8.1 命令式和符号式混合编程

本书到目前为止一直都在使用命令式编程，它使用编程语句改变程序状态。考虑下面这段简单的命令式程序。

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

fancy_func(1, 2, 3, 4) # 10
```

和我们预期的一样，在运行语句 `e = add(a, b)` 时，Python会做加法运算并将结果存储在变量 `e` 中，从而令程序的状态发生改变。类似地，后面的两条语句 `f = add(c, d)` 和 `g = add(e, f)` 会依次做加法运算并存储变量。

虽然使用命令式编程很方便，但它的运行可能很慢。一方面，即使 `fancy_func` 函数中的 `add` 是被重复调用的函数，Python也会逐一执行这3条函数调用语句。另一方面，我们需要保存变量 `e` 和 `f` 的值直到 `fancy_func` 中所有语句执行结束。这是因为在执行 `e = add(a, b)` 和 `f = add(c, d)` 这2条语句之后我们并不知道变量 `e` 和 `f` 是否会被程序的其他部分使用。

与命令式编程不同，符号式编程通常在计算流程完全定义好后才被执行。多个深度学习框架，如**Theano和TensorFlow，都使用了符号式编程**。通常，符号式编程的程序需要下面3个步骤：

1. 定义计算流程；
2. 把计算流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

下面我们用符号式编程重新实现本节开头给出的命令式编程代码。

```
def add_str():
    return '''
def add(a, b):
    return a + b
'''
```

```
def fancy_func_str():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''

def evoke_str():
    return add_str() + fancy_func_str() + '''
print(fancy_func(1, 2, 3, 4))
'''

prog = evoke_str()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

输出：

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))

10
```

以上定义的3个函数都仅以字符串的形式返回计算流程。最后，我们通过 `compile` 函数编译完整的计算流程并运行。由于在编译时系统能够完整地获取整个程序，因此有更多空间优化计算。例如，编译的时候可以将程序改写成 `print((1 + 2) + (3 + 4))` ，甚至直接改写成 `print(10)` 。这样不仅减少了函数调用，还节省了内存。

对比这两种编程方式，我们可以看到以下两点。

- 命令式编程更方便。当我们在Python里使用命令式编程时，大部分代码编写起来都很直观。同时，命令式编程更容易调试。这是因为我们可以很方便地获取并打印所有的中间变量值，或者使用Python的调试工具。
- 符号式编程更高效并更容易移植。一方面，在编译的时候系统容易做更多优化；另一方面，符号式编程可以将程序变成一个与Python无关的格式，从而可以使程序在非Python环境下运行，以避开Python解释器的性能问题。

8.1.1 混合式编程取两者之长

大部分深度学习框架在命令式编程和符号式编程之间二选一。例如，**Theano和受其启发的后来者TensorFlow使用了符号式编程，Chainer和它的追随者PyTorch使用了命令式编程，而Gluon则采用了混合式编程的方式。**