# 搭建自己的深度学习框架

## 计算图模型

### 定义

计算图的概念最早可以追溯到20世纪50年代，由计算机科学家John W. Carr和Peter M. Naur在其论文中提出。他们提出了一种称为“流程图”（Flowchart）的图形化表示方法，用于描述计算机程序的执行过程。随着计算机科学的发展，计算图的概念得到了进一步的发展和应用。在深度学习领域，计算图被广泛应用于描述和优化神经网络模型。

计算图（Computational Graph）是一种用于描述计算过程的图形化表示方法，它将计算过程抽象成节点和边的有向图，其中节点表示计算操作，边表示数据流动。在计算图模型中，每个节点都代表一个计算操作，例如加法、乘法、卷积等。节点之间的边表示数据在计算过程中的流动方向，即数据从一个节点流向另一个节点进行计算。这种数据流动方式使得计算图模型能够自然地表示并行计算过程，因为节点之间的数据流动可以被分配到不同的计算单元中进行处理。输入节点和节点输出的值称为张量（Tensor），其中包含标量（Scalars）、向量（Vector）和矩阵（Matrix）以及更高阶的张量。

计算图模型的优点是可以直观地描述计算过程，易于理解和优化。它还可以帮助开发者更好地理解和优化深度学习模型，因为深度学习模型本质上就是由多个计算操作组成的计算图。通过分析计算图，可以发现模型中的瓶颈操作，进而进行优化，提高模型的性能和效率。计算图模型也是许多深度学习框架的基础，例如TensorFlow和PyTorch。这些框架将计算图模型作为底层运行时的表示方式，通过构建计算图来定义模型，并通过计算图来自动化地进行反向传播和优化。

### 计算图结构

下面用两个简单的示例以及代码实现来展示计算图模型的结构。图7-1为计算两个输入张量的和，其中计算和既是两个输入节点的输出节点，也作为新的输入节点向下传递。

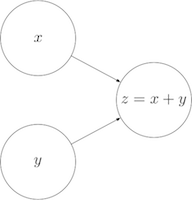


图 7‑1 张量求和计算图

一旦计算变得更加复杂，那么使用计算图将会使计算流程会变得更清晰。例如，图7-2定义了一个仿射变换，最终计算图表示的是将三个输入张量通过变换以Ax+b为结果输出。

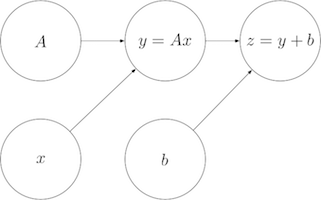


图 7‑2 仿射变换计算图

下面给出计算图中节点的原型实现以及上述两个计算图的python代码实现：

1. class Operation:
2. """表示执行计算的图形节点。
3. `Operation` 是 `Graph` 中的一个节点，它取零或更多对象作为输入，
4. 并产生零个或多个对象作为输出。
5. """
6. def \_\_init\_\_(self, input\_nodes=[]):
7. """构造父类算子 Operation
8. """
9. self.input\_nodes = input\_nodes
10. # 初始化consumer列表（即接收此操作输出作为输入的节点）
11. self.consumers = []
12. # 将此操作附加到所有输入节点的consumer列表
13. for input\_node in input\_nodes:
14. input\_node.consumers.append(self)
15. # 将此操作附加到当前活动的默认图中的操作列表
16. \_default\_graph.operations.append(self)
17. def compute(self):
18. """
19. Operation 具体实现
20. """
21. Pass
22. class add(Operation):
23. """返回 x + y 按元素相加.
24. """
25. def \_\_init\_\_(self, x, y):
26. """构建 add 节点
27. Args:
28. x: 第一个加数节点
29. y: 第二个加数节点
30. """
31. super().\_\_init\_\_([x, y])
32. def compute(self, x\_value, y\_value):
33. """计算 add operation 的结果
34. Args:
35. x\_value: 第一个加数节点
36. y\_value: 第二个加数节点
37. """
38. return x\_value + y\_value
39. class matmul(Operation):
40. """将矩阵 a 乘以矩阵 b, 生成 a \* b。
41. """
42. def \_\_init\_\_(self, a, b):
43. """构建 matmul 节点
44. Args:
45. a: 第一个矩阵
46. b: 第二个矩阵
47. """
48. super().\_\_init\_\_([a, b])
49. def compute(self, a\_value, b\_value):
50. """计算 matmul operation 的输出
51. Args:
52. a\_value: 第一个矩阵值
53. b\_value: 第二个矩阵值
54. """
55. return a\_value.dot(b\_value)

在这两个操作中，我们假设张量是NumPy数组，其中已经为我们实现了逐元素加法和矩阵乘法(.dot)。

值得一提的是，并非计算图中的所有节点都是操作。例如，在仿射变换图中，‘and’不是操作。相反，它是计算图的输入，一旦我们想要计算输出，就必须为其提供一个值。为了提供这样的值，我们引入了占位符（Placeholder）的概念。在计算图中，占位符是一种特殊的节点，它的作用是在计算图构建时占据一个位置，等到实际运行时再提供数据。占位符通常用于表示输入数据或其他需要在运行时才能确定的数据，例如训练数据、测试数据、超参数等。

占位符的主要功能是为计算图提供输入数据，并在运行时动态地将数据传递给计算图中的其他节点进行计算。在TensorFlow等深度学习框架中，占位符通常通过feed\_dict参数进行数据传递，即在运行计算图时，通过feed\_dict参数将占位符替换为实际的数据。占位符的使用可以使计算图更加灵活和通用，因为它可以在运行时接受不同类型和形状的数据，并将其传递给计算图中的其他节点进行计算。这种灵活性使得占位符在深度学习中得到了广泛的应用，例如在构建神经网络模型时，我们可以使用占位符来表示输入数据、标签、学习率等参数，以及在训练和测试过程中动态地传递数据。占位符的使用使得深度学习模型的构建和调试更加方便和高效，同时也为模型的扩展和应用提供了更多的可能性。

下面给出占位符（placeholder）类的python实现：

1. class placeholder:
2. """
3. 表示在计算计算图的输出时必须提供值的占位符节点
4. """
5. def \_\_init\_\_(self):
6. """构建 placeholder
7. """
8. self.consumers = []
9. # 将此占位符附加到当前活动的默认图中的占位符列表
10. \_default\_graph.placeholders.append(self)

除了占位符以外，还有一类被称为变量（Variable）特殊的节点，它的值可以在计算过程中被修改和更新。变量通常用于表示模型参数、学习参数或其他需要在计算过程中更新的数据。与占位符不同，变量的值在计算图构建时就已经确定，并在计算过程中被动态地更新。在TensorFlow等深度学习框架中，变量通常通过Variable类进行定义和管理。变量的主要功能是存储和更新模型参数，例如神经网络中的权重和偏置。在训练过程中，我们通常需要通过反向传播算法来更新模型参数，以使模型的预测结果更加准确。在这种情况下，变量提供了一种方便的方式来存储和更新模型参数，同时也使得模型的训练过程更加高效和灵活。变量的另一个重要功能是持久化，即将变量的值保存在磁盘上，以便在需要时重新加载。这种持久化功能使得我们可以在训练过程中保存模型参数的状态，并在需要时重新加载，以进行模型的推断或继续训练。

变量（Variable）类的python实现如下：

1. class Variable:
2. """
3. 表示一个变量（即计算图的一个固有的、可变的参数）。
4. """
5. def \_\_init\_\_(self, initial\_value=None):
6. """构建 Variable
7. Args:
8. initial\_value: 此变量的初始值
9. """
10. self.value = initial\_value
11. self.consumers = []
12. # 将此变量附加到当前活动的默认图中的变量列表
13. \_default\_graph.variables.append(self)

最后，我们需要一个将所有操作、占位符和变量捆绑在一起的类。在创建新图时，我们可以调用它的 as\_default 方法将 \_default\_graph 设置为这个图。这样，我们就可以创建操作、占位符和变量，而不必每次都传入对图形的引用。

1. class Graph:
2. """表示计算图
3. """
4. def \_\_init\_\_(self):
5. """构建 Graph"""
6. self.operations = []
7. self.placeholders = []
8. self.variables = []
9. def as\_default(self):
10. global \_default\_graph
11. \_default\_graph = self

现在让我们使用我们构建的类来为图7-3所表示的仿射变换创建计算图：

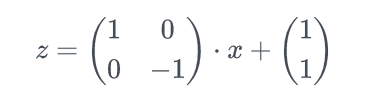


图 7‑3 仿射变换实例

1. # 创建 gragh,variables,placeholder,隐藏层节点y，输出节点z
2. Graph().as\_default()
3. A = Variable([[1, 0], [0, -1]])
4. b = Variable([1, 1])
5. x = placeholder()
6. y = matmul(A, x)
7. z = add(y, b)

至此，我们完成了一个计算图的创建，那么如何完成对计算结果的输出呢？我们创建一个封装操作执行的会话类（Session），希望能够创建一个会话实例并在这个实例上调用一个run方法，传递想要计算的操作和一个包含占位符值的字典。

为了计算由operation表示的函数，我们需要以正确的顺序应用计算。例如，我们不能在计算出中间结果之前进行下一步计算。因此，我们必须确保以正确的顺序执行操作，以保证作为operation输入的每个节点值都是已经计算完成的。这可以通过后序遍历来实现。

1. import numpy as np
2. class Session:
3. """表示计算图的特定执行。
4. """
5. def run(self, operation, feed\_dict={}):
6. """计算 Operation 的输出
7. Args:
8. operation: 我们要计算其输出的operation。
9. feed\_dict: 将占位符映射到 Session 的值的字典
10. """
11. # 对图执行后序遍历以使节点按正确顺序排列
12. nodes\_postorder = traverse\_postorder(operation)
13. # 迭代所有节点以确定它们的值
14. for node in nodes\_postorder:
15. if type(node) == placeholder:
16. # 将节点值设置为来自 feed\_dict 的占位符值
17. node.output = feed\_dict[node]
18. elif type(node) == Variable:
19. # 将节点值设置为变量的值属性
20. node.output = node.value
21. else: # Operation
22. # 从输入节点的输出值中获取此操作的输入值
23. node.inputs = [input\_node.output for input\_node in node.input\_nodes]
24. # 计算此操作的输出
25. node.output = node.compute(\*node.inputs)
26. # 将列表转换为 numpy 数组
27. if type(node.output) == list:
28. node.output = np.array(node.output)
29. # 返回请求的节点值
30. return operation.output
31. def traverse\_postorder(operation):
32. """执行后序遍历，按照必须计算的顺序返回节点列表
33. Args:
34. operation: 开始遍历的 operation
35. """
36. nodes\_postorder = []
37. def recurse(node):
38. if isinstance(node, Operation):
39. for input\_node in node.input\_nodes:
40. recurse(input\_node)
41. nodes\_postorder.append(node)
42. recurse(operation)
43. return nodes\_postorder

最后，让我们用一个例子测试我们的类是否正常运行：

1. session = Session()
2. output = session.run(z, {
3. x: [1, 2]
4. })
5. print(output)
6. # output = [2, -1]

### 静态图与动态图

在了解计算图的基本构成后，那么下一个问题就是：计算图要如何自动化生成呢？在机器学习框架中可以生成静态图和动态图两种计算图。静态生成可以根据前端语言描述的神经网络拓扑结构以及参数变量等信息构建一份固定的计算图。因此静态图在执行期间可以不依赖前端语言描述，常用于神经网络模型的部署，比如移动端人脸识别场景中的应用等。

动态图则需要在每一次执行神经网络模型依据前端语言描述动态生成一份临时的计算图，这意味着计算图的动态生成过程灵活可变，该特性有助于在神经网络结构调整阶段提高效率。主流机器学习框架TensorFlow、MindSpore均支持动态图和静态图模式；PyTorch则可以通过工具将构建的动态图神经网络模型转化为静态结构，以获得高效的计算执行效率。了解两种计算图生成方式的优缺点及构建执行特点，可以针对待解决的任务需求，选择合适的生成方式调用执行神经网络模型。

尽管这两个库都使用有向无环图（或 DAG）来表示它们的机器学习和深度学习模型，但它们让数据和计算在图中流动的方式仍然存在很大差异。本文将通过代码示例以可视化方式介绍这些差异。

#### 静态图

静态图的生成与执行原理如 图7-4 所示，采用先编译后执行的方式，该模式将计算图的定义和执行进行分离。

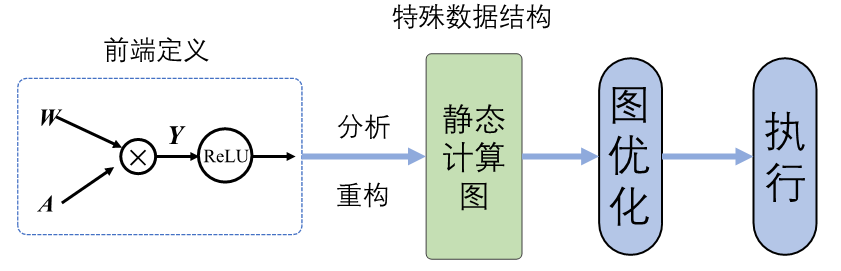


图 7‑4 静态图生成与执行

使用前端语言定义模型形成完整的程序表达后，机器学习框架首先对神经网络模型进行分析，获取网络层之间的连接拓扑关系以及参数变量设置、损失函数等信息。然后机器学习框架会将完整的模型描述编译为可被后端计算硬件调用执行的固定代码文本，这种固定代码文本通常被称为静态计算图。当使用静态计算图进行模型训练或者推理过程时，无需编译前端语言模型。静态计算图直接接收数据并通过相应硬件调度执行图中的算子来完成任务。静态计算图可以通过优化策略转换成等价的更加高效的结构，提高后端硬件的计算效率。

在部分机器学习框架中进行前端定义时，需要声明并编写包含数据占位符、损失函数、优化函数、网络编译和执行环境以及网络执行器等在内的预定义配置项，此外还需要使用图内控制流算子编写控制语句。随着机器学习框架设计的改进与发展，框架趋向于提供的友好的编程接口和统一的模型构建模式，比如MindSpore提供动静态统一的前端编程表达。因此为了便于理解静态生成的过程与原理，此处使用更加简洁的语言逻辑描述模型。

构建并执行下列伪代码，来详细讲解静态图的生成与执行。

1. def model(X, flag):
2. if flag>0:
3. Y = matmul(W1, X)
4. else:
5. Y = matmul(W2, X)
6. Y = Y + b
7. Y = relu(Y)
8. return Y

因为机器学习框架在进行静态生成编译时并不读取输入数据，所以需要使用上一小节提到的数据占位符在静态图中表示。由于静态生成时模型无数据输入，因此代码第2行中的条件控制，也无法进行逻辑计算，条件控制在编译阶段并不会完成判断，因此需要将条件控制算子以及所有的分支计算子图加入计算图中。在静态计算图执行计算阶段网络接收数据流入，调度条件控制算子根据输入数据进行逻辑判断，控制数据流入不同的分支计算子图中进行后续计算。在部分机器学习框架中前端语言Python的控制流不能够被正确编译为等价的静态图结构，因此需要机器学习框架的控制原语来实现控制流。

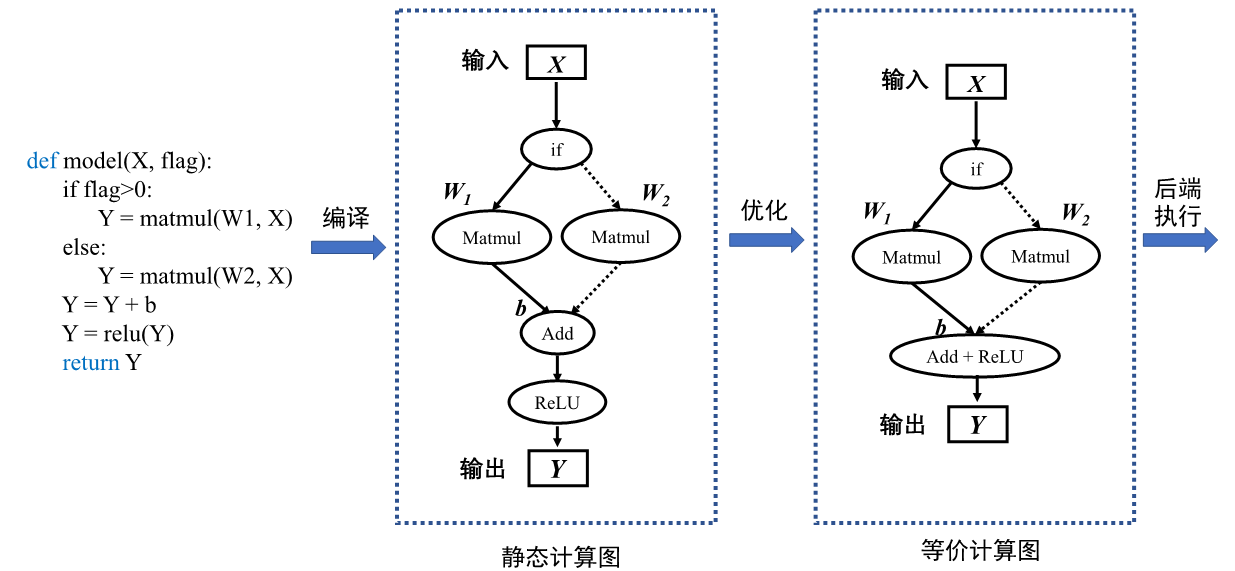


图 7‑5 静态生成

静态计算图具有两大优势：计算性能与直接部署。静态图经过机器学习框架编译时能够获取模型完整的图拓扑关系。机器学习框架掌控全局信息便更容易制定计算图的优化策略，比如算子融合将网络中的两个或多个细粒度的算子融合为一个粗粒度算子，比如图7-5 中将Add算子与ReLU合并为一个操作，可节省中间计算结果的存储、读取等过程，降低框架底层算子调度的开销，从而提升执行性能和效率，降低内存开销。因此使用静态图模型运行往往能够获取更好的性能和更少的内存占用。在部署模型进行应用时，可以将静态计算图序列化保存。在模型推理阶段，执行序列化的模型即可，无需重新编译前端语言源代码。机器学习框架可以将静态计算图转换为支持不同计算硬件直接调用的代码。结合计算图序列化和计算图转硬件代码两种特性，静态图模型可以直接部署在不同的硬件上面，提供高效的推理服务。

尽管静态图在结构和资源分配方面提供了许多优化的可能性，但是其缺点也比较明显：

1. 对可变维度输入的扩展很差。例如，如果没有大量预处理模板代码，具有在28×28图像上训练的静态计算图的 CNN（卷积神经网络）架构将无法在100×100等不同尺寸的图像上表现良好。
2. 调试困难。静态图很难进行调试，主要是因为前端语言构建的神经网络模型经过编译后，计算图结构便固定执行阶段不再改变，并且经过优化用于执行的静态图代码与原始代码有较大的差距。代码执行过程中发生错误时，机器学习框架会返回错误在优化后的静态图代码位置。用户难以直接查看优化后的代码，因此无法定位原始代码错误位置，增加了代码调试难度。比如在示例中，若add算子和relu算子经给优化合并为一个算子，执行时合并算子报错，用户可能并不知道错误指向的是add算子错误 还是relu算子错误。

此外在神经网络模型开发迭代环节，不能即时打印中间结果。若在源码中添加输出中间结果的代码，则需要将源码重新编译后，再调用执行器才能获取相关信息，降低了代码调试效率。对比之下，动态图模式则相比较灵活，接下来解释什么是动态生成机制。

#### 动态图

动态图原理如 图7-6所示，采用解析式的执行方式，其核心特点是编译与执行同时发生。动态图采用前端语言自身的解释器对代码进行解析，利用机器学习框架本身的算子分发功能，算子会即刻执行并输出结果。动态图模式采用用户友好的命令式编程范式，使用前端语言构建神经网络模型更加简洁，深受广大深度学习研究者青睐。

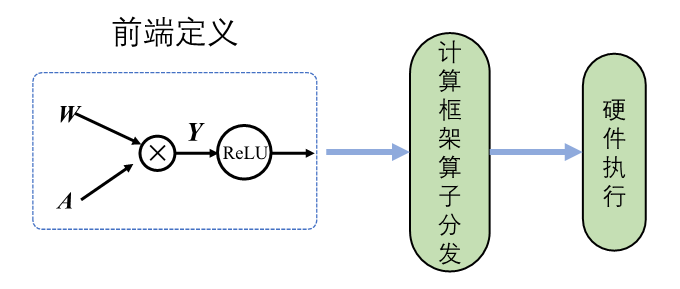


图 7‑6 动态图原理

依旧用上一小节的伪代码来讲解动态图和静态图的区别。静态图和动态图除了在前端语言表达上略有差异，本质的区别在于编译执行过程。使用前端语言构建完成模型表达后，动态生成并不采用机器学习框架编译器生成完整的静态计算图，而是采用前端语言的解释器Python API调用机器学习框架，框架利用自身的算子分发功能，将Python调用的算子在相应的硬件如CPU、GPU、NPU等上进行加速计算，然后再将计算结果返回给前端。该过程并不产生静态的计算图，而是按照前端语言描述模型结构，按照计算依赖关系进行调度执行，动态生成临时的图拓扑结构。

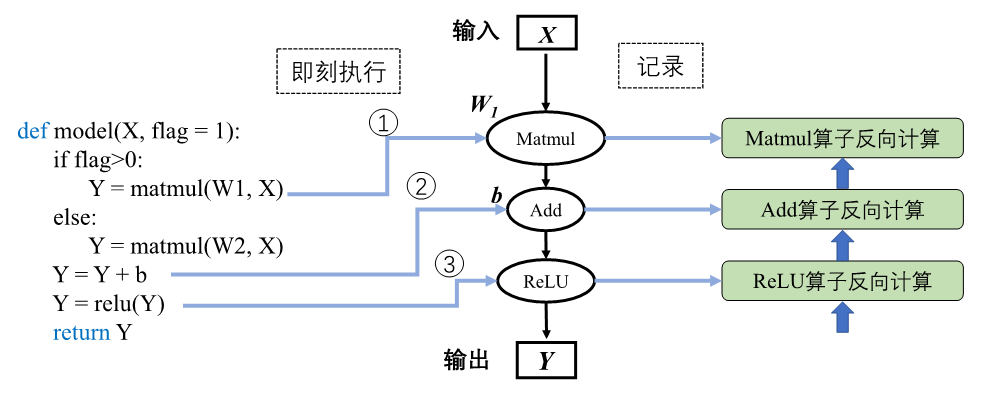


图 7‑7 动态生成

神经网络前向计算按照模型声明定义的顺序进行执行。当模型接收输入数据 后，机器学习框架开始动态生成图拓扑结构，添加输入节点并准备将数据传输给后续节点。模型中存在条件控制时，动态图模式下会即刻得到逻辑判断结果并确定数据流向，因此在图中假设判断结果为真的情况下，图结构中仅会添加关于张量的Matmul算子节点。按照代码制定的模型计算顺序与算子依赖关系，机器学习框架会依次添加Add算子节点和ReLU算子节点。机器学习框架会在添加节点的同时完成算子分发计算并返回计算结果，同时做好准备向后续添加的节点传输数据。当模型再次进行前向计算时，动态生成的图结构则失效，并再次根据输入和控制条件生成新的图结构。相比于静态生成，可以发现动态生成的图结构并不能完整表示前端语言描述的模型结构，需要即时根据控制条件和数据流向产生图结构。由于机器学习框架无法通过动态生成获取完整的模型结构，因此动态图模式下难以进行模型优化以提高计算效率。

在静态生成方式下，由于已经获取完整的神经网络模型定义，因此可以同时构建出完整的前向计算图和反向计算图。而在动态生成中，由于边解析边执行的特性，反向梯度计算的构建随着前向计算调用而进行。在执行前向过程中，机器学习框架根据前向算子的调用信息，记录对应的反向算子信息以及参与梯度计算的张量信息。前向计算完毕之后，反向算子与张量信息随之完成记录，机器学习框架会根据前向动态图拓扑结构，将所有反向过程串联起来形成整体反向计算图。最终，将反向图在计算硬件上执行计算得到梯度用于参数更新。

尽管动态生成中完整的网络结构在执行前是未知的，不能使用静态图中的图优化技术来提高计算执行性能。但动态图可以将新的预处理层动态添加到网络本身，可以很好地拓展对于不同维度输入的可扩展性，这就使得对于初学者更加友好，提高了算法开发迭代效率和神经网络模型改进速率。同时，与静态图相比，动态生成调试起来更加容易，这也是许多人从Tensorflow转向Pytorch的原因之一。由于节点是在任何信息流经节点之前动态创建的，因此用户可以完全控制训练过程中使用的变量，因此错误变得非常容易发现。

静态生成和动态生成的过程各有利弊。表7-1从不同角度进行了总结：

|  |  |  |
| --- | --- | --- |
| **特性** | **静态图** | **动态图** |
| 即时获取中间结果 | 否 | 是 |
| 代码调试难易 | 难 | 易 |
| 控制流实现方式 | 特定的语法 | 前端语言语法 |
| 性能 | 优化策略多，性能更佳 | 图优化受限，性能较差 |
| 内存占用 | 内存占用少 | 内存占用相对较多 |
| 内存占用 | 可直接部署 | 不可直接部署 |

表 7‑1 静态图和动态图对比

针对两种模式的特性，结合任务需求选择合适的模式可以事半功倍，学术科研以及模型开发调试阶段，为了快速验证思想和迭代更新模型结构可以选择动态图模式进行构建算法；网络模型确定，为了加速训练过程或者为硬件部署模型，可以选择静态图模式。

## 7.2自动求导

### 简介

在深度学习中，自动求导是一种非常重要的技术，许多算法都是在自动求导的基础上设计的。使用自动求导可以自动计算损失函数对网络参数的导数，以便优化网络参数。网络模型通常由多个层组成，每个层包含多个神经元，神经元接收来自前一层的输入，并通过激活函数将其转换为输出。网络的输出被用于预测或分类任务，优化网络参数通常涉及到计算损失函数对网络参数的导数，以便在优化算法中更新参数。这个过程可以使用自动求导来自动计算。例如梯度下降（Gradient Descent）算法，梯度下降本质上就是求解损失关于网络参数的梯度，不断计算这个梯度对网络参数进行更新。现代的神经网络框架都实现了自动求导的功能，只需要要定义好网络前向计算的逻辑，在运算时自动求导模块就会自动把梯度算好，不用自己手写求导梯度。

我们以一个迷你神经网络框架tinynn为例，如图7-8所示，神经网络框架整个网络是由一层层的 layer 叠起来的（全连接层、卷积层、激活函数层、Pooling 层等等）。

在实现的时候需要显示为每层定义好前向forward和反向backward（梯度计算）的计算逻辑。从本质上看 这些layer其实是一组基础算子的组合，而这些基础算子（加减乘除、矩阵变换等等）的导函数本身都比较简单，如果能够将这些基础算子的导函数写好，同时把不同算子之间连接逻辑记录（计算依赖图）下来，那么这个时候就不再需要自己写反向了，只需要计算损失，然后从损失函数开始，让梯度自己用预先定义好的导函数，沿着计算图反向流动即可以得到参数的梯度，这个就是自动求导的核心思想。tinynn中之所以有 layer 这个概念，一方面是符合我们直觉上的理解，另一方面是为了在没有自动求导的情况下方便实现。有了自动求导，我们可以抛开layer这个概念，神经网络的训练可以抽象为定义好一个网络的计算图，然后让数据前向流动，让梯度自动反向流动。

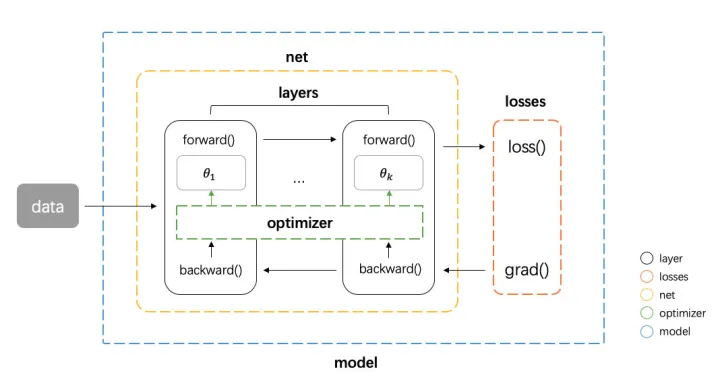


图 7‑8 tinynn架构图

此外，我们可以通过PyTorch的一小段核心的训练代码（来源官方文档 MNIST 例子）来感受PyTorch的基本思路和我们上面描述的是一致的：定义好计算图 -> forward 得到损失 -> 梯度反向流动。

1. for batch\_idx, (data, target) in enumerate(train\_loader):
2. data, target = data.to(device), target.to(device)
3. optimizer.zero\_grad() # 初始化梯度
4. output = model(data) # 从 data 到 output 的计算图
5. loss = F.nll\_loss(output, target) # 从 output 到 loss 的计算图
6. loss.backward() # 梯度从 loss 开始反向流动
7. optimizer.step() # 使用梯度对参数更新

### 自动求导设计

知道了自动求导的基本流程之后，我们考虑如何来实现。先考虑没有自动求导，为每个运算手动写反向传播的情况，在这种情况下我们实际上定义了两个计算图，一个前向一个反向，考虑最简单的线性回归的运算 “WX+B” ，其计算图如下所示。

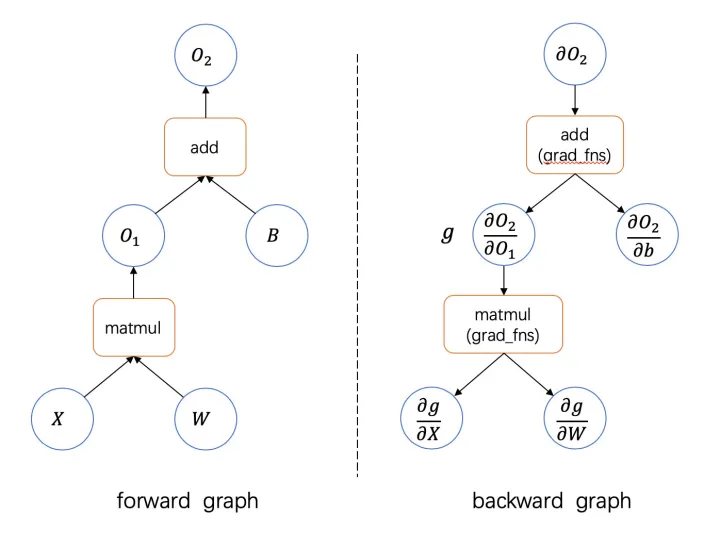


图 7‑9 线性回归计算图

可以看到这两个计算图的结构实际上是一样的，只是在前向流动的是计算的中间结果，反向流动的是梯度，以及中间的运算反向的时候是导数运算。实际上我们可以把两者结合到一起，只定义一次前向计算图，让反向计算图自动生成。

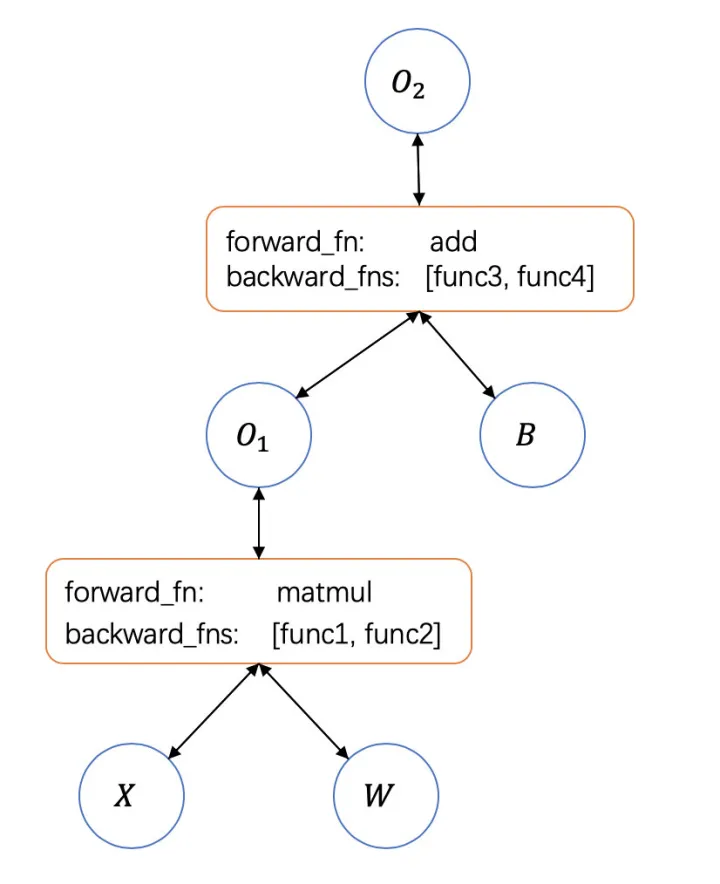


图 7‑10 自动生成反向计算图

从实现的角度看，如果我们不需要自动求导，那么网络框架中的 Tensor 类只需要对 Tensor 运算符有定义，能够进行数值运算（tinynn 中就简单的使用 ndarray 作为 Tensor 的实现）。但如果要实现自动求导，那么 Tensor 类需要额外做几件事：

1.增加一个梯度的变量保存当前 tensor 的梯度

2.保存当前 tensor 依赖的 tensor（如上图中O1依赖于X,W）

3.保存下对各个依赖tensor的导函数（这个导函数的作用是将当前 tensor 的梯度传到依赖的tensor上）

### 自动求导实现

我们按照上面的分析开始实现 Tensor 类如下，初始化方法中首先把 tensor 的值保存下来，然后设置一个名为 **requires\_grad** 的 bool 变量表明这个tensor是否需要求梯度，还要设置名为 dependency 的列表用于保存该 tensor 依赖的 tensor 以及对于他们的导函数。

zero\_grad() 方法比较简单，将当前 tensor 的梯度设置为 0，防止梯度的累加。自动求导从调用计算图的最后一个节点 tensor 的 backward() 方法开始（在神经网络中这个节点一般是 loss）。backward() 方法主要流程为

* 确保改 tensor 确实需要求导 self.requires\_grad == True。
* 将从上个 tensor 传进来的梯度加到自身梯度上，如果没有（反向求导的起点 tensor），则将梯度初始化为 1.0。
* 对每一个依赖的 tensor 运行保存下来的导函数，计算传播到依赖 tensor 的梯度，然后调用依赖 tensor 的 backward() 方法。可以看到这其实就是 Depth-First Search 计算图的节点。

1. def as\_tensor(obj):
2. if not isinstance(obj, Tensor):
3. obj = Tensor(obj)
4. return obj
5. class Tensor:
6. def \_\_init\_\_(self, values, requires\_grad=False, dependency=None):
7. self.\_values = np.array(values)
8. self.shape = self.values.shape
9. self.grad = None
10. if requires\_grad:
11. self.zero\_grad()
12. self.requires\_grad = requires\_grad
13. if dependency is None:
14. dependency = []
15. self.dependency = dependency
16. @property
17. def values(self):
18. return self.\_values
19. @values.setter
20. def values(self, new\_values):
21. self.\_values = np.array(new\_values)
22. self.grad = None
23. def zero\_grad(self):
24. self.grad = np.zeros(self.shape)
25. def backward(self, grad=None):
26. assert self.requires\_grad, "Call backward() on a non-requires-grad tensor."
27. grad = 1.0 if grad is None else grad
28. grad = np.array(grad)
29. # accumulate gradient
30. self.grad += grad
31. # propagate the gradient to its dependencies
32. for dep in self.dependency:
33. grad\_for\_dep = dep["grad\_fn"](grad)
34. dep["tensor"].backward(grad\_for\_dep)

从最后的使用反推，我们希望这样使用 Tensor 类：

1. W = Tensor([[1], [3]], requires\_grad=True) # 2x1 tensor
2. X = Tensor([[1, 2], [3, 4], [5, 6], [7, 8]], requires\_grad=True)
3. # 4x2 tensor
4. O = X @ W # suppose to be a 4x1 tensor

在这里我们将对运算符进行重载，使X和W完成矩阵乘法输出正确的O的同时，让O能记下它所依赖的W和X。

1. class Tensor:
2. # ...
3. def \_\_matmul\_\_(self, other):
4. # 1. calculate forward values
5. values = self.values @ other.values
6. # 2. if output tensor requires\_grad
7. requires\_grad = ts1.requires\_grad or ts2.requires\_grad
8. # 3. build dependency list
9. dependency = []
10. if self.requires\_grad:
11. # O = X @ W
12. # D\_O / D\_X = grad @ W.T
13. def grad\_fn1(grad):
14. return grad @ other.values.T
15. dependency.append(dict(tensor=self, grad\_fn=grad\_fn1))
16. if other.requires\_grad:
17. # O = X @ W
18. # D\_O / D\_W = X.T @ grad
19. def grad\_fn2(grad):
20. return self.values.T @ grad
21. dependency.append(dict(tensor=other, grad\_fn=grad\_fn2))
22. return Tensor(values, requires\_grad, dependency)
23. # ...

这个方法里面做了三件事：

1.计算矩阵乘法结果（这个是必须的）

2.确定是否需要新生成的 tensor 是否需要梯度，这个由两个操作数决定。比如在这个例子中，如果 W 或者 X 需要梯度，那么生成的 O也是需要计算梯度的（这样才能够计算 W 或者 X 的梯度）。

3.建立 tensor 的依赖列表

自动求导中最关键的部分就在这里，还是以 O = X @ W 为例子，这里我们会先检查是否 X需要计算梯度，如果需要，我们需要把导函数 D\_O / D\_X 定义好，保存下来；同样的如果 W 需要梯度，我们将 D\_O / D\_W 定义好保存下来。最后生成一个 dependency 列表保存着在新生成的 tensor O 中。然后回顾backward()方法，backward() 方法会遍历 tensor 的 dependency ，将用保存的 grad\_fn 计算要传给依赖 tensor 的梯度，然后调用依赖 tensor 的 backward() 方法将梯度传递下去，从而实现了梯度在整个计算图的流动。

1. grad\_for\_dep = dep["grad\_fn"](grad)
2. dep["tensor"].backward(grad\_for\_dep)

通过这一节，我们了解了自动求导的概念和作用，针对一个简单的线性变换实例完成了从设计计算图到自动求导的实现，直观感受自动求导背后的逻辑。

## 7.3 例子

## 参考文献

使用Zotero工具栏中的Add/Edit Bibliography插入参考文献。