Microsoft

人工智能系统 System for AI

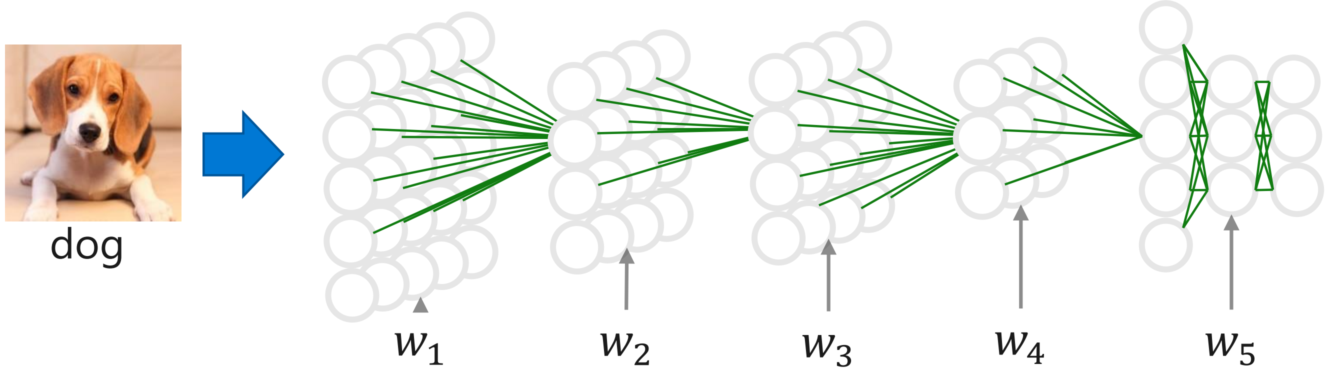
深度神经网络计算框架基础
Computation frameworks for DNN

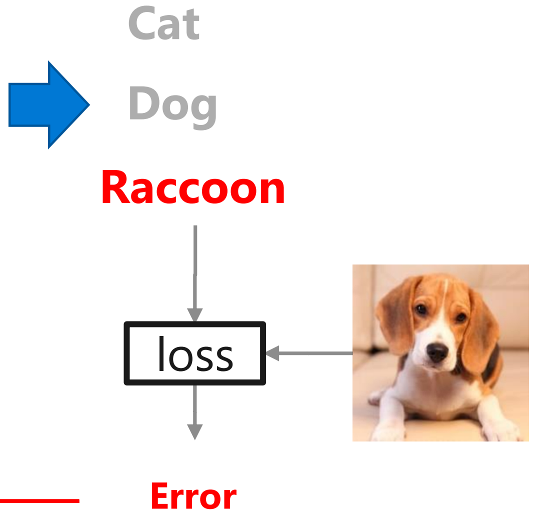
主要内容

- 主要内容
  - Tensor
  - Data-flow graph, DAG
  - Backpropagation and auto-differentiation
  - Graph execution and scheduling
  - Symbolic and imperative execution, static vs dynamic graph
  - Hardware device support
- 参考系统
  - Caffe, Theano, DistBelief
  - TensorFlow, CNTK
  - PyTorch, Chainer, DyNet

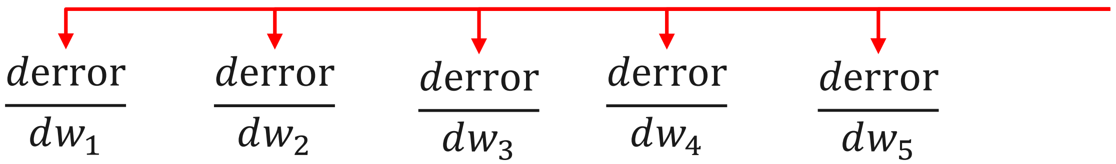
回顾：深度学习基础

1. 定义一个带参数的函数（神经网络）：$\hat{y}_i = f(w, x_i)$





2. 定义优化目标： $\min_{w} \mathrm{L}(w) = Loss(\hat{y}_i, y_i)$

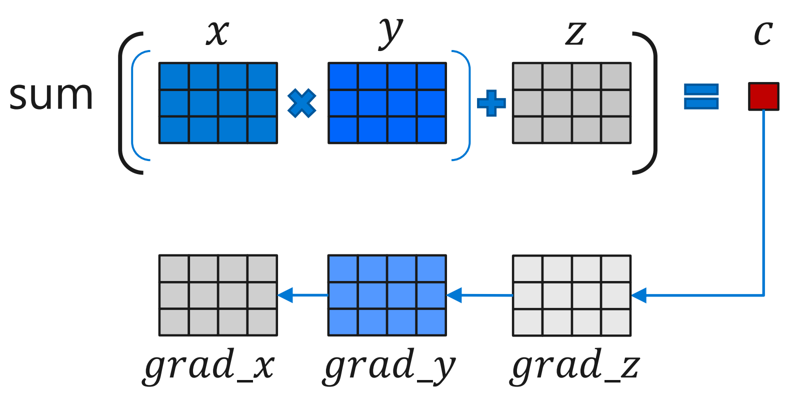


3. 计算梯度并更新参数： $w_i \leftarrow w_i - \eta \nabla_{w_i} L(w)$

例：一种极端的计算方法

- 用某种高级语言从头实现一个模型的计算过程

```
import numpy as np
N, D = 3, 4
x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



例：另一种极端的计算方法

- 为常用模型在某个加速设备上实现一个高度优化的计算库

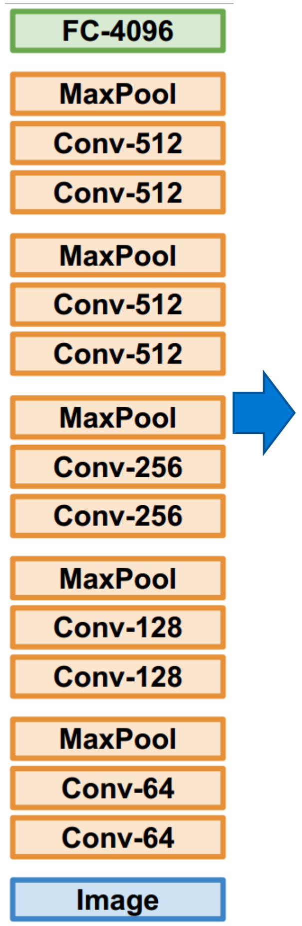
```
import xxlib

x, y = load_data()

y = xxlib.resnet152(x)
```

深度学习计算框架的目的

- 提供灵活的编程模型和编程接口
  - 简洁的神经网络计算原语编程语言
  - 提供直观地模型构建方式
  - 较好的支持与现有生态环境融合
- 提供高效和可扩展的计算能力
  - 自动推导计算图
  - 自动编译优化算法，包括不限于：公共子表达式消除，内核融合，内存布局优化等
  - 根据不同体系结构和硬件设备自动并行化
  - 自动分布式化，并扩展到多个计算节点
  - 持续优化

早期的深度学习框架（-2010）

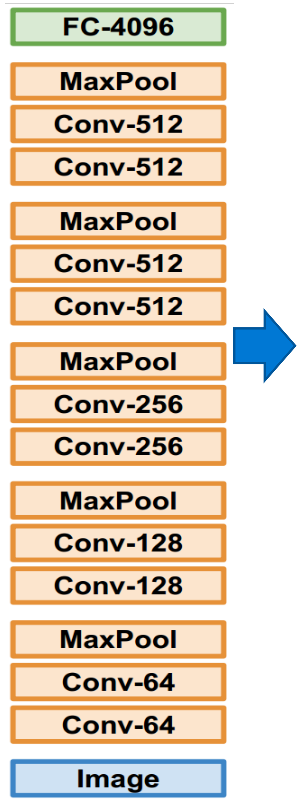
- 主要解决的问题
  - 基于CNN网络的图像识别类模型为主，由一些常用的layers组成，如：
    - Convolution, Pooling, BatchNorm, Activation等
- 主要特点
  - 通过简单配置文件的形式定义神经网络
  - 模型可由一些常用layer构成一个简单的图
  - 框架提供每一个layer及其梯度计算实现
  - 支持多设备加速：CPU和GPU的高效计算
  - 代表框架：Caffe
- 优点
  - 提供了一定程度的可编程性
  - 计算性能高：支持GPU加速计算




```
name: "ResNet-152"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224

layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
                bias_term: false
        }
}

layer {
        bottom: "conv1"
        top: "conv1"
        name: "bn_conv1"
        type: "BatchNorm"
        batch_norm_param {
                use_global_stats: true
        }
}
```

早期的深度学习框架（-2010）

- 主要特点
  - 通过简单配置文件的形式定义神经网络
  - 模型可由一些常用layer构成一个简单的图
  - 框架提供每一个layer及其梯度计算实现
  - 支持多设备加速：CPU和GPU的高效计算
  - 代表框架：Caffe
- 优点
  - 提供了一定程度的可编程性
  - 计算性能高：支持GPU加速计算



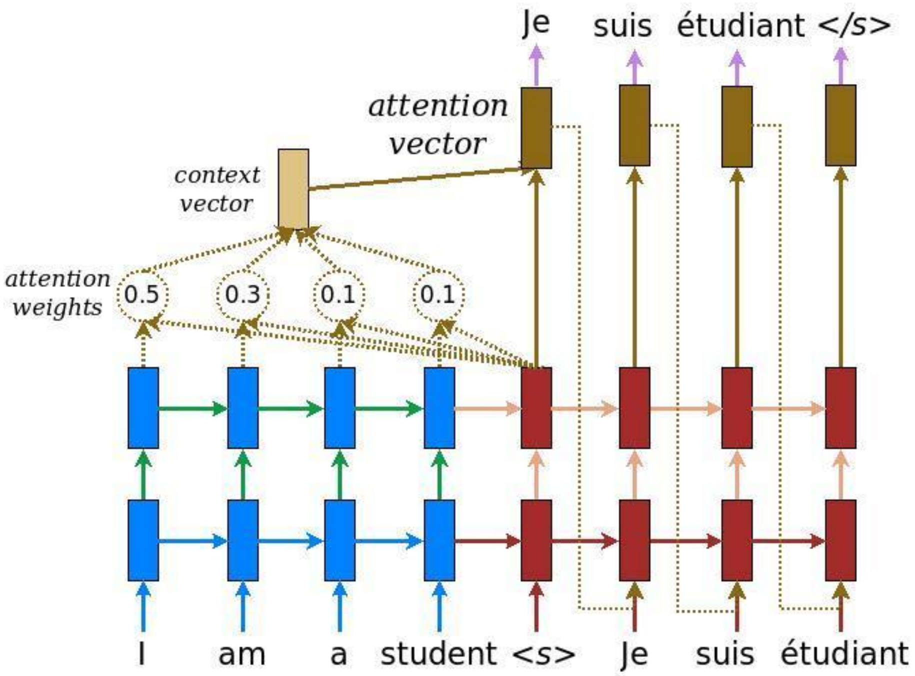

```
name: "ResNet-152"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224

layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
                bias_term: false
        }
}

layer {
        bottom: "conv1"
        top: "conv1"
        name: "bn_conv1"
        type: "BatchNorm"
        batch_norm_param {
                use_global_stats: true
        }
}
```

Efficiency

Flexibility

第一代框架的局限性（I）

- 灵活性的限制难以满足深度学习的快速发展
  - 层出不穷的新型网络结构（Layers）要求针对每种Layer都要重新实现其前向和后向计算函数
  - 如attention layer，Bach normalization layer, smapled softmax等



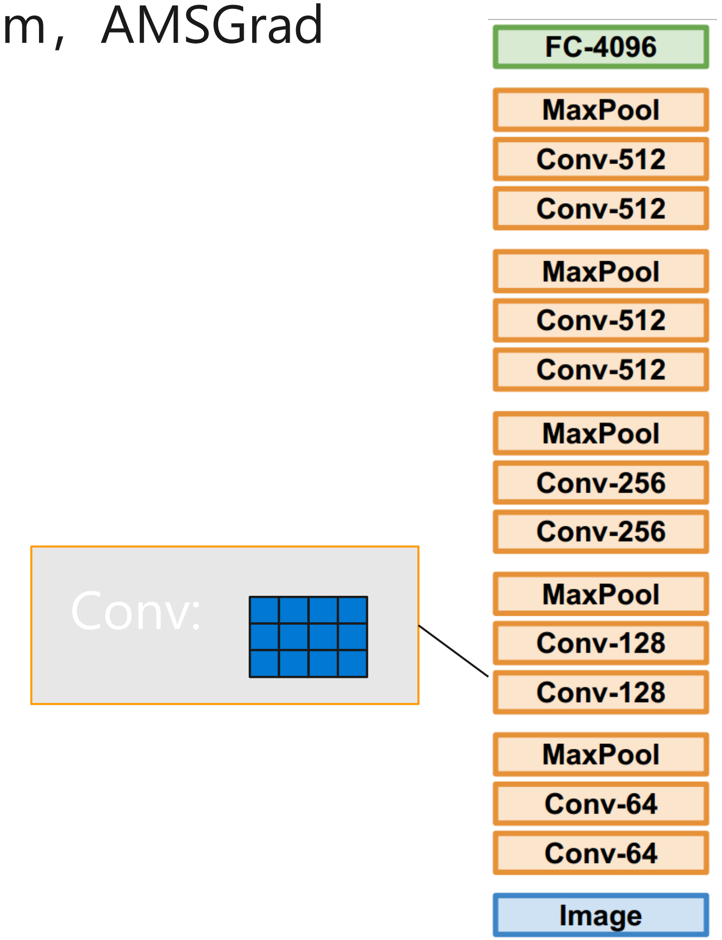

```
Class AttenionLayer<CPU>
{
 void forward(inputs..)
  {
  }
 void backward(inputs, grad)
  {
  }
….
};
Class AttenionLayer<GPU>
{
…
};

REGISTER_LAYER(“Attention”, AttenionLayer);
```

第一代框架的局限性(II)

- 新的优化器（Optimizer）要求对梯度和参数进行更通用复杂的运算
  - 如Adagrad，Adadelta，RMSprop，Adam，AdaMax，Nadam，AMSGrad

SGD: $w \leftarrow w - \eta \nabla_w$

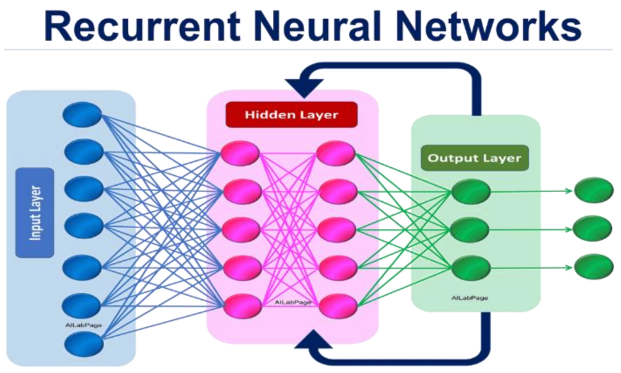
SGD with momentum: $w \leftarrow w - (\gamma \nabla_w^{t-1} + \eta \nabla_w^{t})$

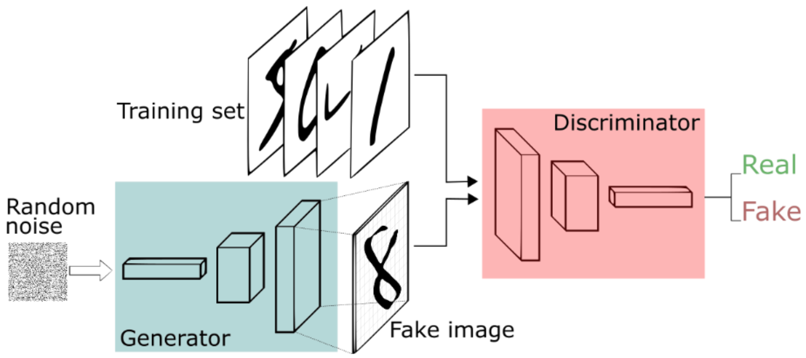


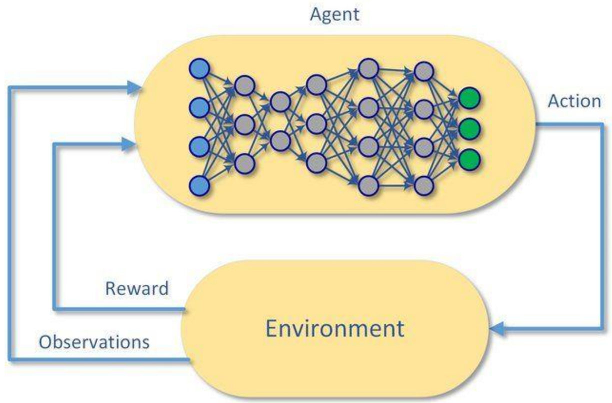
https://ruder.io/optimizing-gradient-descent/

第一代框架的局限性(III)

- 基于简单的"前向+后向" 的训练模式难以满足新的训练模式
  - 循环神经网络需要引入控制流，如RNN
  - 对抗神经网络需要两个网络交替训练
  - 强化学习模型需要和外部环境进行交互，如AlphaGo







第二代深度学习框架（2010-）

- 兼顾编程的灵活性和计算的高效性

TensorFlow

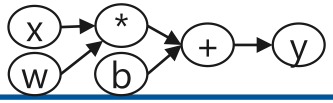
Microsoft CNTK

Caffe2

前端编程语言和接口
Python, Lua, R, C++

自动求导 （Auto Differentiation）

统一模型表示：计算流图



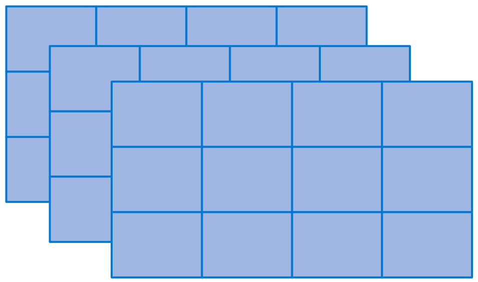
图的优化与调度执行
Batching, Cache, Overlap

内核代码优化与编译
GPU kernel, auto kernel generation

计算硬件
CPU, GPU, RDMA devices

基于数据流图（DAG）的计算框架

- 基本数据结构：Tensor （N维数组）
  - Tensor形状： [2, 3, 4]
  - 元素类型：int, float, string, etc.

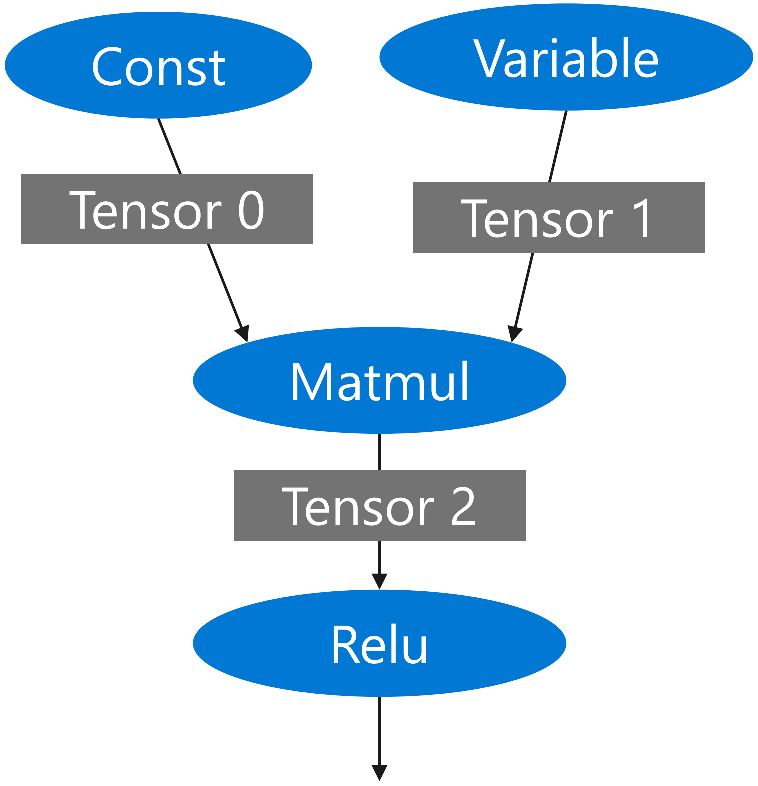


- 基本运算单元：Operator
  - 由最基本的代数算子组成
  - 每个Operator接收N个输入Tensor，并输出M个输出Tensor
  - TensorFlow中有>400个基本operator

Add	Log	While
Sub	MatMul	Merge
Mul	Conv	BroadCast
Div	BatchNorm	Reduce
Relu	Loss	Map
Tanh	Transpose	Reshape
Exp	Concatenate	Select
Floor	Sigmoid	.....

基于数据流图（DAG）的计算框架

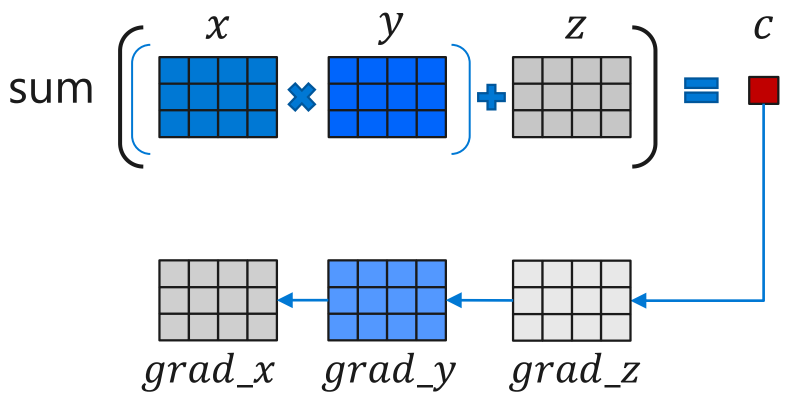
- 用数据流图表示计算逻辑和状态
  - 节点表示Operator
  - 边表示Tensor
- 计算状态（如参数）也是Operator
  - 如Variable Operator
- 特殊的Operator
  - 如：Switch, Merge, While 等用来构建控制流
- 特殊的边
  - 如：控制边用来表示节点之间的依赖关系



编程语言与编程模型

Numpy

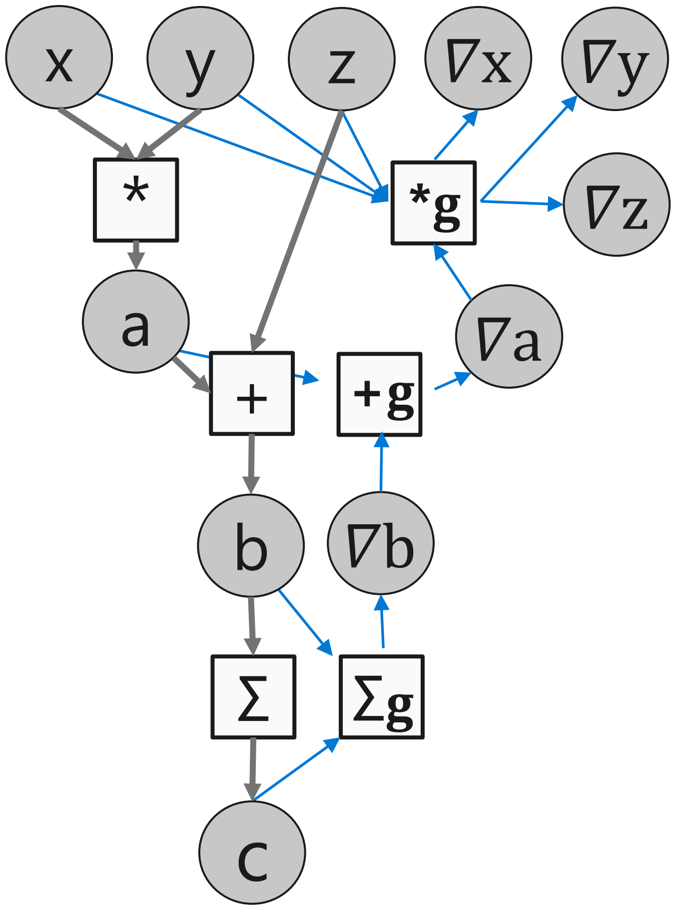
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)


grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



编程语言与编程模型

Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



TensorFlow

```
import tensorflow as tf
np.random.seed(0)

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x,grad_y,grad_z = tf.gradients(c, [x,y,z])

with tf.Session() as sess:
  sess.run([grad_z], feed_dict=values)
```

讨论：

- 数据流图表示深度学习模型的灵活性表现在哪些方面？
- 数据流图还有哪些好处？
- 你知道的其它基于数据流图的计算系统？

自动求导（AD）

- 深度学习计算的核心—计算参数更新的梯度

$$\mathrm{L}(w) = Loss(f(w, x_i), y_i) \rightarrow \boxed{\frac{\partial L(w)}{\partial w}}$$

- 求导计算是一个经典的问题

$$\mathrm{L}(x) = \exp(\exp(x) + \exp(x)^2) + \sin(\exp(x) + \exp(x)^2)$$

符号求导(Symbolic Differentiation)

- 根据简单函数的导数公式，以及导数变换公式精确的计算出一个复杂函数的导数形式化表示
  - 导数转换公式：$\frac{d}{dx}\left(f(x)+g(x)\right)=\frac{d}{dx}f(x)+\frac{d}{dx}g(x)$

    $$\frac{d}{dx}\left(f(x)g(x)\right)=\left(\frac{d}{dx}f(x)\right)g(x)+\left(\frac{d}{dx}g(x)\right)f(x)$$
  - 例如：$\mathrm{L}(x)=\exp(\exp(x)+\exp(x)^2)+\sin(\exp(x)+\exp(x)^2)$的导数可以推导出为
  - $\frac{dL}{dx}=\exp(\exp(x)+\exp(x)^2)((\exp(x)+2\exp(x)^2)+\cos(\exp(\exp(x)+\exp(x)^2)\ (\exp(x)+2\exp(x)^2)$
- 在深度学习中的应用问题
  - 深度学习网络非常大→ 待求导函数复杂→ 难以高效的求解
  - 深度中一些算子无法求导：如Relu, Switch

数值求导(Numerical Differentiation)

- 通过数值逼近的方法计算近似导数

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x + he_i) - f(x)}{h}$$

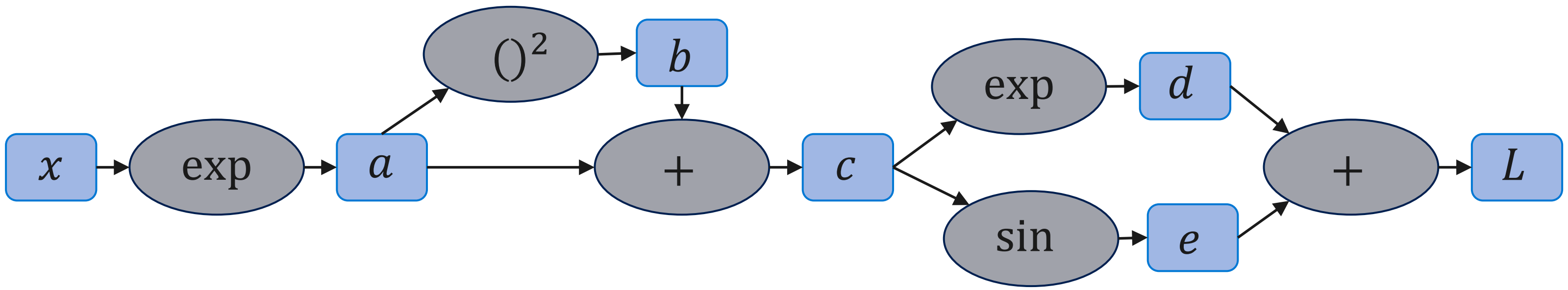
- 在深度学习中的应用问题
  - 由于数值计算中的截断和近似问题导致无法得到精确导数
  - 同样无法适用于深度中一些无法求导的算子：如Relu, Switch

自动求导（Auto Differentiation）

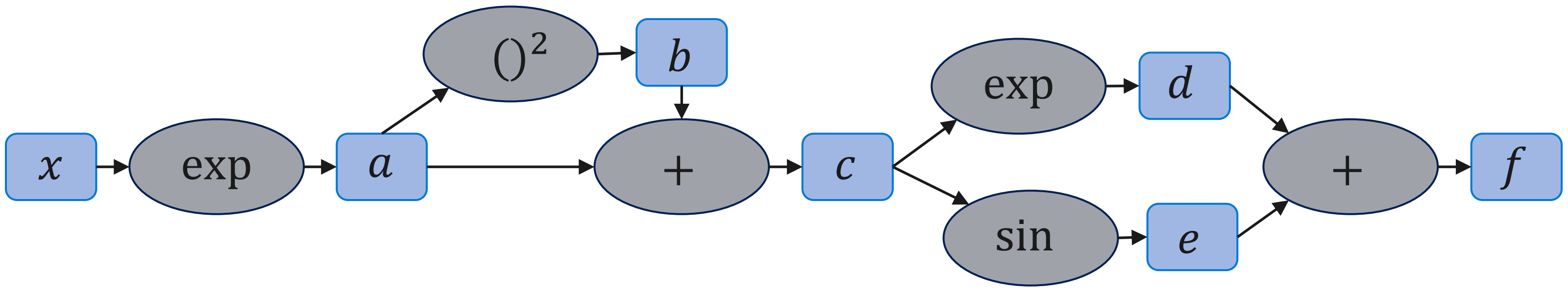
- 通过引入中间变量将一个复杂的函数分解成一系列基本函数
- 将这些基本函数构成一个计算流图

$$L(x) = \exp(\exp(x) + \exp(x)^2) + \sin(\exp(x) + \exp(x)^2)$$

$$\begin{aligned} a &= \exp(x) \\ b &= a^2 \\ c &= a + b \\ d &= \exp(c) \\ e &= \sin(c) \\ L &= d + e \end{aligned}$$



自动求导（Auto Differentiation ）(II)

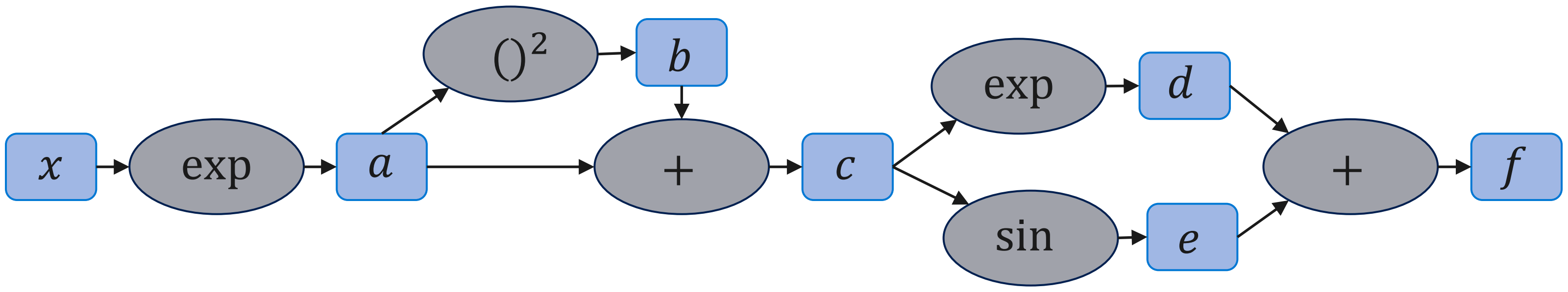


$$
\begin{aligned}
a &= \exp(x) \\
b &= a^2 \\
c &= a + b \\
d &= \exp(c) \\
e &= \sin(c) \\
f &= d + e
\end{aligned}
$$

$$
\begin{aligned}
\frac{df}{dd} &= 1 \\
\frac{df}{de} &= 1 \\
\frac{df}{dc} &= \frac{df}{dd}\frac{dd}{dc} + \frac{df}{de}\frac{de}{dc} \\
\frac{df}{db} &= \frac{df}{dc}\frac{dc}{db} \\
\frac{df}{da} &= \frac{df}{dc}\frac{dc}{da} + \frac{df}{db}\frac{db}{da} \\
\frac{df}{dx} &= \frac{df}{da}\frac{da}{dx}
\end{aligned}
$$

$$
\begin{aligned}
\frac{df}{dd} &= 1 \\
\frac{df}{de} &= 1 \\
\frac{df}{dc} &= \frac{df}{dd}\exp(c) + \frac{df}{de}\cos(c) \\
\frac{df}{db} &= \frac{df}{dc} \\
\frac{df}{da} &= \frac{df}{dc} + \frac{df}{db}2a \\
\frac{df}{dx} &= \frac{df}{da}\exp(x).
\end{aligned}
$$

自动求导（AD）(III)



$$
\begin{aligned}
a &= \exp(x)\\
b &= a^2\\
c &= a + b\\
d &= \exp(c)\\
e &= \sin(c)\\
f &= d + e
\end{aligned}
$$

$$
\begin{aligned}
\frac{df}{dd} &= 1\\
\frac{df}{de} &= 1\\
\frac{df}{dc} &= \frac{df}{dd}\exp(c) + \frac{df}{de}\cos(c)\\
\frac{df}{db} &= \frac{df}{dc}\\
\frac{df}{da} &= \frac{df}{dc} + \frac{df}{db}2a\\
\frac{df}{dx} &= \frac{df}{da}\exp(x).
\end{aligned}
$$

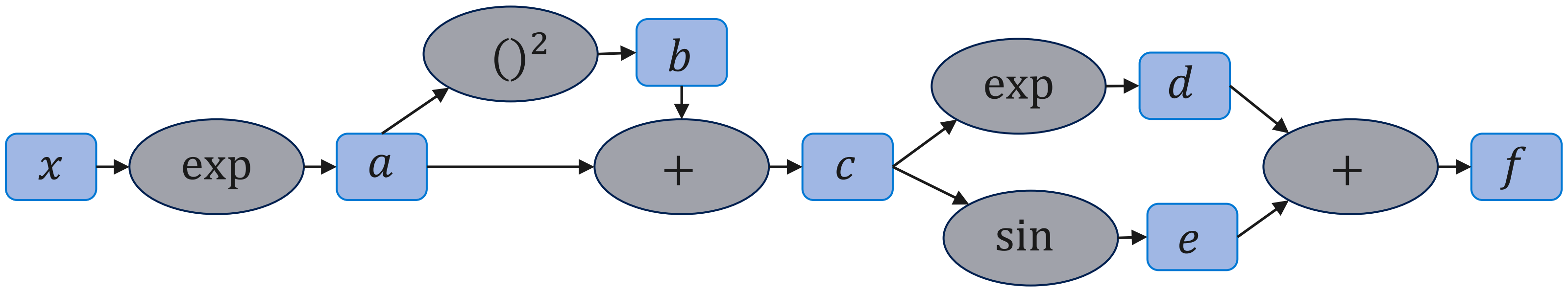
Forward propagation

Backward propagation

讨论：

- Forward 和 Backward propagation计算量是否一样?
- 深度学习中为什么大多使用Backward propagation?

如何在深度学习框架中实现自动求导



前向计算并保留中间计算结果
根据BP的原理依次计算出中间导数

主要问题：
需要保存大量中间计算结果

$$
\begin{aligned}
\frac{df}{dd} &= 1\\
\frac{df}{de} &= 1\\
\frac{df}{dc} &= \frac{df}{dd}\exp(c) + \frac{df}{de}\cos(c)\\
\frac{df}{db} &= \frac{df}{dc}\\
\frac{df}{da} &= \frac{df}{dc} + \frac{df}{db}2a\\
\frac{df}{dx} &= \frac{df}{da}\exp(x).
\end{aligned}
$$

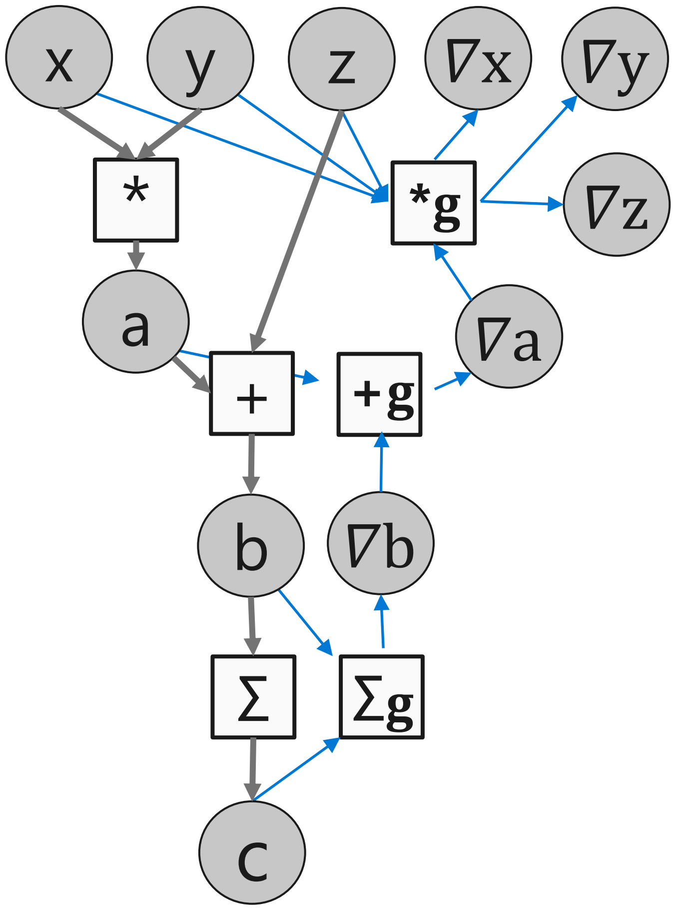
如何在深度学习框架中实现自动求导（II）

将导数的计算也表示成数据流图

优点：

方便全局图优化

节省内存

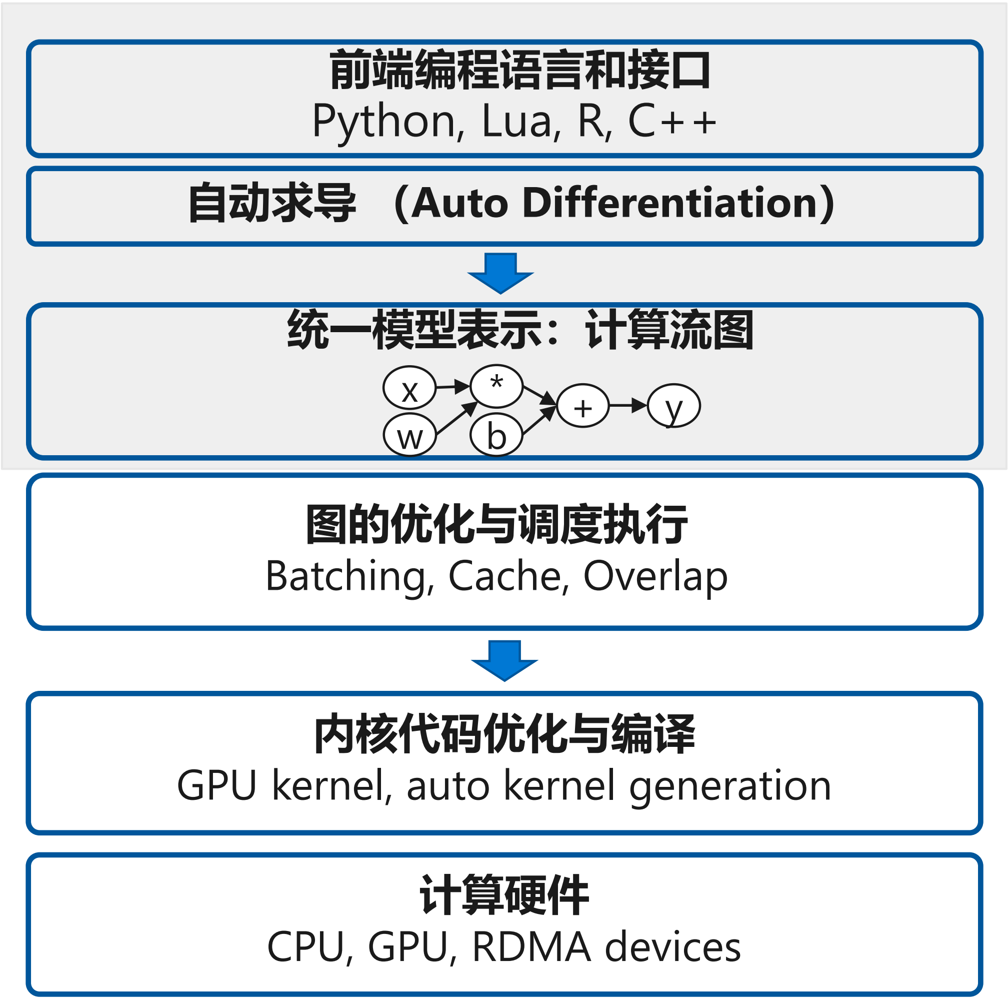


TensorFlow

```
import tensorflow as tf
np.random.seed(0)

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x,grad_y,grad_z = tf.gradients(c, [x,y,z])

with tf.Session() as sess:
  sess.run([grad_z], feed_dict=values)
```

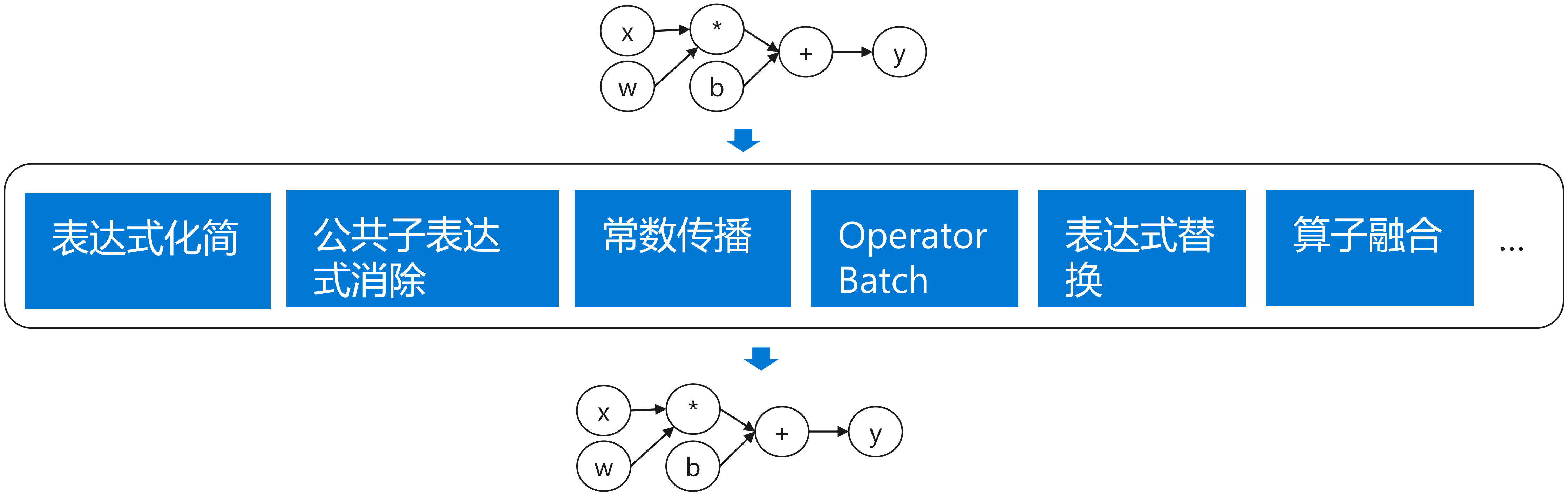
小结:

- **模型表示**：数据流图
- **前端语言**：用来构建数据流图
- **自动求导**：基于backpropagation的原理自动构建求导数据流图



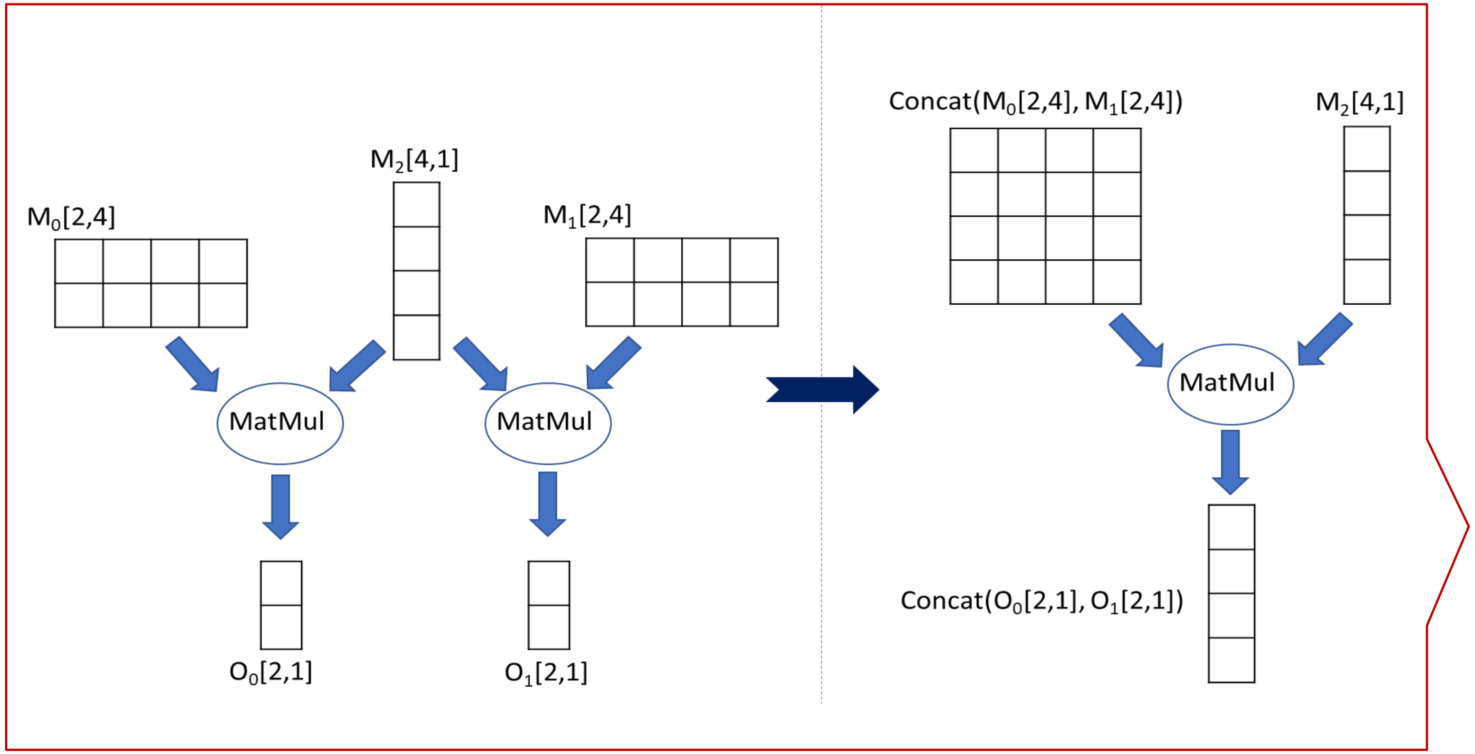
图优化

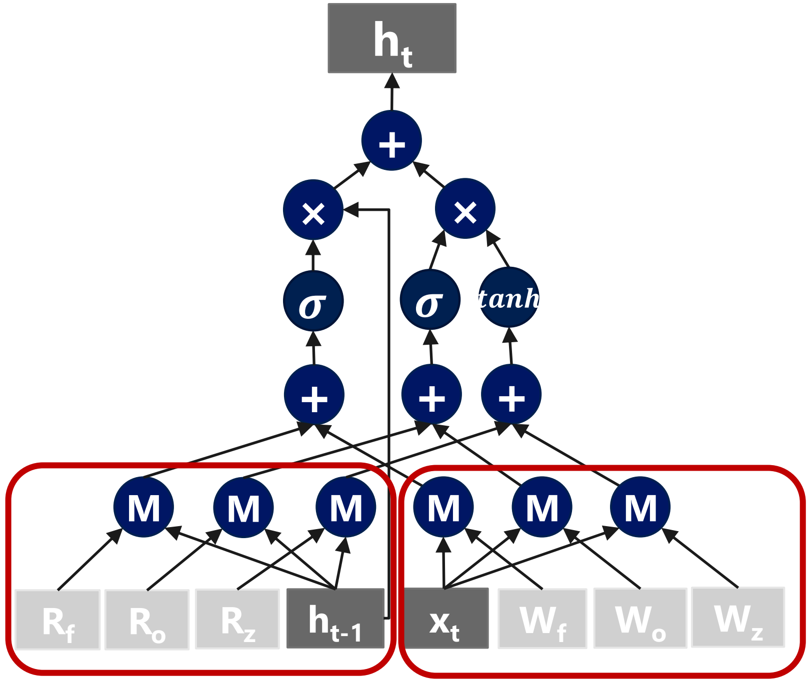
- “先定义后执行”的模式允许框架在计算前看到全图信息
- 数据流图作为深度学习框架中的高层中间表示，可以允许任何等价图优化Pass去化简计算流图或提高执行效率



图优化：GEMM自动融合

- Batch same-type operators to leverage GPU massive parallelism

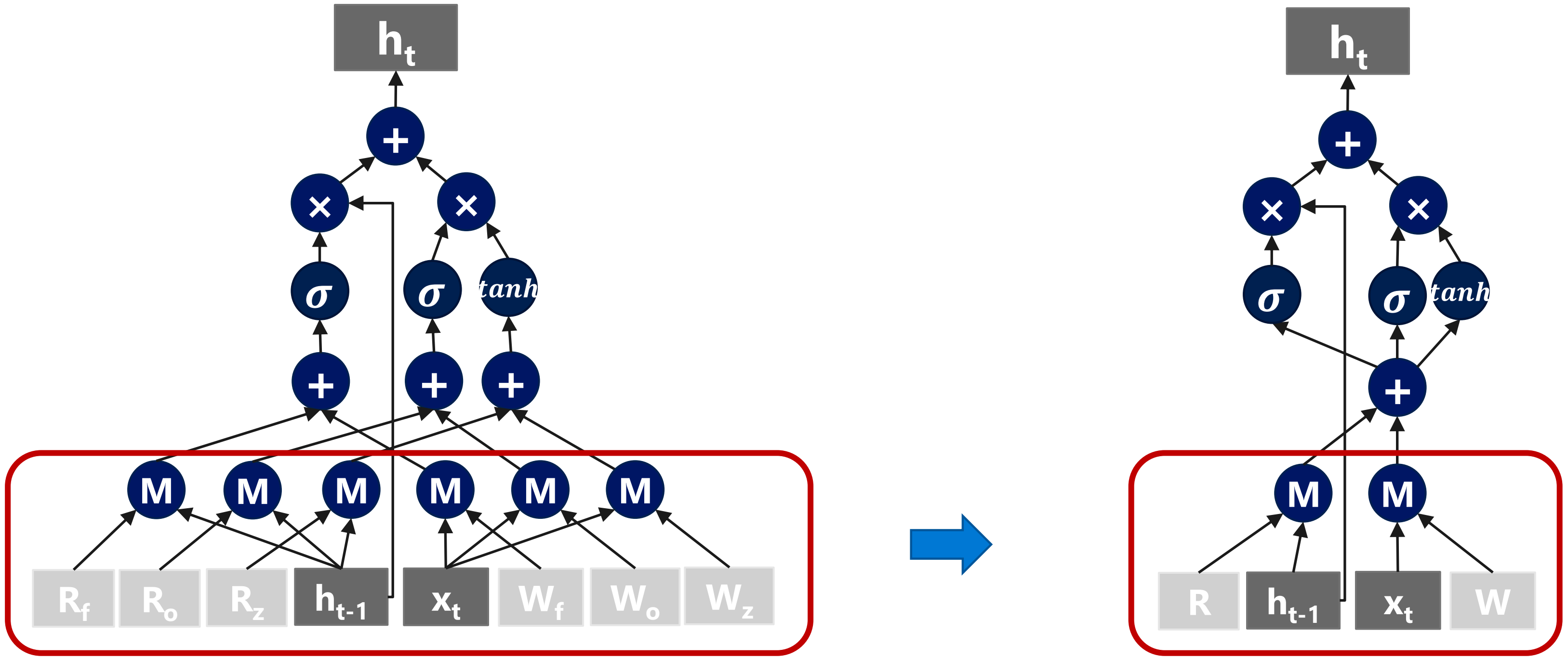




Data-flow graph of a GRU cell

图优化：GEMM自动融合

- Batch same-type operators to leverage GPU massive parallelism

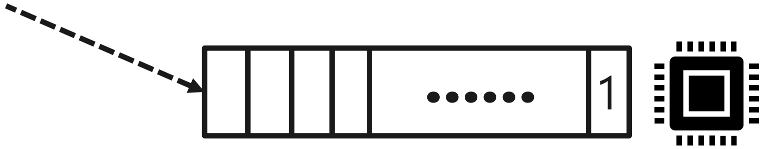


Data-flow graph of a GRU cell

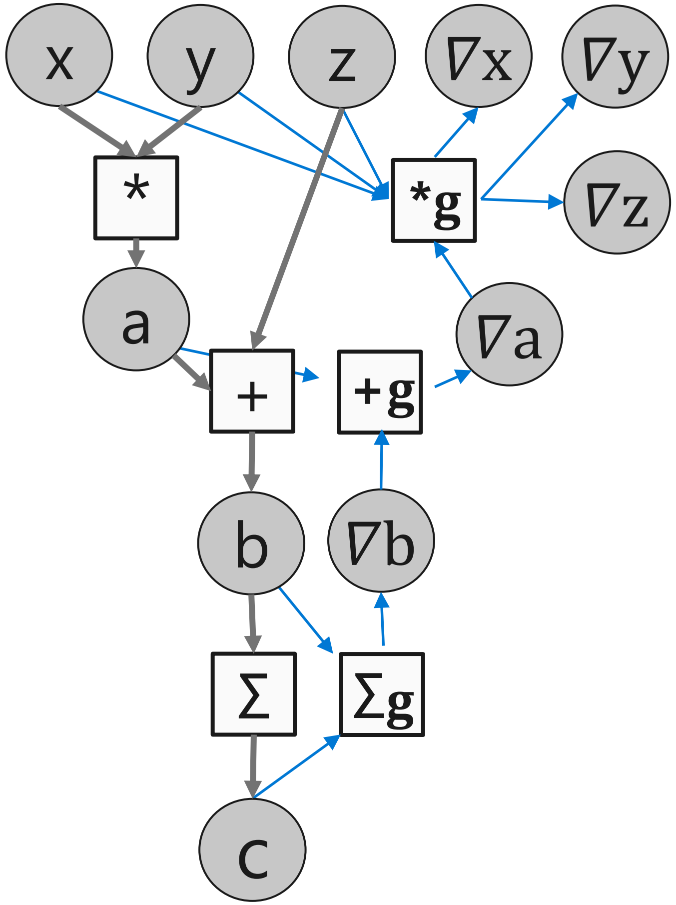
数据流图的调度与执行

根据依赖关系，依次调度运行代码

1. Multiply(x, y) -> a
2. Add(z, a) -> b
3. ReduceSum(b) -> c
4. ReduceSum_grad(c, b) -> b_delta
5. Sum_grad(b_delta, a) -> a_delta
6. Add_grad(a_delta, z) -> z_delta
7. Multiply(a_delta, x, y) -> x_delta, y_delta

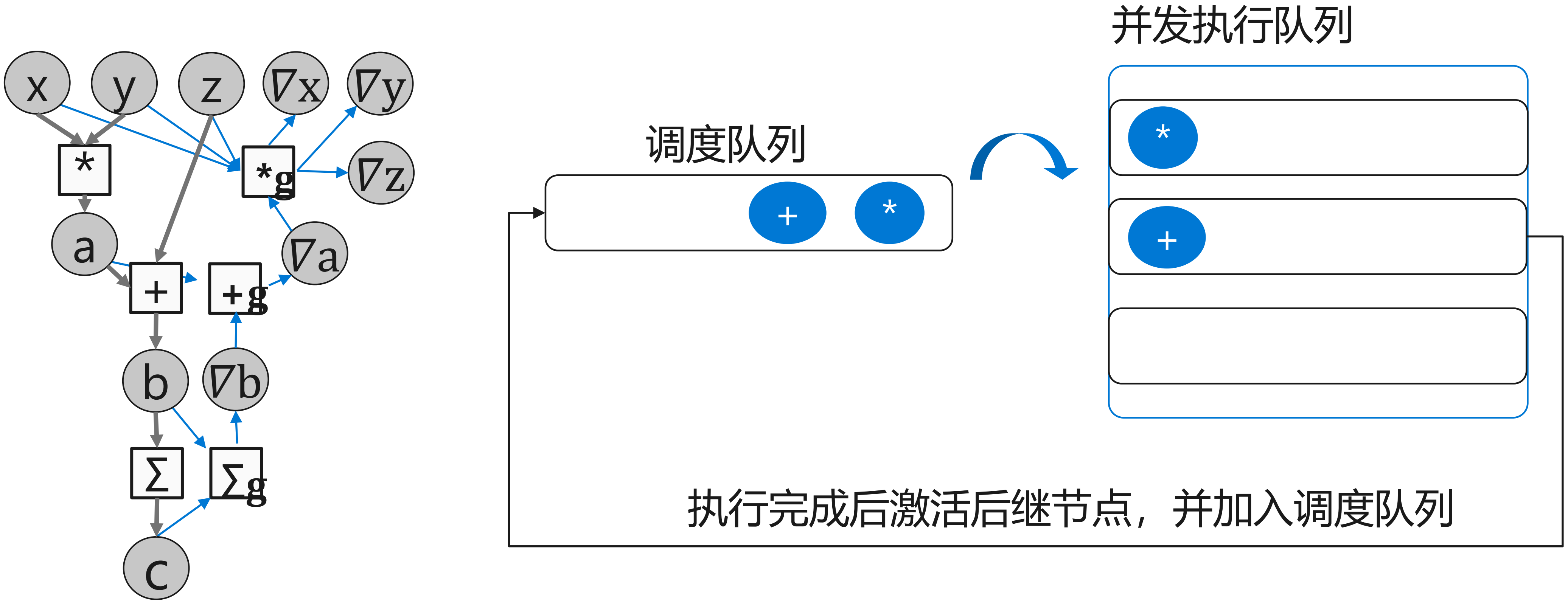


Operators



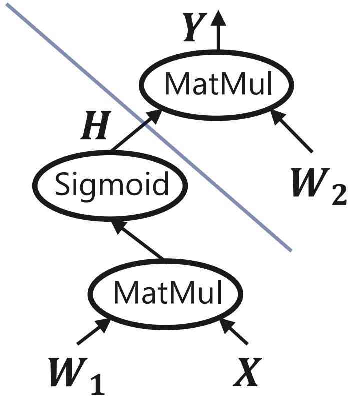
数据流图的并发执行

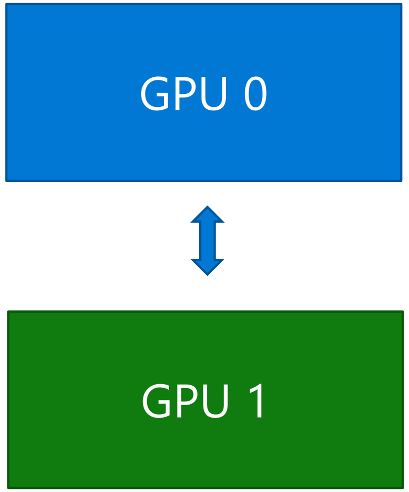
- 数据流图准确的描述了算子之间的依赖关系
- 根据数据流图找到相互独立的算子进行并发调度，提高计算的并行性



数据流图的划分与设备放置

- 显式图划分

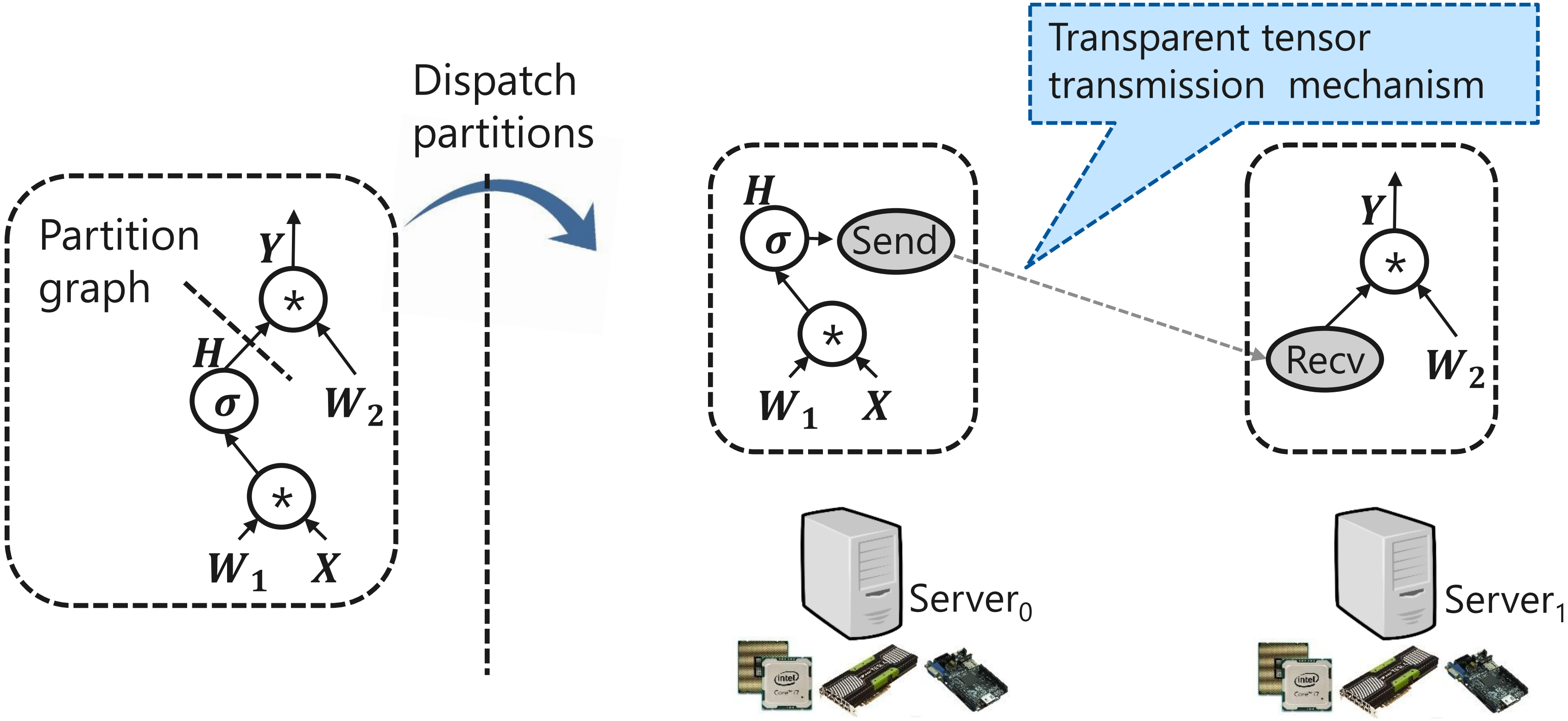





```python
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3000, 4000

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

数据流图的划分与设备放置

- 跨设备的边将被自动替换成一组Send/Recv operators



计算内核(Kernel)与多硬件支持

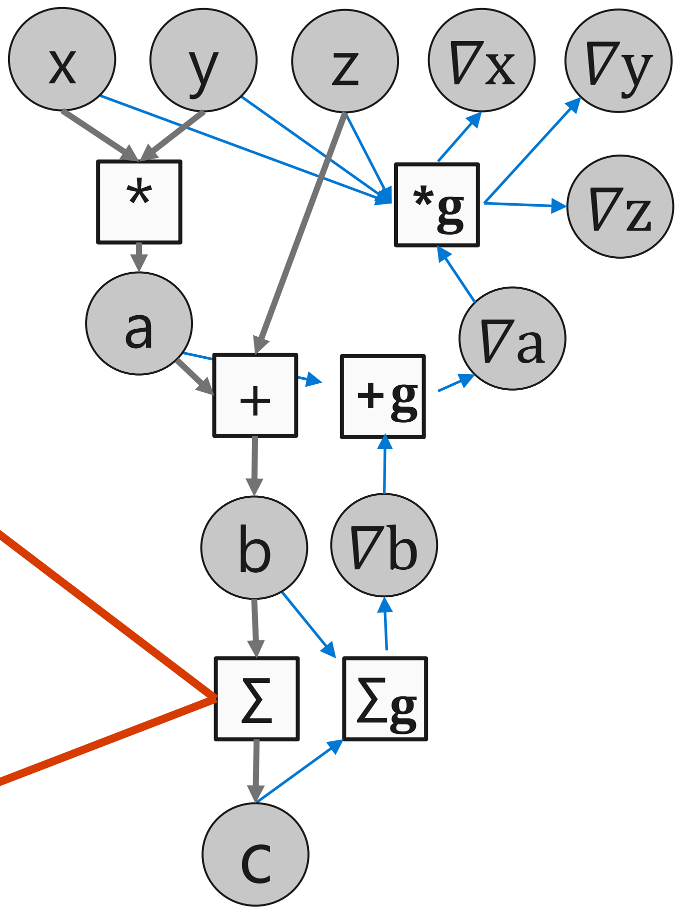
- 内核(Kernel)是定义了一个算子在某种具体设备的计算实现
- 每个Operator都可以注册多个Kernel
  - 根据计算设备的不同可以有：GPU kernel, CPU kernel, ASIC kernel, FPGA kernel
  - 根据数据类型不同可以有：float kernel, int kernel, half kernel等
  - 根据Operator属性不同也可以有多种kernel
- 运行时框架会自动根据Operator的设备类型、数据类型和属性选择对应的kernel来执行

例：TensorFlow中注册的CPU/GPU kernel

```
template <typename Device, typename OUT_T, typename IN_T,
          typename ReductionAxes, typename Scalar>
struct ReduceEigenImpl<Device, OUT_T, IN_T, ReductionAxes,
                       functor::MeanReducer<Scalar>> {
  void operator()(const Device& d, OUT_T out, IN_T in,
                  const ReductionAxes& reduction_axes,
                  const functor::MeanReducer<Scalar>& reducer) {
    static_assert(std::is_same<Scalar, typename OUT_T::Scalar>::value, "");
    Eigen::internal::SumReducer<Scalar> sum_reducer;
    out.device(d) = in.reduce(reduction_axes, sum_reducer) /
                    static_cast<Scalar>(in.size() / out.size());
  }
};

// T: the data type
// REDUCER: the reducer functor
// NUM_AXES: the number of axes to reduce
// IN_DIMS: the number of dimensions of the input tensor
#define DEFINE(T, REDUCER, IN_DIMS, NUM_AXES)                        \
  template void ReduceFunctor<GPUDevice, REDUCER>::Reduce(           \
      OpKernelContext* ctx, TTypes<T, IN_DIMS - NUM_AXES>::Tensor out, \
      TTypes<T, IN_DIMS>::ConstTensor in,                            \
      const Eigen::array<Index, NUM_AXES>& reduction_axes,           \
      const REDUCER& reducer);
```

CPU code

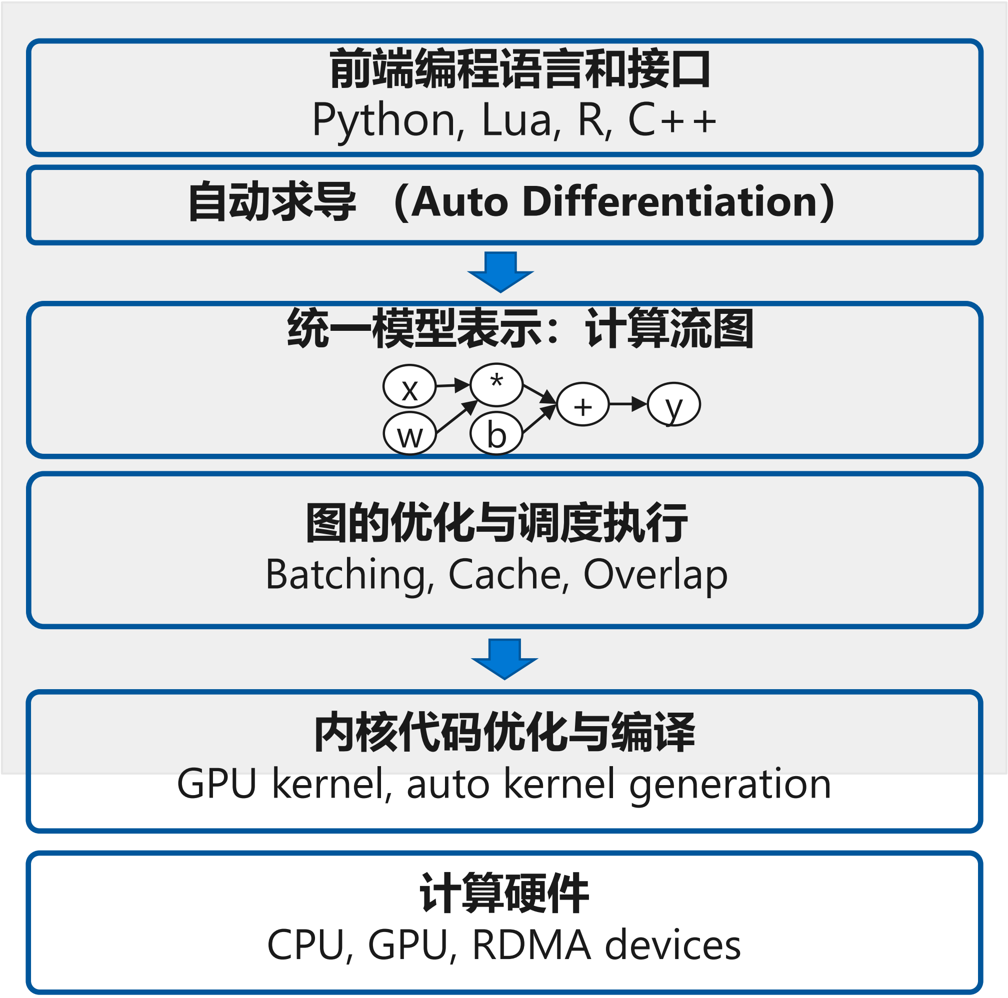
GPU code



小结:

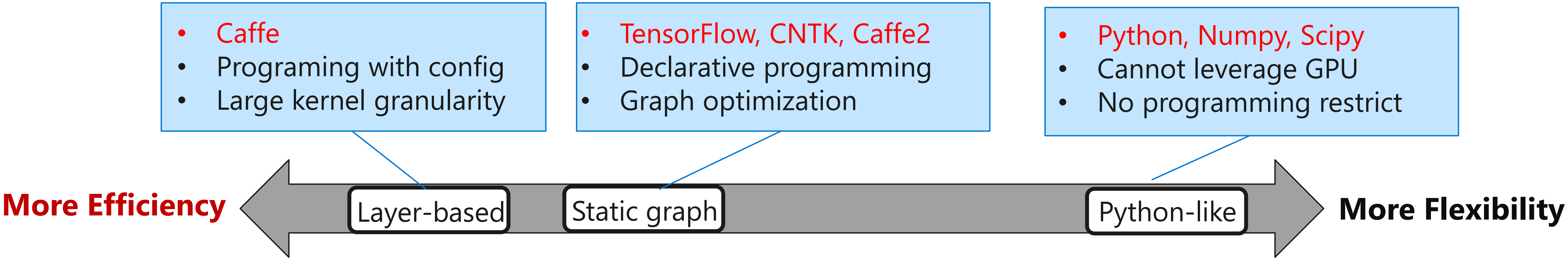
- **模型表示**：数据流图
- **前端语言**：用来构建数据流图
- **自动求导**：基于backpropagation的原理自动构建求导数据流图

- **图的优化**: 图化简
- **调试执行**: 并发调度
- **设备放置**: 显式和隐式图切分
- **算子内核**: 模块化支持多设备



基于静态数据流图的计算框架的优点和缺点

- 优点
  - 计算效率较高：静态定义图后可以进行全局图优化
  - 内存使用效率高：可以根据数据流图准确分析内存使用生命周期
  - 较好的灵活性：可以表示大部分可由基本算子组成的网络
- 缺点
  - 可调试性：先申明后执行导致不能实时得到计算结果，难以Debug
  - 表达的灵活性：受限于算子集合，如对于带有控制流的网络支持不友好或无法支持



基于动态数据流图的计算框架

- 边定义边执行-Define-by-run
  - 不预先定义计算流图
- 特点：
  - 模型可由任意Layer构成
  - 用户自己定义Layer可以内置layer组成
  - 可实时得到计算结果
  - 控制流由高层语言表示：如Python
  - 支持多设备加速：CPU和GPU的高效计算
  - 代表框架：PyTorch
- 优点
  - 提供了灵活的可编程性和可调试性

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda())
y = Variable(torch.randn(N, D).cuda())
z = Variable(torch.randn(N, D).cuda())

for i in range(10):
  a = x * y
  b = a + z
  c = c + torch.sum(b)


c.backward()
```

基于动态数据流图的计算框架： PyTorch Example

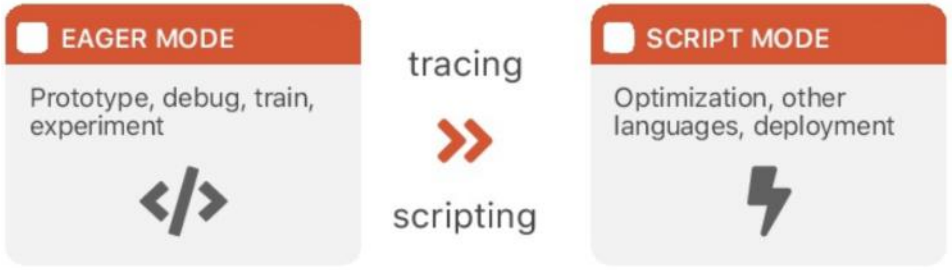
```python
class LinearLayer(Module):
    def __init__(self, in_sz, out_sz):
        super().__init__()
        t1 = torch.randn(in_sz, out_sz)
        self.w = nn.Parameter(t1)
        t2 = torch.randn(out_sz)
        self.b = nn.Parameter(t2)

    def forward(self, activations):
        t = torch.mm(activations, self.w)
        return t + self.b
```

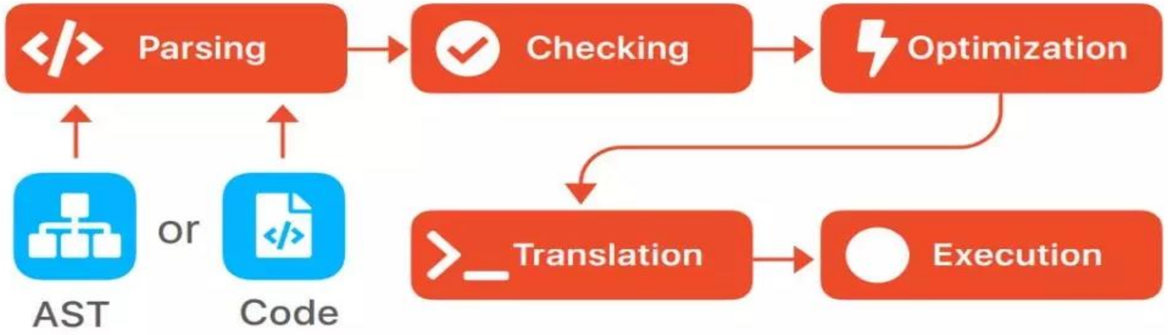
```python
class FullBasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 128, 3)
        self.fc = LinearLayer(128, 10)

    def forward(self, x):
        t1 = self.conv(x)
        t2 = nn.functional.relu(t1)
        t3 = self.fc(t1)
        return nn.functional.softmax(t3)
```

执行性能存在的问题与优化

- 由于丢失了全局数据流图，天然损失了全局图优化的好处
  - 无法进行全图化简优化
  - 无法进行全图精确内存分配管理
- 先天不足、后天来补：
  - **高效的C++核心**
    - 所以核心Layer的后端都是通过C++实现，对GPU也有较好的内核实现
  - **分离控制流和数据流的执行**
    - 控制流通过高级Python语言来执行，计算量较大的数据流部分再由C++执行
    - 通过GPU的异步接口调度GPU内核，从而实现调度和执行逻辑的重叠
  - **Tensor分配的缓存机制**
    - 通过对已分配的Tensor进行缓存来减少动态内存分配时的开销
  - **多进程执行**
    - 使用多进程来避免Python多线程的同步锁

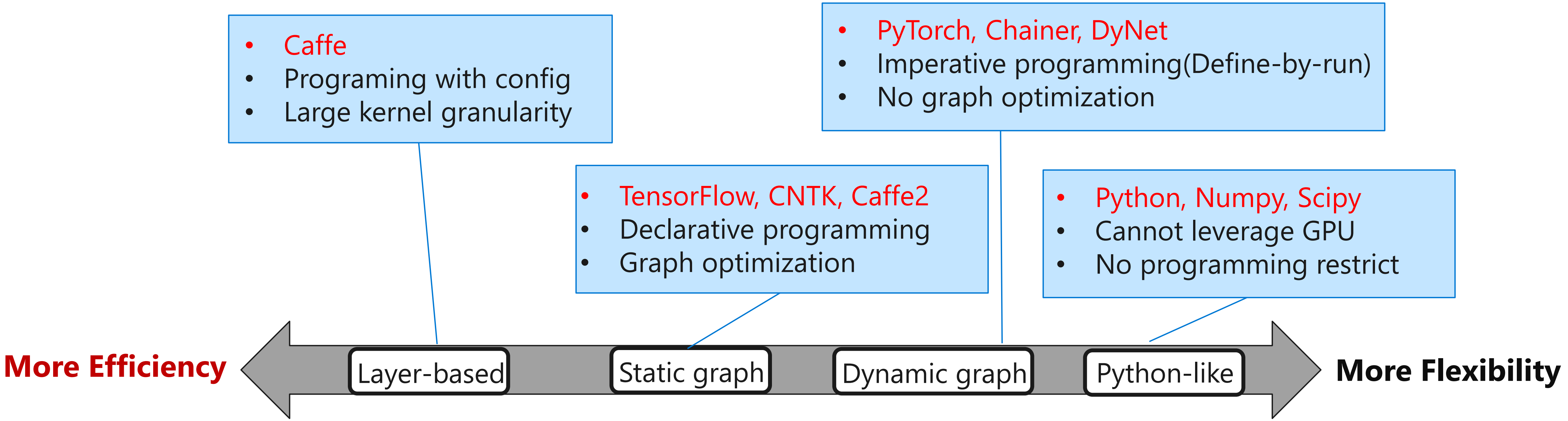
执行效率优化-JIT






```
def dumb_function(x):
    return x.t().t()

>>> traced_fn = torch.jit.trace(dumb_function,
...                             torch.ones(2,2))
>>> traced_fn.graph_for(torch.ones(2,2))
graph(%x : Float(*, *)) {
return (%x);
}
```

小结



Compiler is used to optimize general framework to be more efficient, while keeping the existing flexibility!

计算框架的进步

	Gen 1 *pre-2010*	Gen 2 *2010-present*	**Gen 3** *present-*
Frameworks	**Custom** purpose machine learning **algorithms** Theano DisBelief Caffe	**Deep learning frameworks** provide easier ways to leverage various libraries	**Machine Learning Language and Compiler** A Full-Featured Programming Language for ML: Expressive and flexible Control flow, recursion, sparsity Powerful Compiler Infrastructure: Code optimization, sparsity optimization, hardware targeting
Hardware	**Algebra & linear libs** • CPU • GPU	AI framework Dense matmul engine	SIMD → MIMD Sparsity Support Control Flow and Dynamicity Associated Memory

参考阅读

- Large Scale Distributed Deep Networks, NIPS’12
- Caffe: Convolutional Architecture for Fast Feature Embedding, MM’14
- TensorFlow: A System for Large-Scale Machine Learning, OSDI’16
- PyTorch: An Imperative Style, High-Performance Deep Learning Library, NIPS’19
- Theano: A Python framework for fast computation of mathematical expressions, arXiv 16
- Automatic Differentiation in Machine Learning: a Survey, [Link]
- Automatic Differentiation and Neural Networks, [Link]

课后作业

- 推荐补充阅读材料