾位置, 表示当前求导的子串。它首先分析表达式中是否有低优先级的运算符+和- (不能在任何括号内), 如果有则按照对应的求导法则 (分别求导再相加减) 进行求导, 否则再看是否有 * 和 /, 如果有也按照对应的法则求导 $d(A * B) = A * dB + B * dA$, $\frac{dA}{dB} = \frac{A * dB - B * dA}{B^2}$ 。如果都没有, 那么可能是如下情况:

- + 最外层有括号, 对括号内部分继续求导;
- + 数值常量, 返回 0;
- + 标识符, 判断是否和当前求导变量位置相同, 相同返回 1, 不同返回 0。

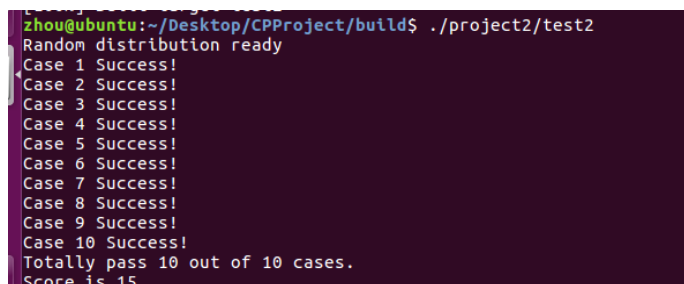
例如, Case2 中有 2 个用到 A 的位置, 对第 1 个 A 求导时, 第 2 个 A 视为常数, 这样求导的结果为 $1 * A + A * 0$, 对第 2 个 A 求导时, 结果同理为 $0 * A + A * 1$, 相加就是正确结果。虽然看似浪费计算过程, 但这一算法可以更好地适应下标不同的情况。

综上, 我们的求导流程为:

1. 对每个求导变量分析;
2. 对当前求导变量的每一个出现位置分析;
3. 对当前出现位置, 把当前位置上出现的标识符及下标加上 d 后放到等式左边, 右边放上对应的 dC 乘上求导结果;
4. 把所有求导变量的所有位置的求导结果拼接起来, 整合到 kernel 中。

实现流程 & 实验结果内容:

首先在 solution2.cc 的 main 函数中调用 generate.cc, 该文件会依次读取 10 个 case 的 json 文件并将其转换成导数的 json 文件, 以不同的文件名重新输出到 /project2/cases 文件夹中; 随后的实现过程与 Project1 相同, 都是调用 solution2.cc 中的 AST 生成器将符合文法的 kernel 字符串转换成语法树, 并将其交付给 IRPrinter 以输出代码。以下为结果截图:



```

Zhou@ubuntu:~/Desktop/CPPProject/build$ ./project2/test2
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.

```

以及生成的代码文件截图:

```
grad_case6.cc (-/Desktop/CPPProject/project2/kernels) - gedit
#include "../run2.h"

void grad_case6(float (&C)[8][16][3][3], float (&dA)[2][8][5][5], float (&dB)[2][16][7][7]) {
    for (int n=0; n<2; ++n) {
        for (int c=0; c<16; ++c) {
            for (int a0=0; a0<7; ++a0) {
                for (int b0=0; b0<7; ++b0) {
                    for (int k=0; k<8; ++k) {
                        for (int p=0; p<5; ++p) {
                            for (int q=0; q<5; ++q) {
                                dB[n][c][a0][b0] = dB[n][c][a0][b0] + dA[n][k][p][q] * (((a0 -
p >= 0) && (a0 - p < 3)) ? (((b0 - q >= 0) && (b0 - q < 3)) ? (C[k][c][(a0 - p)
[(b0 - q)]) : (0))) : (0))) ;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

下面以 case 6 为例解释一下所用技术的正确性：

```
{
    "name": "grad_case6",
    "ins": ["B", "C"],
    "outs": ["A"],
    "data_type": "float",
    "kernel": "A<2, 8, 5, 5>[n, k, p, q] = B<2, 16, 7, 7>[n, c, p + r, q +
s] * C<8, 16, 3, 3>[k, c, r, s];",
    "grad_to": ["B"]
}
```

首先可以看到，case 6 只是一个简单的矩阵乘法。通过 generate.cc 的处理我们可以得到如下的求导后的 kernel：

```
grad_case6.cc case6_graded.json
A[n,k,p,q]=B[n,c,p+r,q+s]*C[k,c,r,s]
{
    "data_type": "float",
    "ins": [ "C", "dA" ],
    "kernel" : "dB<2,16,7,7>[n,c,p+r,q+s]=dB<2,16,7,7>[n,c,p+r,q+s]+dA<2,8,5,5>
[n,k,p,q]*(C<8,16,3,3>[k,c,r,s]);",
    "name" : "grad_case6",
    "outs" : [ "dB" ]
}
```

可以看到，经过求导的 kernel 串中的左值表达式里数组下标中存在运算符，这不符合要求。在 solution2.cc 中的 handleS 函数中新增了对这种情况的处理：对每个不符合要求的下标，我们均将其在字符串替换为新的索引，如生成的代码图中“p+r”被替换成“a0”等等，与此同时，新的索引的范围也会依据旧索引的范围被重新计算。在此过程中，“+”右边的旧索引会被抛弃，如生成的代码图中的“r”会被替换成“a0-p”。但这时候需要注意 a0-p 不能越界，于是通过原来的 r 的范围($0 \leq r < 3$)生成 select 语句，并生成如图中的“a?b:c”语句；“s”被替换成“b0-q”的情况同理。最后，替换好的字符串会交付给其他的 handleXXX 函数继续进行语法树的生成；语法树生成的部分在 Project 1 中已有涉及，不再赘述。

另：我们自定义的 Select 语句的文法如下：

```
Select ::= ^(Cond,VTrue,VFalse)
Cond ::= Cond && Cond | (IdExpr ≥ IntV) | (IdExpr < IntV)
```

VTrue ::= RHS

VFalse ::= RHS

总结：

在这个过程中，所用到的最重要的编译知识是语法树以及文法分析部分。作为本次 Project 的主体部分之一的 solution2.cc 本质上就是一个语法分析器——通过 Project1 中定义的以及 Project2 中由于"a?b:c"语句的需要而添加的文法，以及 IR 包中的各种节点，再借由 solution2.cc 中的各种 handleXX 函数（即对不同非终结符号或终结符号的分析器），我们能方便的生成一棵完整的语法分析树，并将其通过 IRPrinter 转换成合法的 C 代码。