Automatic Differentiation and Deep Learning

with examples from PyTorch

Soumith Chintala

automatic differentiation

Automatic Differentiation

Given y = f(x1, x2), the ability to compute dy/dx1, dy/dx2

Automatic Differentiation

Given $y = f(x1, x2)$, the ability to compute $dy/dx1$, $dy/dx2$

- using the chain rule in the process

Automatic Differentiation

Given y = f(x1, x2), the ability to compute dy/dx1, dy/dx2

- using the chain rule in the process
- Two flavors:
  - forward-mode
  - reverse-mode

Forward-mode autodiff

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \text{ at } (x_1, x_2) = (2, 5)$$

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 5$
v_1	$= \ln v_{-1}$	$= \ln 2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$
v_3	$= \sin v_0$	$= \sin 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$
y	$= v_5$	$= 11.652$

Forward Tangent (Derivative) Trace		
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
$\dot{v}_0$	$= \dot{x}_2$	$= 0$
$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$\boldsymbol{\dot{y}}$	$\boldsymbol{= \dot{v}_5}$	$\mathbf{= 5.5}$

Example from: Baydin, Pearlmutter et. al. Automatic differentiation in machine learning: a survey

Forward-mode autodiff

- Computes Jacobian-vector products

Forward-mode autodiff

- Computes Jacobian-vector products
- Each evaluation gives one row of the Jacobian

Forward-mode autodiff

- Computes Jacobian-vector products
- Each evaluation gives one row of the Jacobian
- Typically used when: dimensionality of y >> dimensionality of x

Forward-mode autodiff

- Computes Jacobian-vector products
- Each evaluation gives one row of the Jacobian
- Typically used when: dimensionality of y >> dimensionality of x

- Popular software implementations:
  - HIPS/autograd
  - JAX by Google
  - Flux.jl

Reverse-mode autodiff

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \text{ at } (x_1, x_2) = (2, 5)$$

Forward Primal Trace		
$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace		
$\bar{x}_1 = \bar{v}_{-1}$		$= 5.5$
$\bar{x}_2 = \bar{v}_0$		$= 1.716$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$	$= 1$	

Example from: Baydin, Pearlmutter et. al. Automatic differentiation in machine learning: a survey

Reverse-mode autodiff

- Computes Vector-Jacobian products

Reverse-mode autodiff

- Computes Vector-Jacobian products
- Each evaluation gives one column of the Jacobian

Reverse-mode autodiff

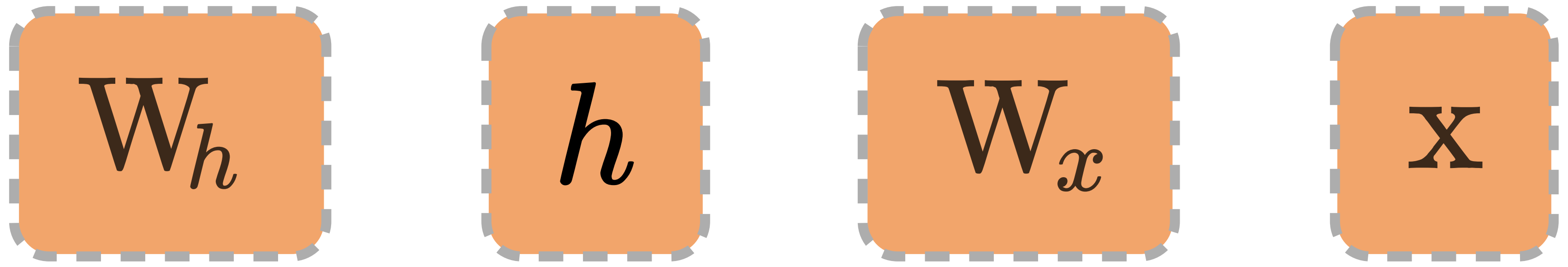
- Computes Vector-Jacobian products
- Each evaluation gives one column of the Jacobian
- Typically used when: dimensionality of x >> dimensionality of y
  - Like in deep learning

Reverse-mode autodiff

- Computes Vector-Jacobian products
- Each evaluation gives one column of the Jacobian
- Typically used when: dimensionality of x >> dimensionality of y
  - Like in deep learning

- Popular software implementations
  - All deep learning frameworks (PyTorch, TensorFlow, MXNet, Caffe, etc.)
  - HIPS/autograd
  - Jax by Google
  - Flux.jl

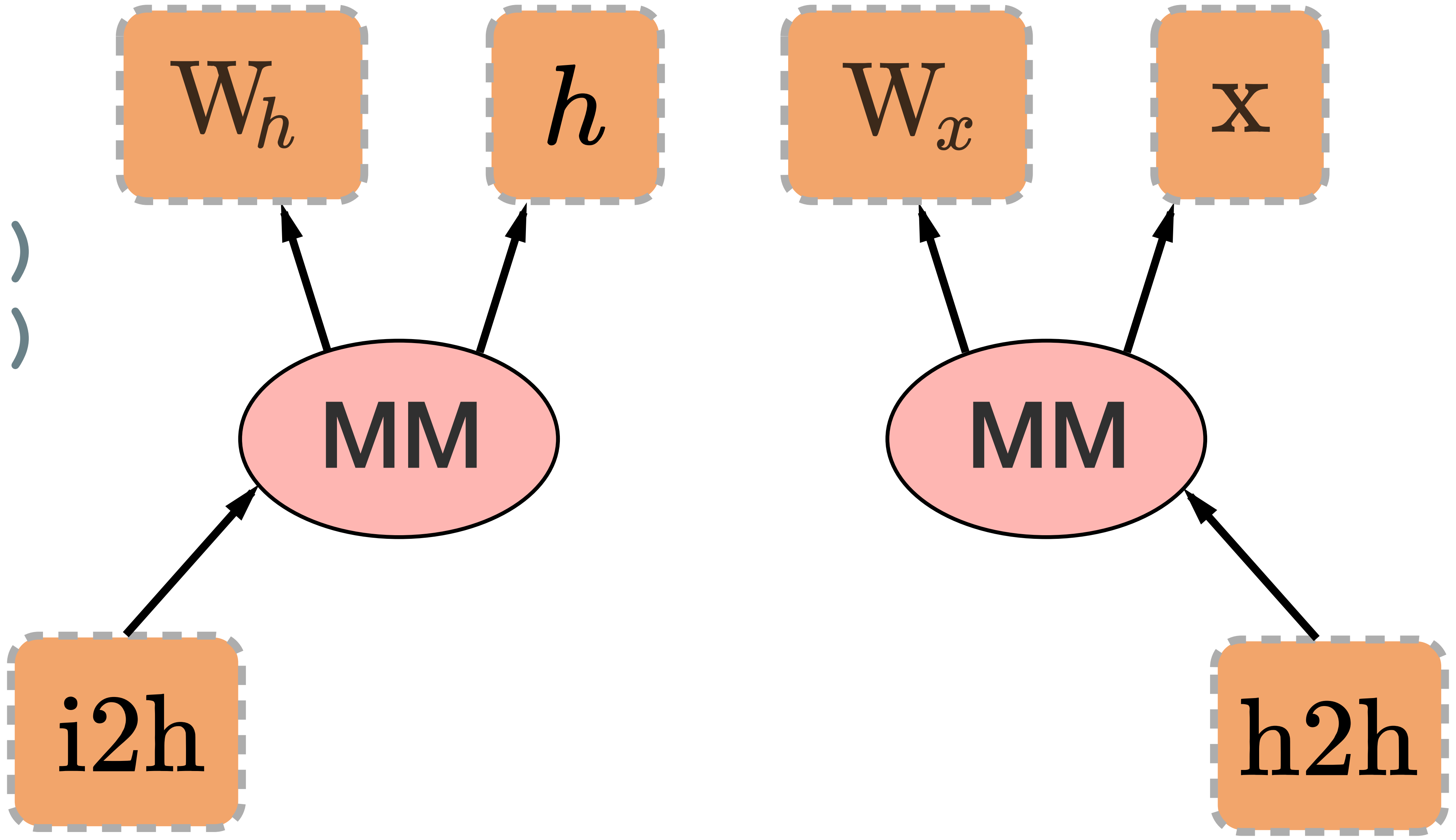
PyTorch Autograd




```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

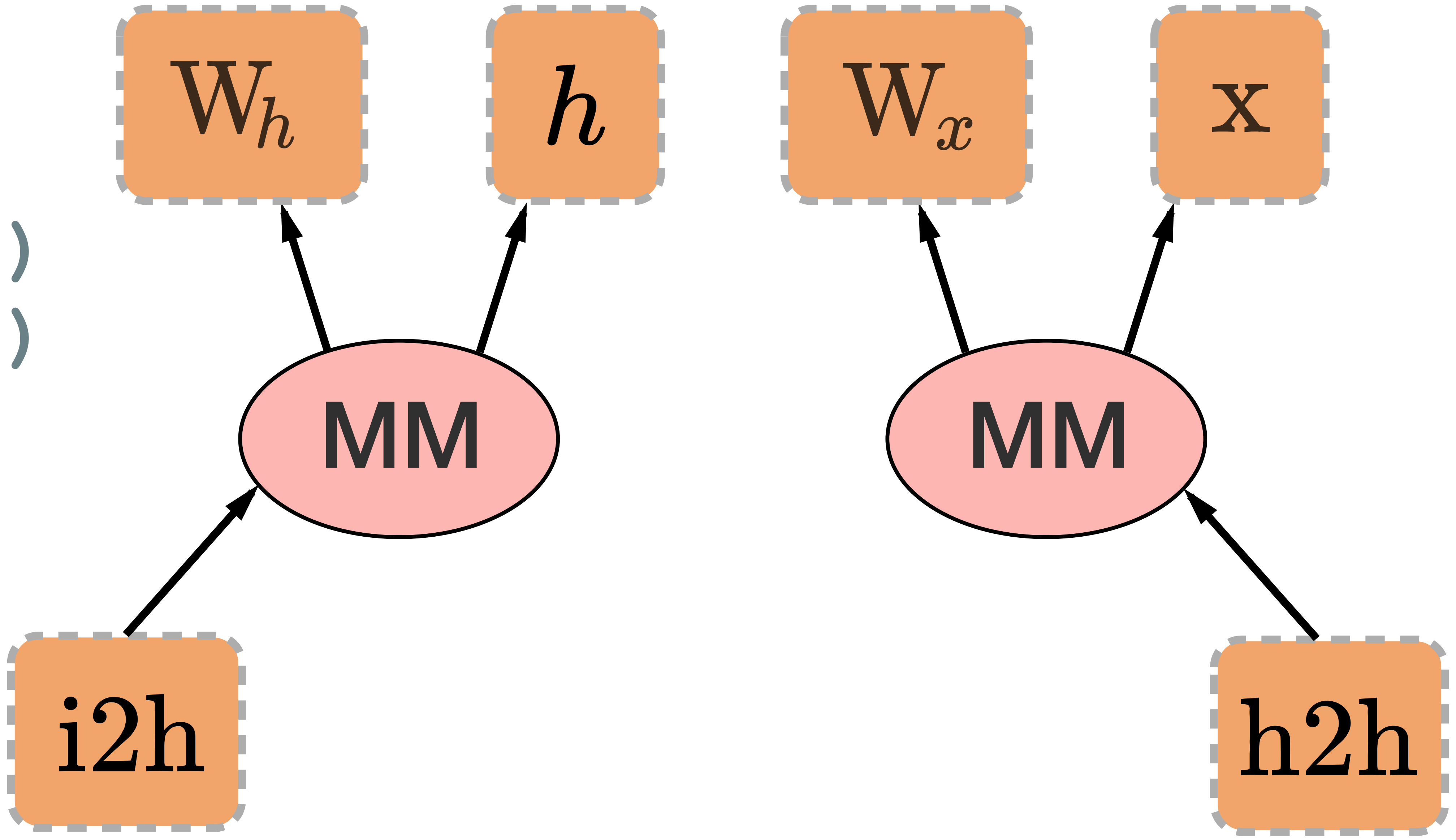
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```



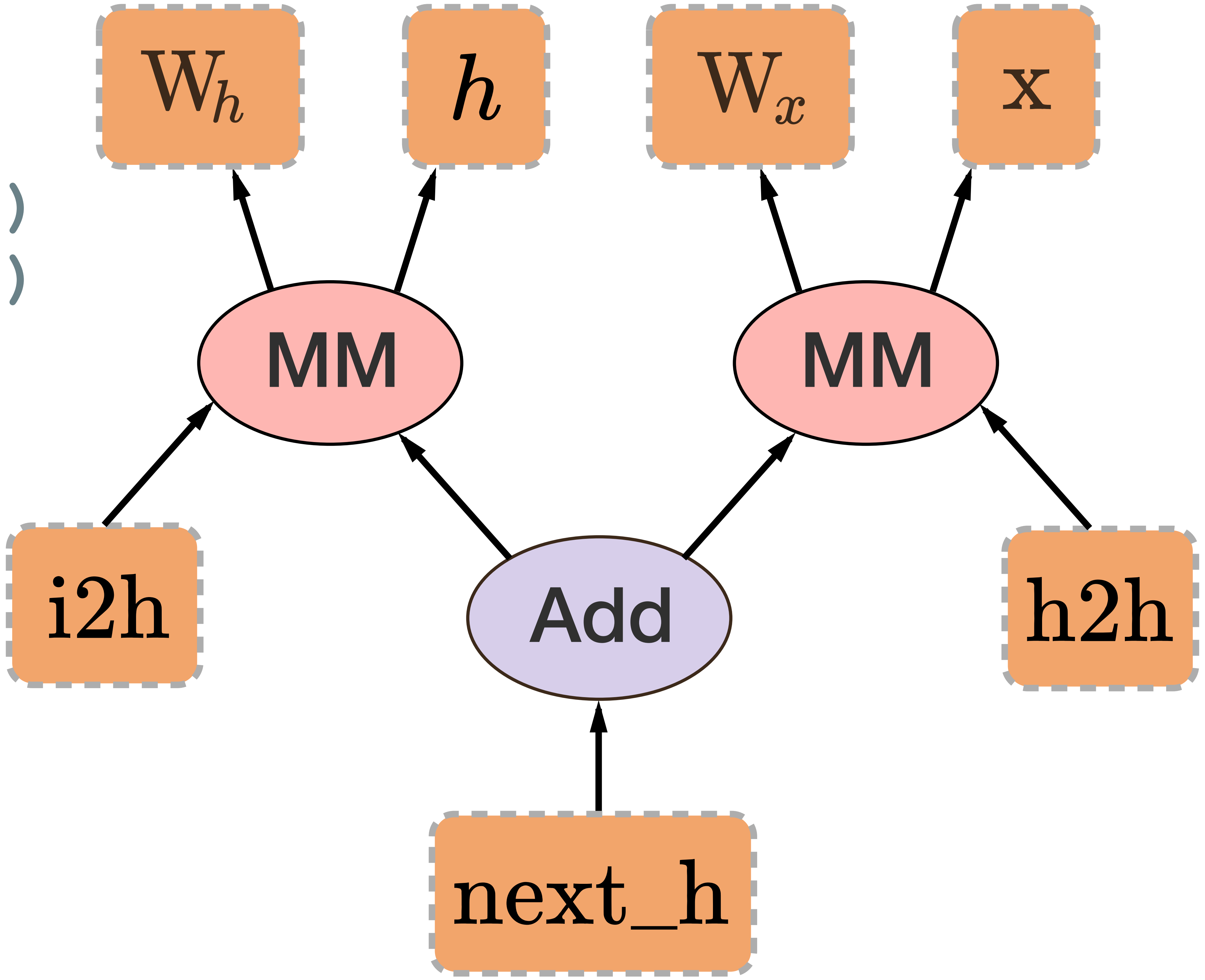
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



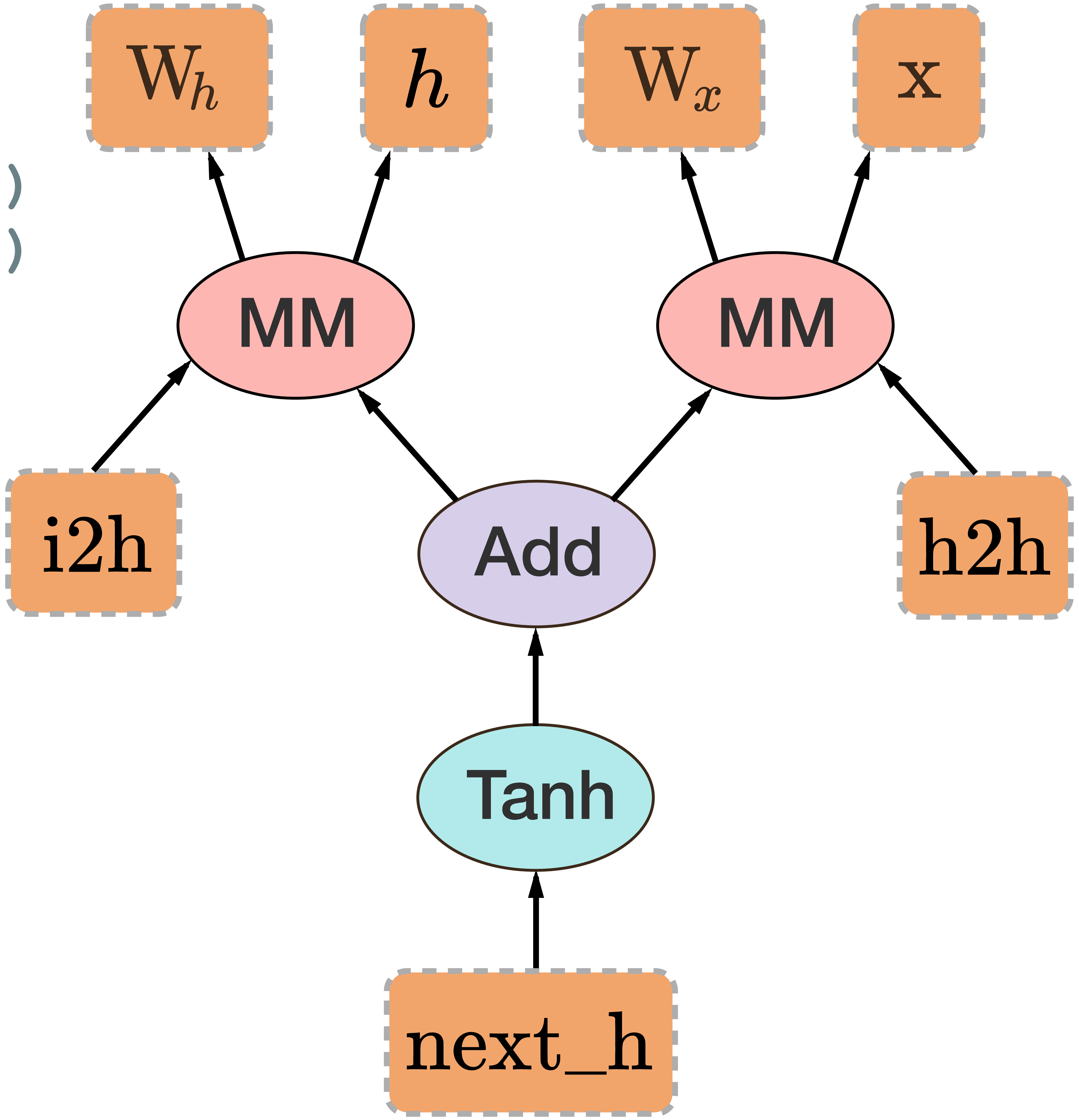
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



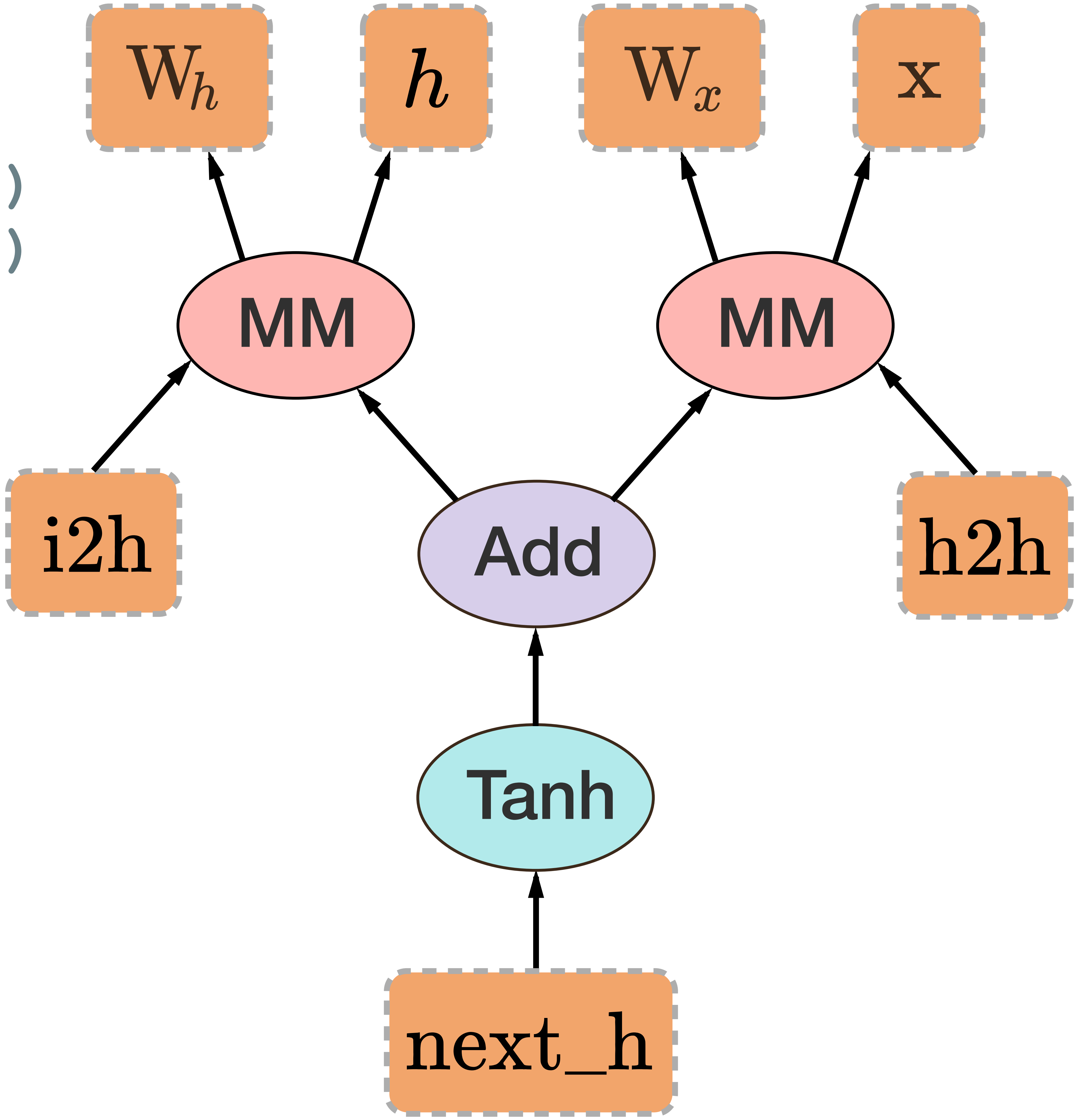
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```



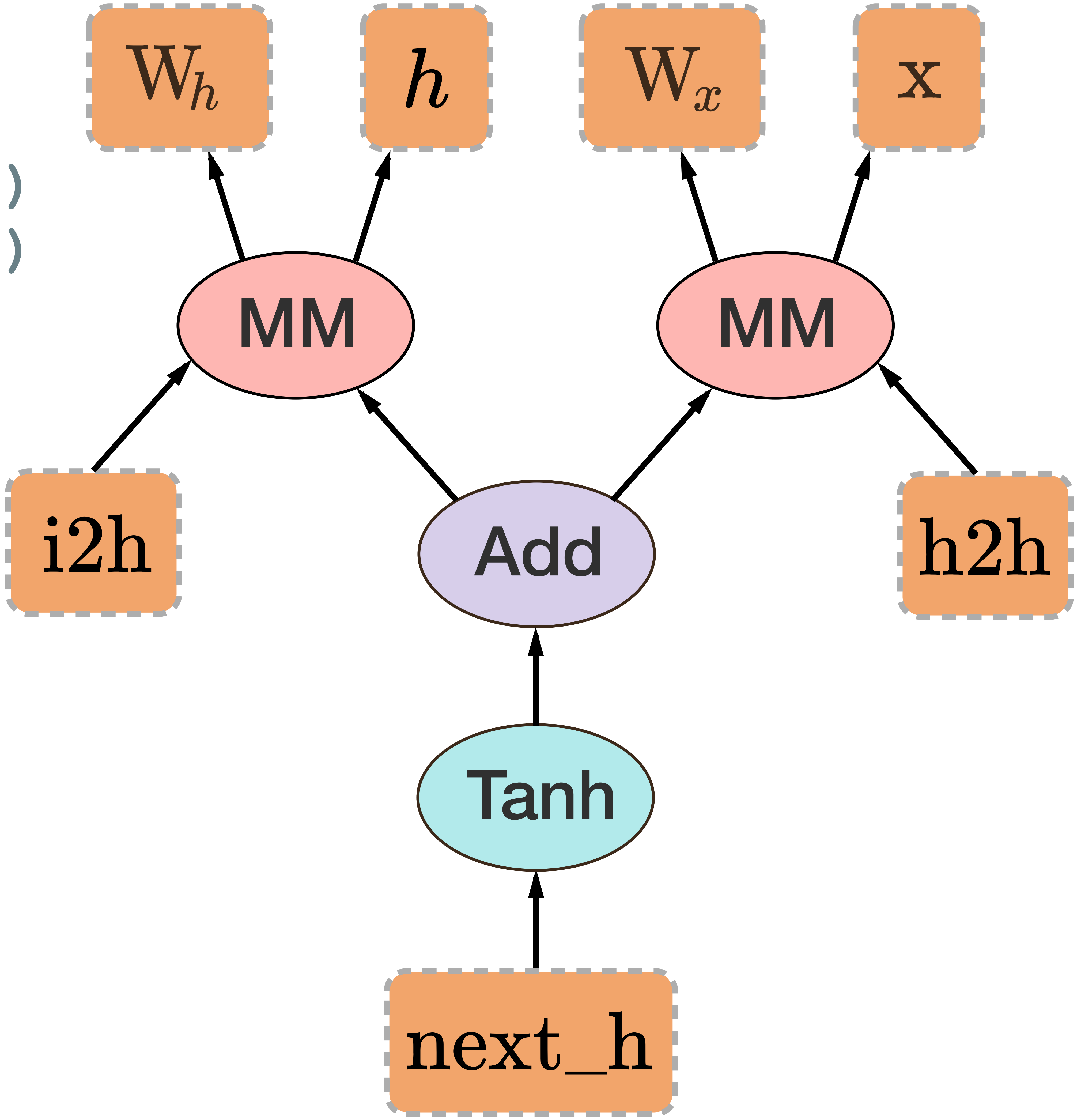
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



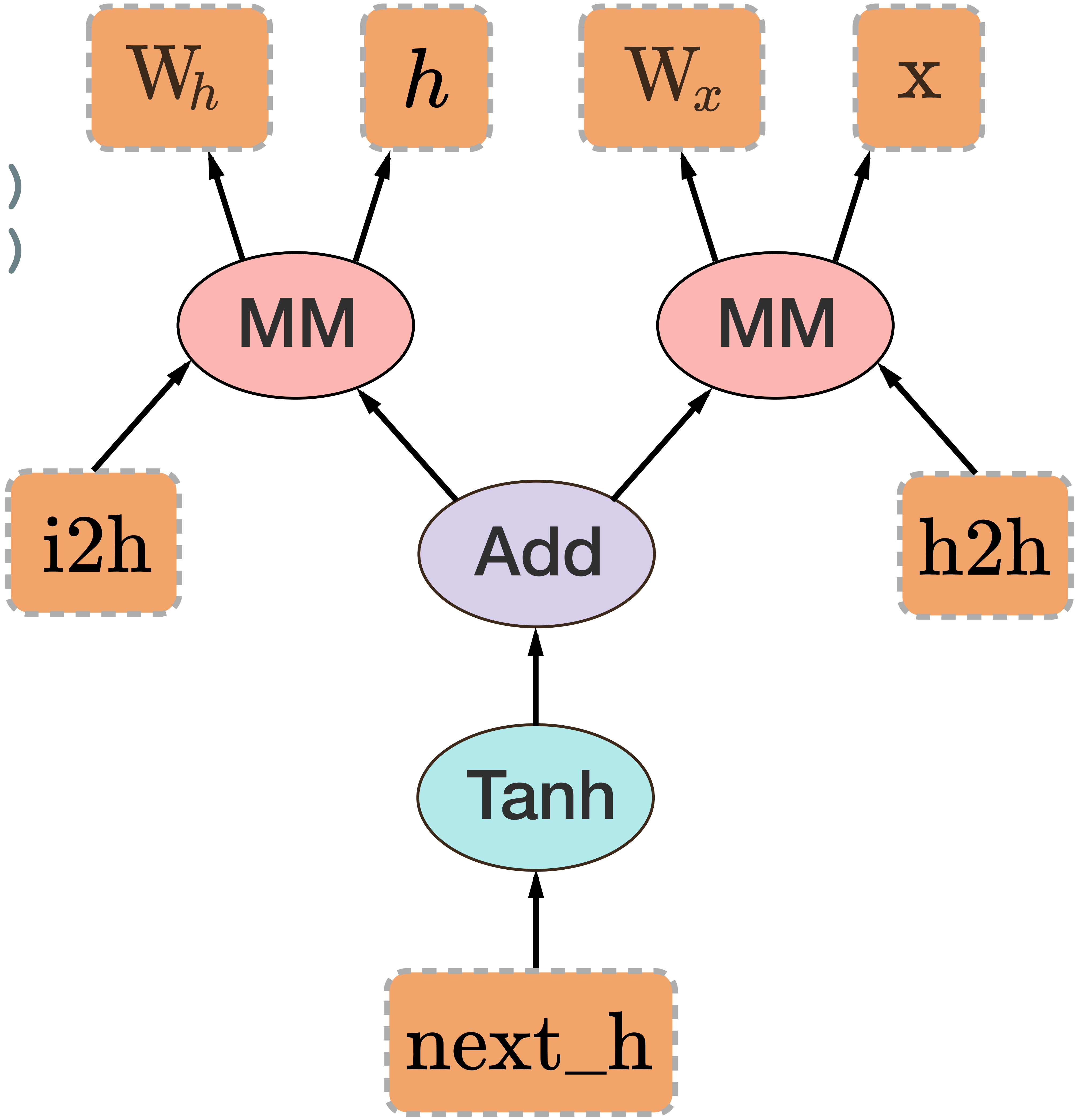
PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20),
create_graph=True, retain_graph=True)

torch.autograd.grad([next_h], [W_h.grad])
```



PyTorch Autograd

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)


i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20),
create_graph=True, retain_graph=True)

torch.autograd.grad([next_h], [W_h.grad])
```



The ability to take n-th order derivatives

Deep Learning

Problem Statement

- Deep Learning Workloads

Problem Statement

- Deep Learning Workloads

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

N samples, each of some shape D

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

mini-batch of M samples (M << N),
each of shape D

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

neural network with weights

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

backpropagation:
compute derivatives wrt loss, using chain rule

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

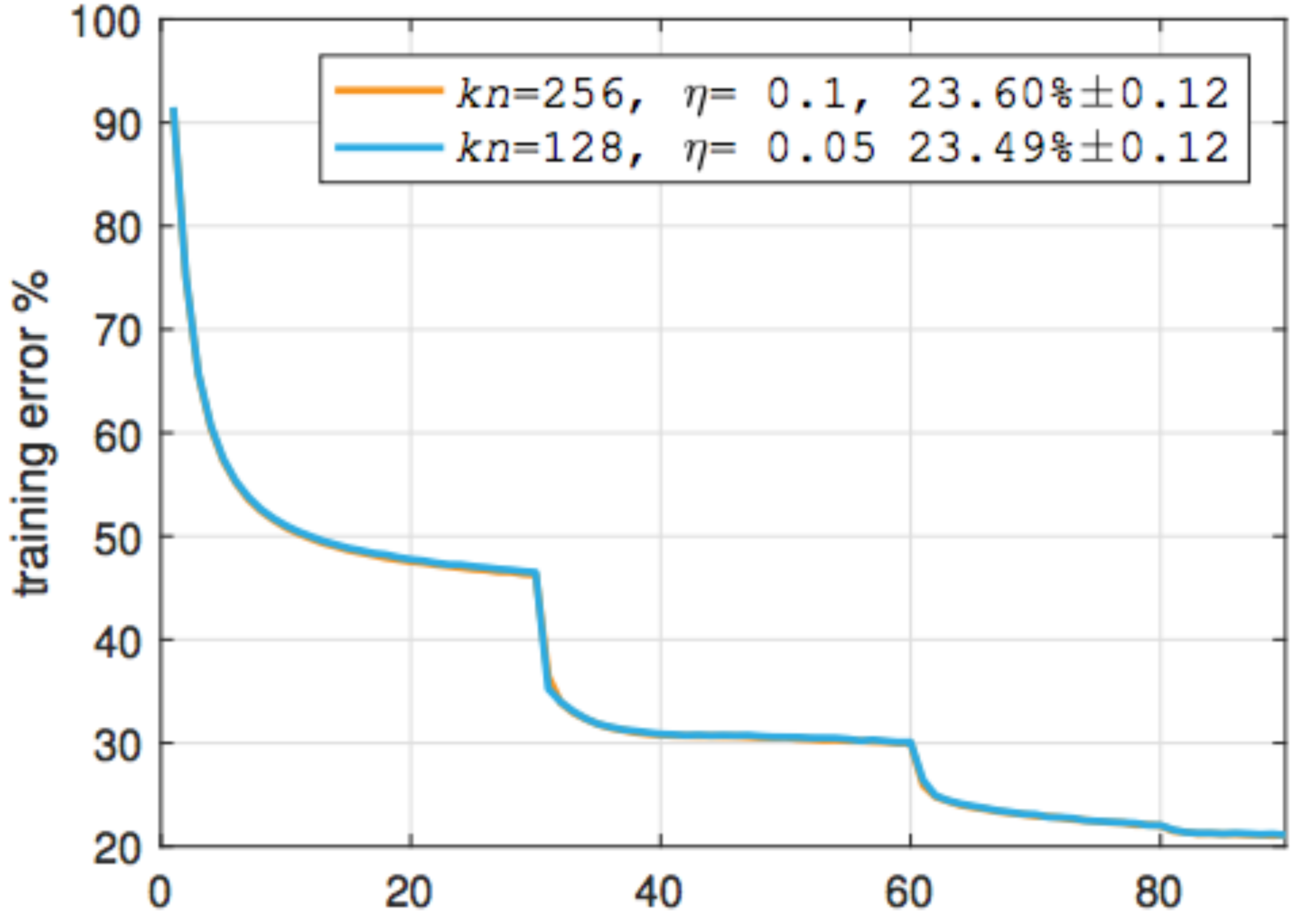
update weights using the computed gradients

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

```
for epoc
    for                                    .ning_data):

                                           jet)
```



Problem Statement

- Deep Learning Workloads

neural network with weights

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Types of typical operators

Convolution

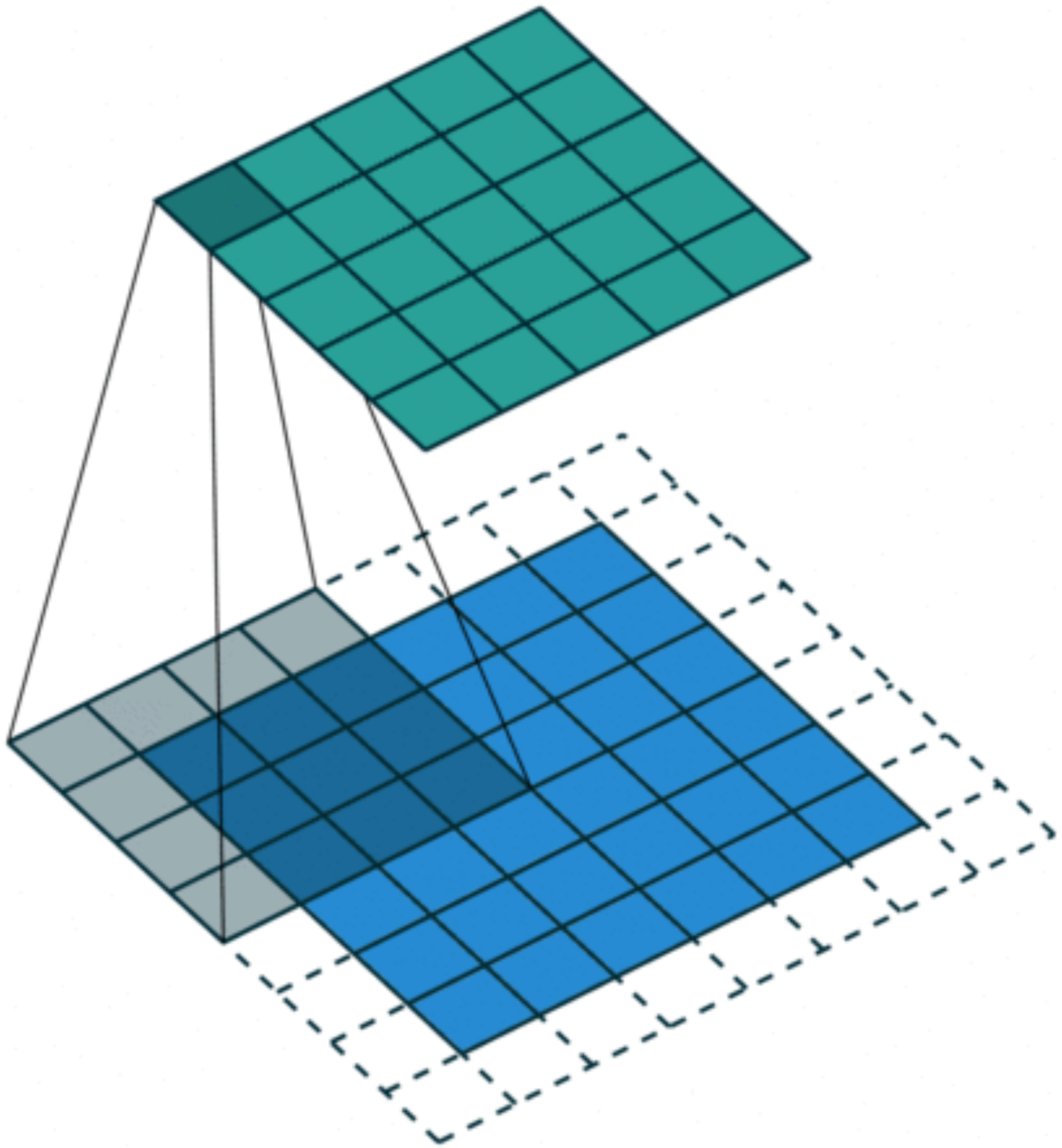
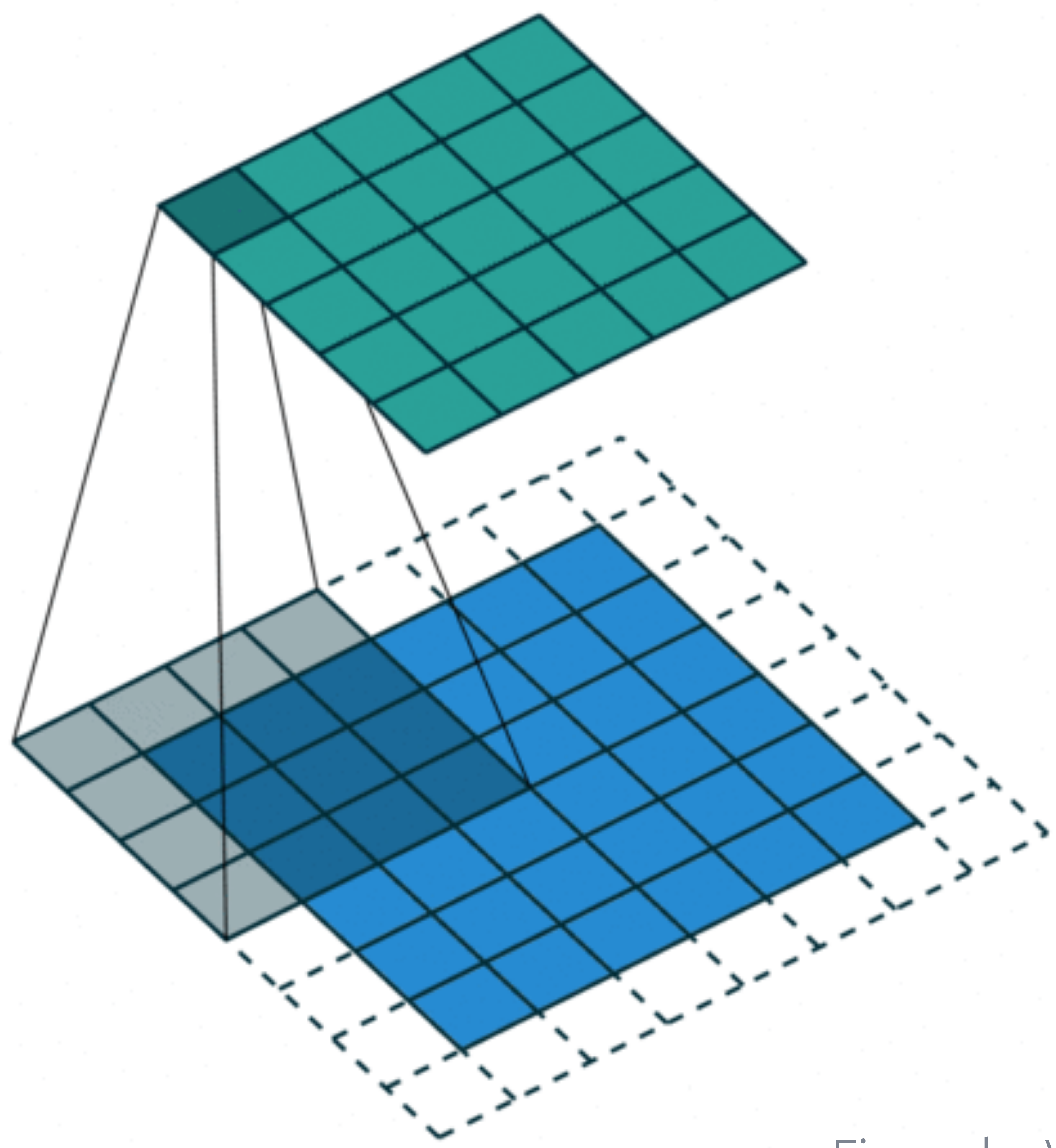


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Convolution



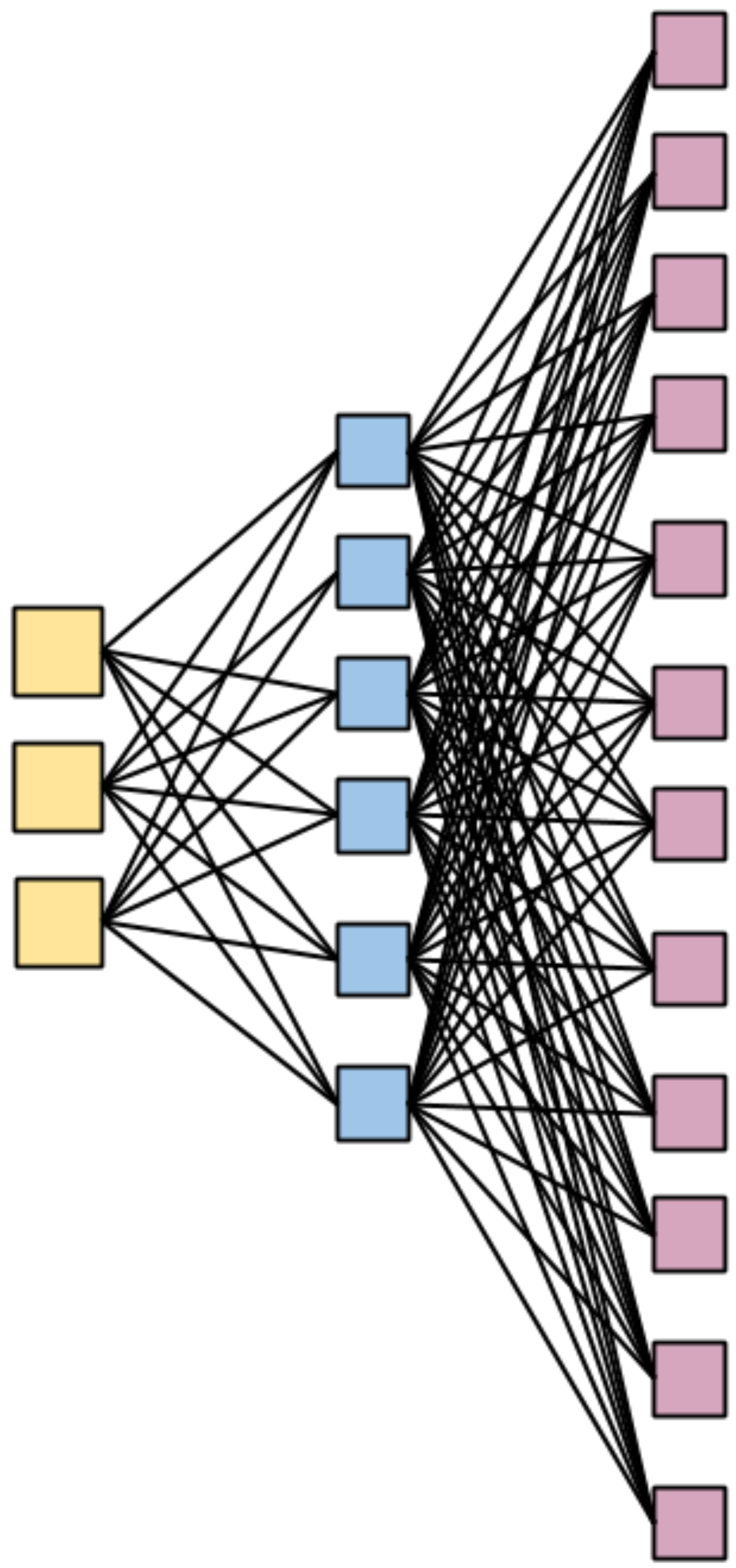
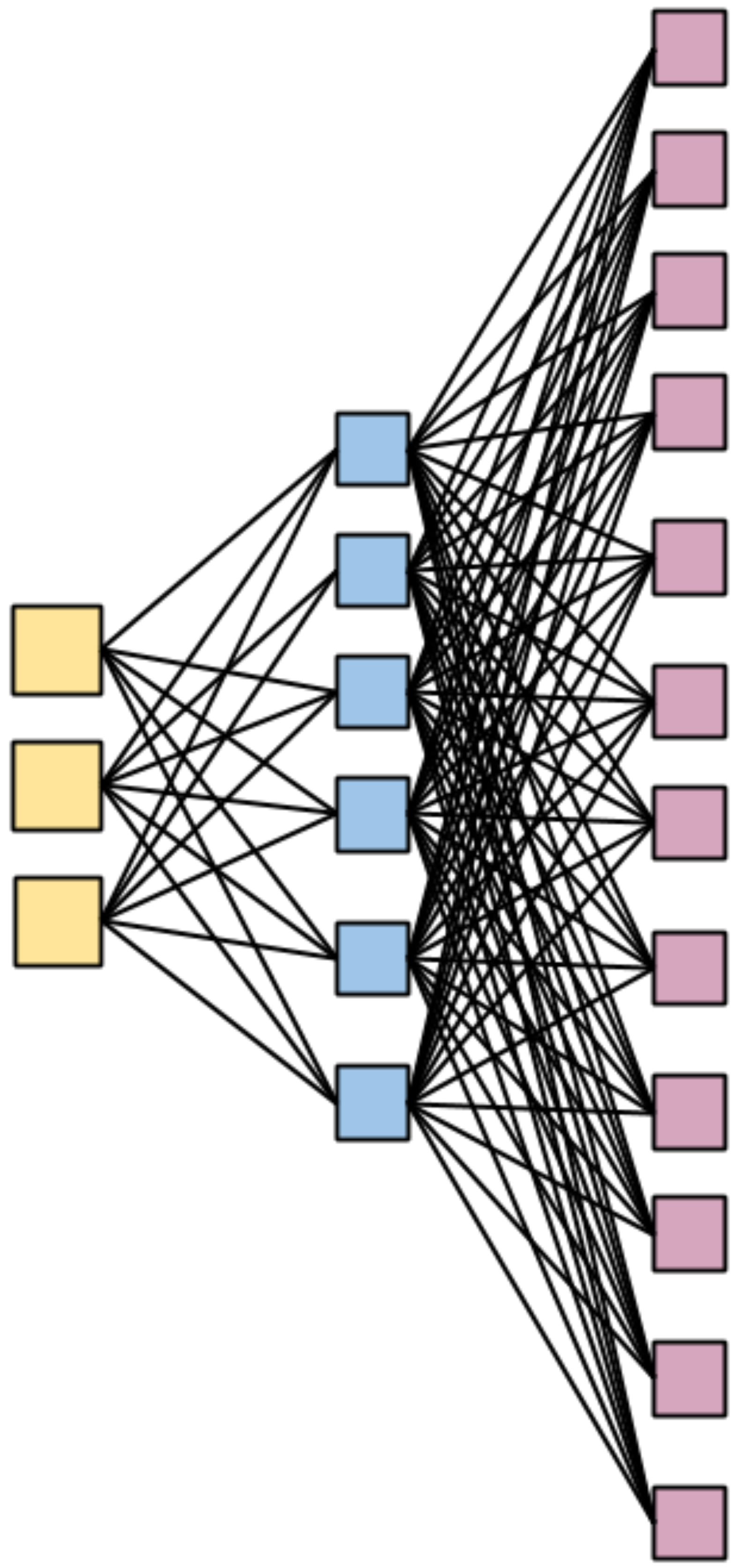


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

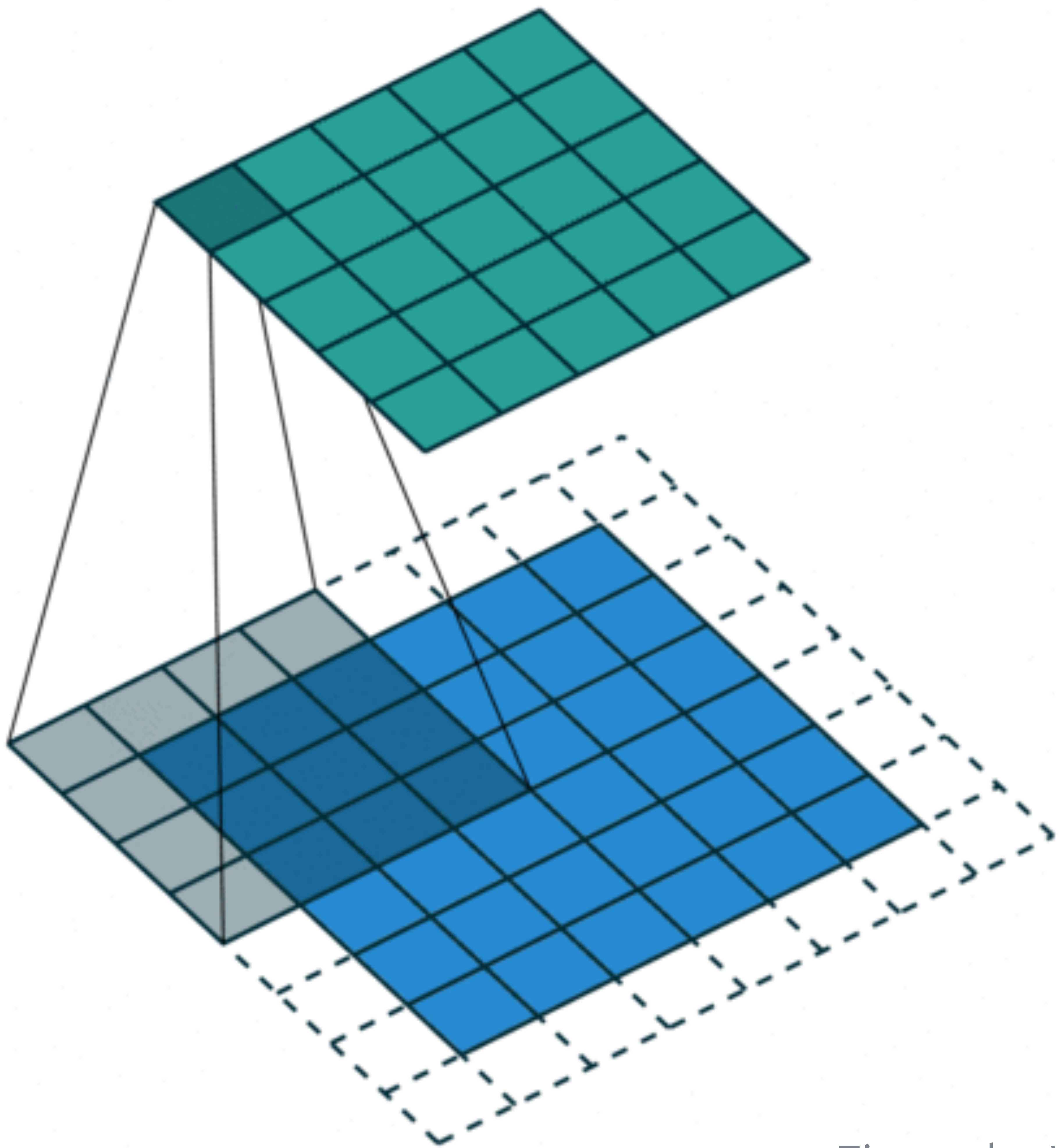
Convolution



```
for oc in output_channel:
    for ic in input_channel:
        for h in output_height:
            for w in output_width:
                for kh in kernel_height:
                   for kw in kernel_width:
                       output_pixel += input_pixel * kernel_value
```

Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators



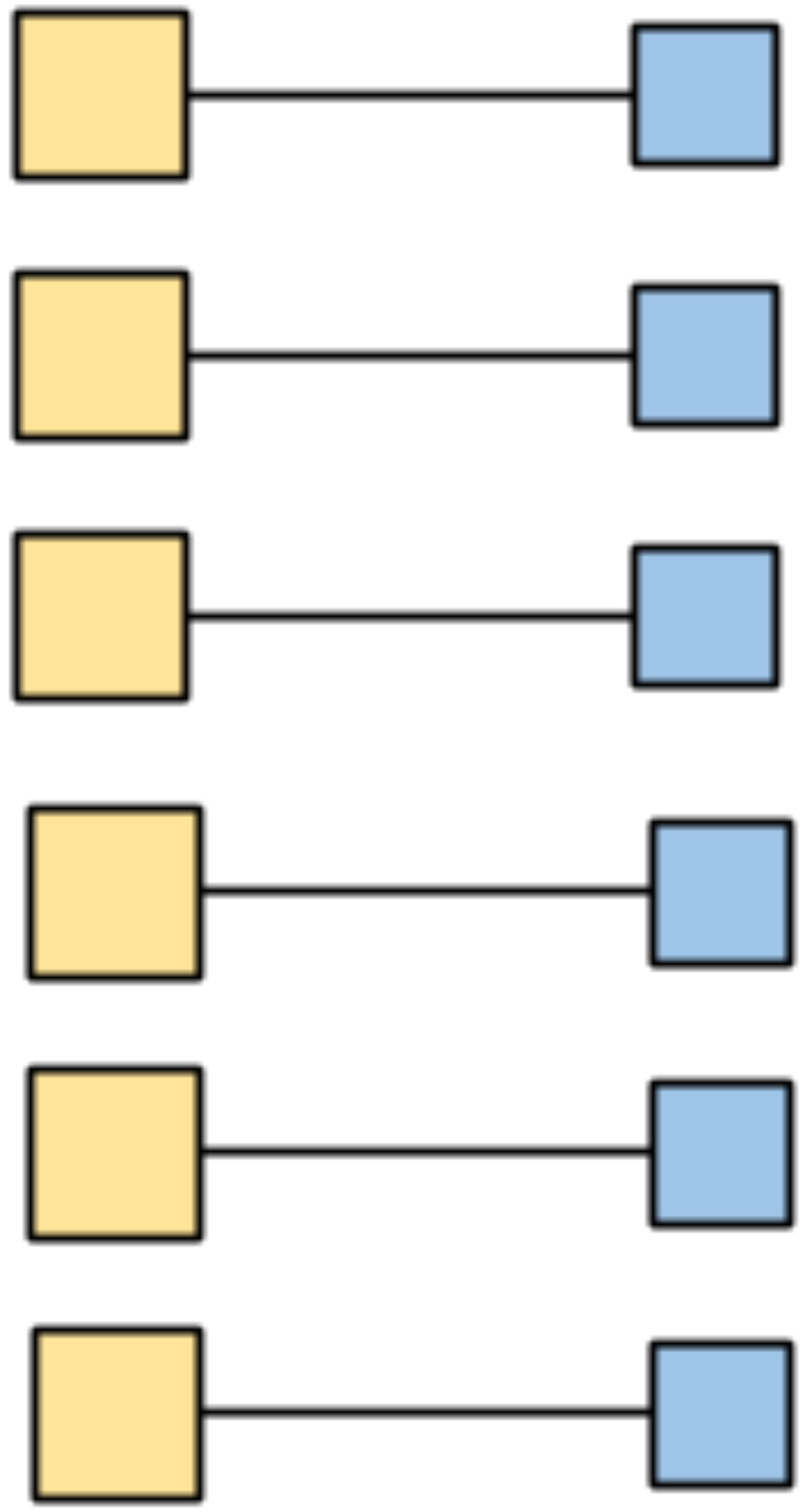


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Matrix Multiply

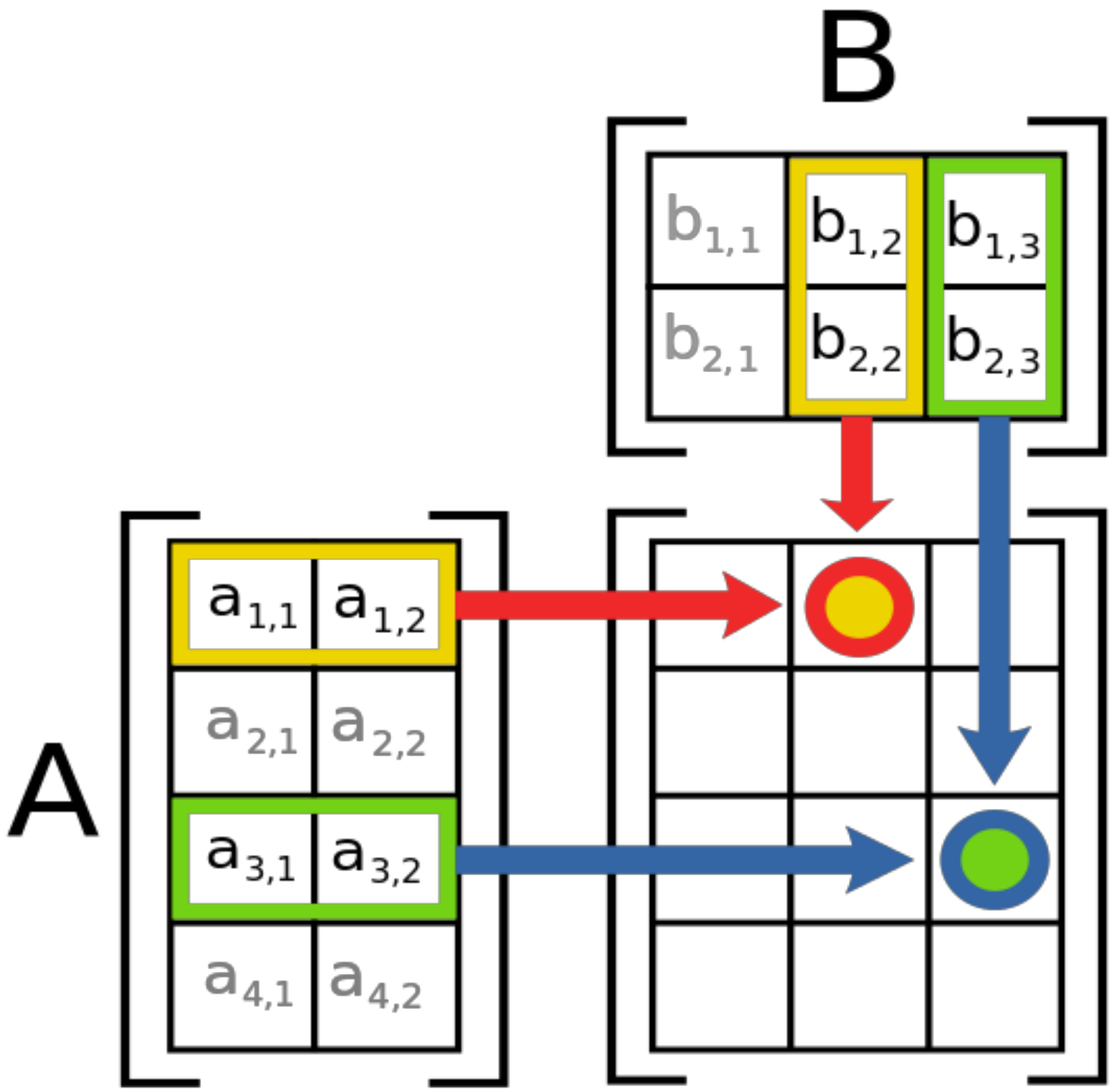


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Types of typical operators

Pointwise operations

```
for (i=0; i < data_length; i++) {
    output[i] = input1[i] + input2[i]
}
```

Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Types of typical operators

Reduction operations

```
double sum = 0.0;
for (i=0; i < data_length; i++) {
    sum += input[i];
}
```

Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Chained Together

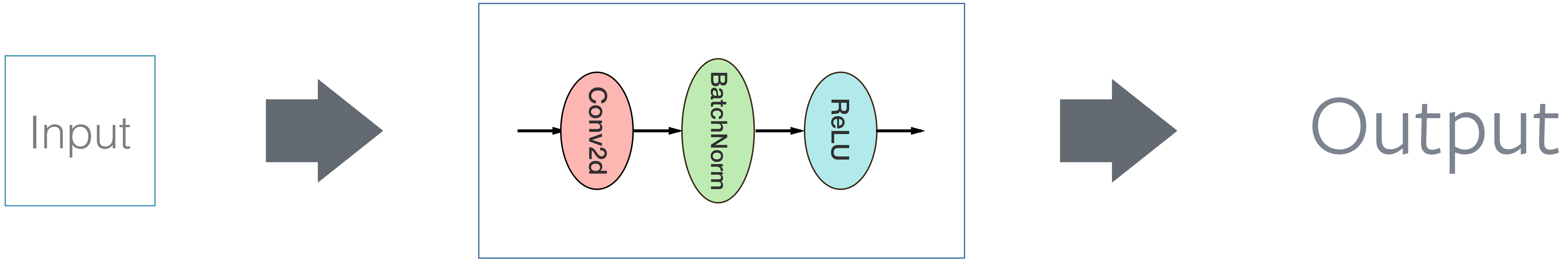


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Chained Together

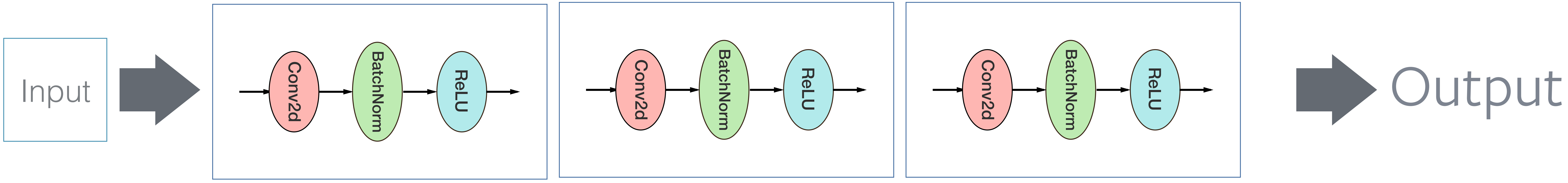


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Chained Together

"deep"

Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Chained Together

"deep"

recurrent

Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Trained with Gradient Descent

"deep"

recurrent

Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Problem Statement

- Deep Learning Workloads

an easy way to see recurrence

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads

an easy way to see recurrence

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output, hidden = [], zeros()
        for t in data.size(0):
            out, hidden = model(data[t], hidden)
            output.append(out)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Problem Statement

- Deep Learning Workloads
  - Vision models
    - model is very deep, straight-line chain with no recurrence
    - lots of convolutions
    - typically run on GPUs

Problem Statement

- Deep Learning Workloads
  - Vision models
    - model is very deep, straight-line chain with no recurrence
    - lots of convolutions
    - typically run on GPUs
  - NLP models
    - LSTM-RNN
      - model is 1 to 4 "layers" deep
      - two matmuls across space and time along with pointwise ops
      - typically run on CPUs if small, GPUs if large

Deep Learning Frameworks

- Make this easy to program

```
for epoch in range(max_epochs):
    for data, target in enumerate(training_data):
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

Neural Networks

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

Neural Networks

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

Neural Networks

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
input = Variable(torch.randn(10, 20))
output = model(input)
```

Optimization package

SGD, Adagrad, RMSProp, LBFGS, etc.

```
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01, momentum=0.9)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = F.cross_entropy(output, target)
    loss.backward()
    optimizer.step()
```

Deep Learning & Python

- Most deep learning frameworks in Python
- Global interpreter-lock
- application logic is order of magnitude slower than C++

Deep Learning & Python

- Most deep learning frameworks in Python
- Global interpreter-lock
- application logic is order of magnitude slower than C++
- most frameworks implemented in C++, with bindings to Python

Deep Learning & Hardware

- Typically support CPU & GPU

Deep Learning & Hardware

- Typically support CPU & GPU
- More recently: TPU, xPU etc.

Deep Learning & Compilers

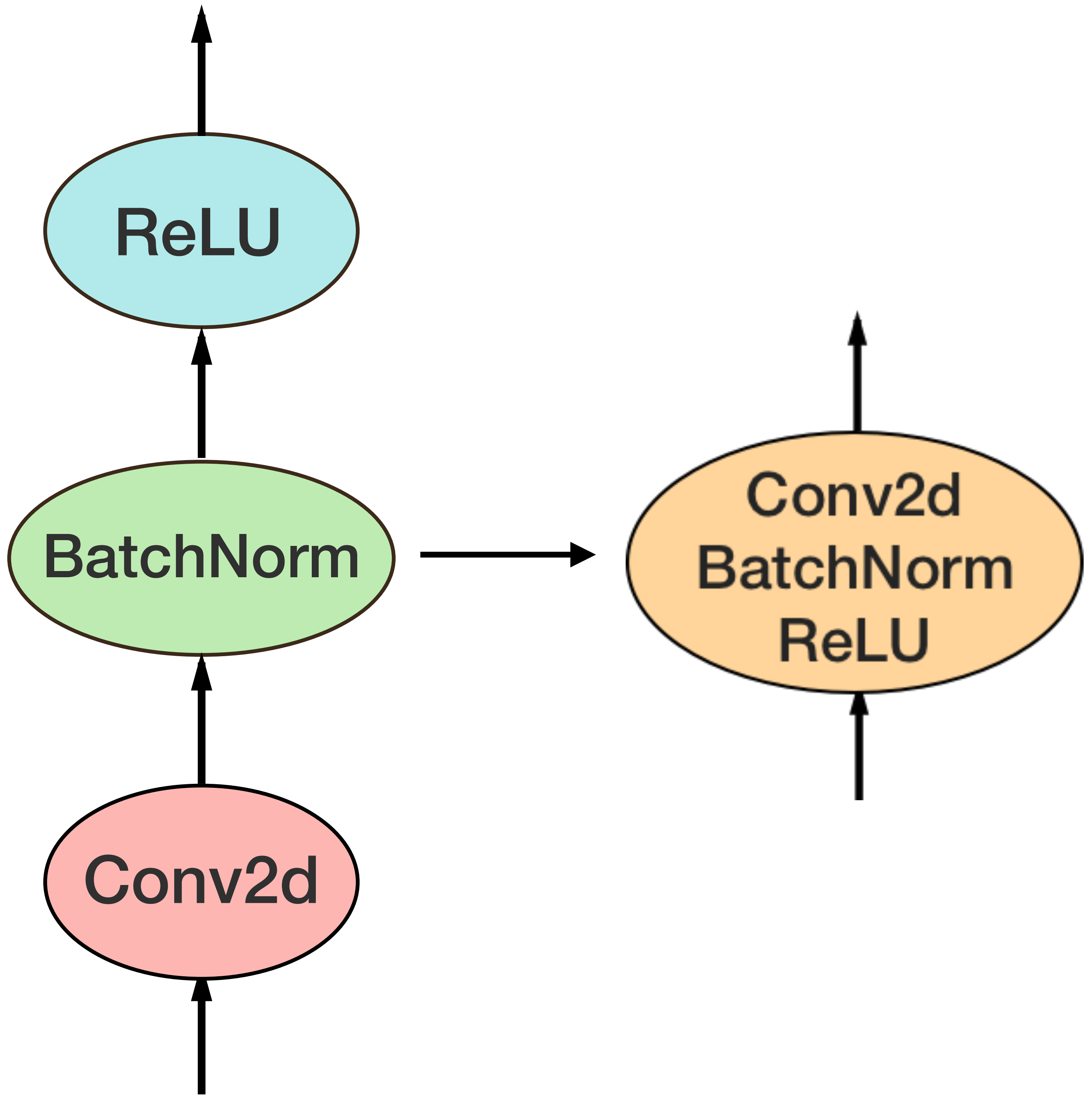
- Most modern frameworks support compilation

Deep Learning & Compilers

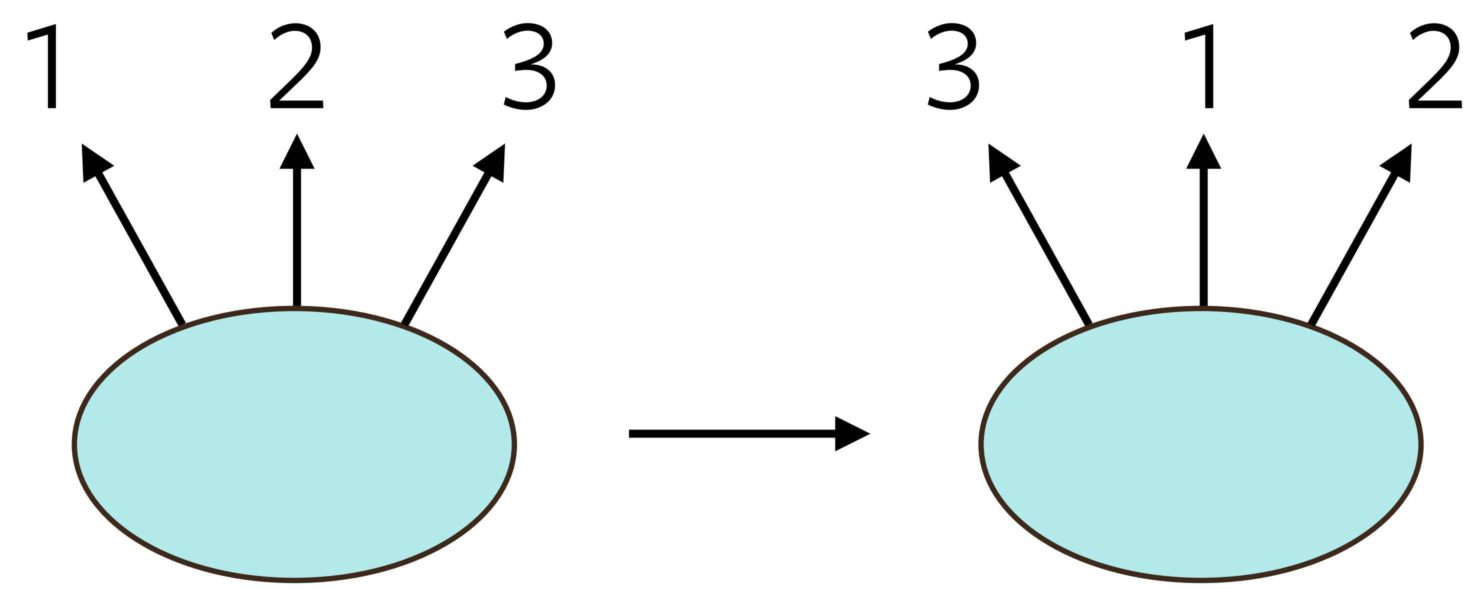
- Most modern frameworks support compilation
- Runtime-retargeting / code generation

Compilation benefits

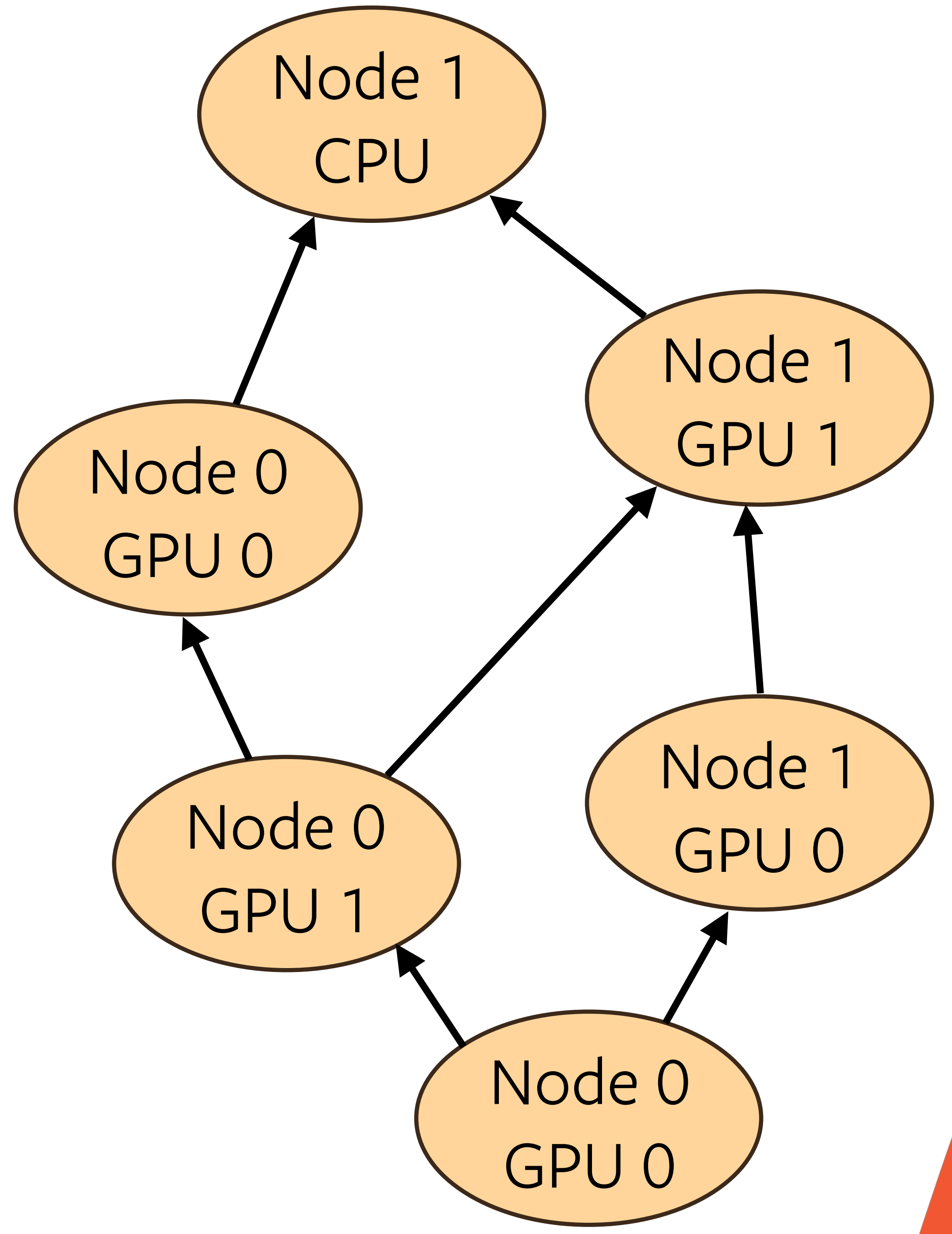
Kernel fusion



Out-of-order execution



Automatic work placement



Deep Learning & Distributed

- Most modern frameworks support distributed training

Deep Learning & Distributed

- Most modern frameworks support distributed training
- Parallelize over batches (data-parallel) and models (model-parallel)

Deep Learning & Distributed

- Most modern frameworks support distributed training
- Parallelize over batches (data-parallel) and models (model-parallel)
- PyTorch's distributed built on top of an MPI-like stack

TURN KEY SOLUTION

torch.nn.DistributedDataParallel

JUST WRAP YOUR MODEL

```
torch.distributed.init_process_group(world_size=4, init_method='...’)
model = torch.nn.DistributedDataParallel(model)

for epoch in range(max_epochs):
   for data, target in enumerate(training_data):
     output = model(data)
     loss = F.nll_loss(output, target)
     loss.backward()
     optimizer.step()
```

DISTRIBUTED DATA PARALLEL

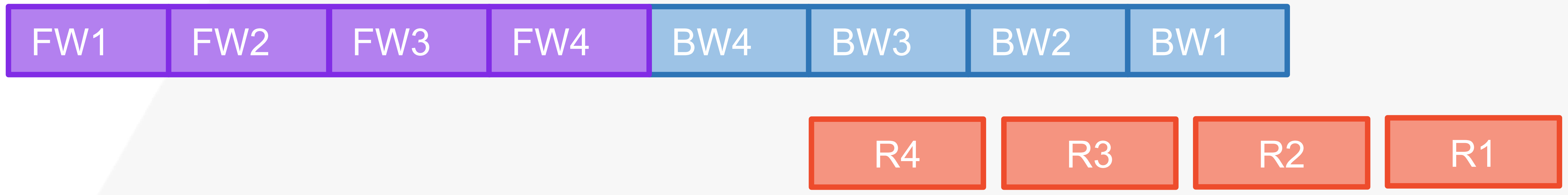
Performance-driven design

- Overlapping BWs with all-reductions
- Coalescing small tensors into buckets
  - A bucket is a big coalesced tensor

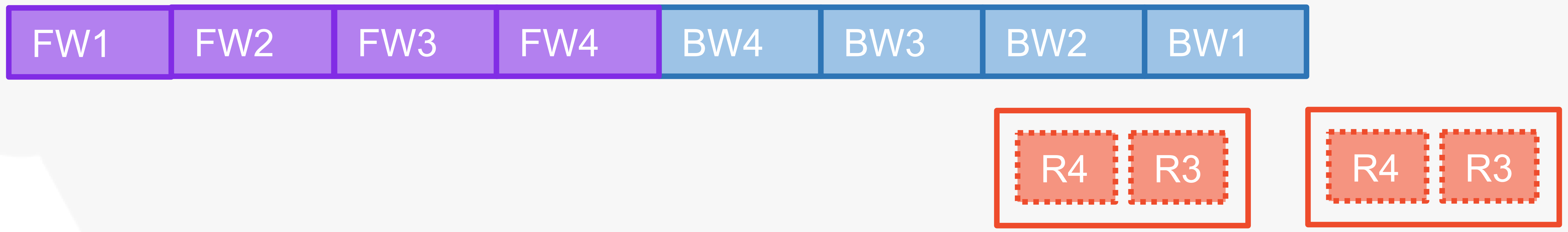
NO OVERLAPPING

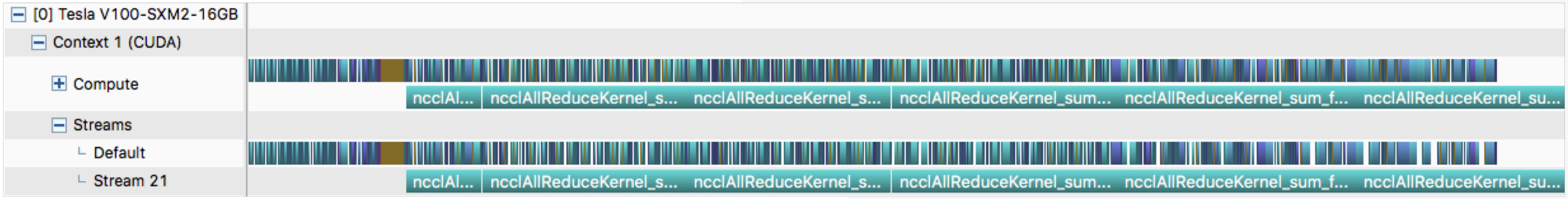
An iteration: Forward (FW) -> Backward(BW) -> AllReduce(R)

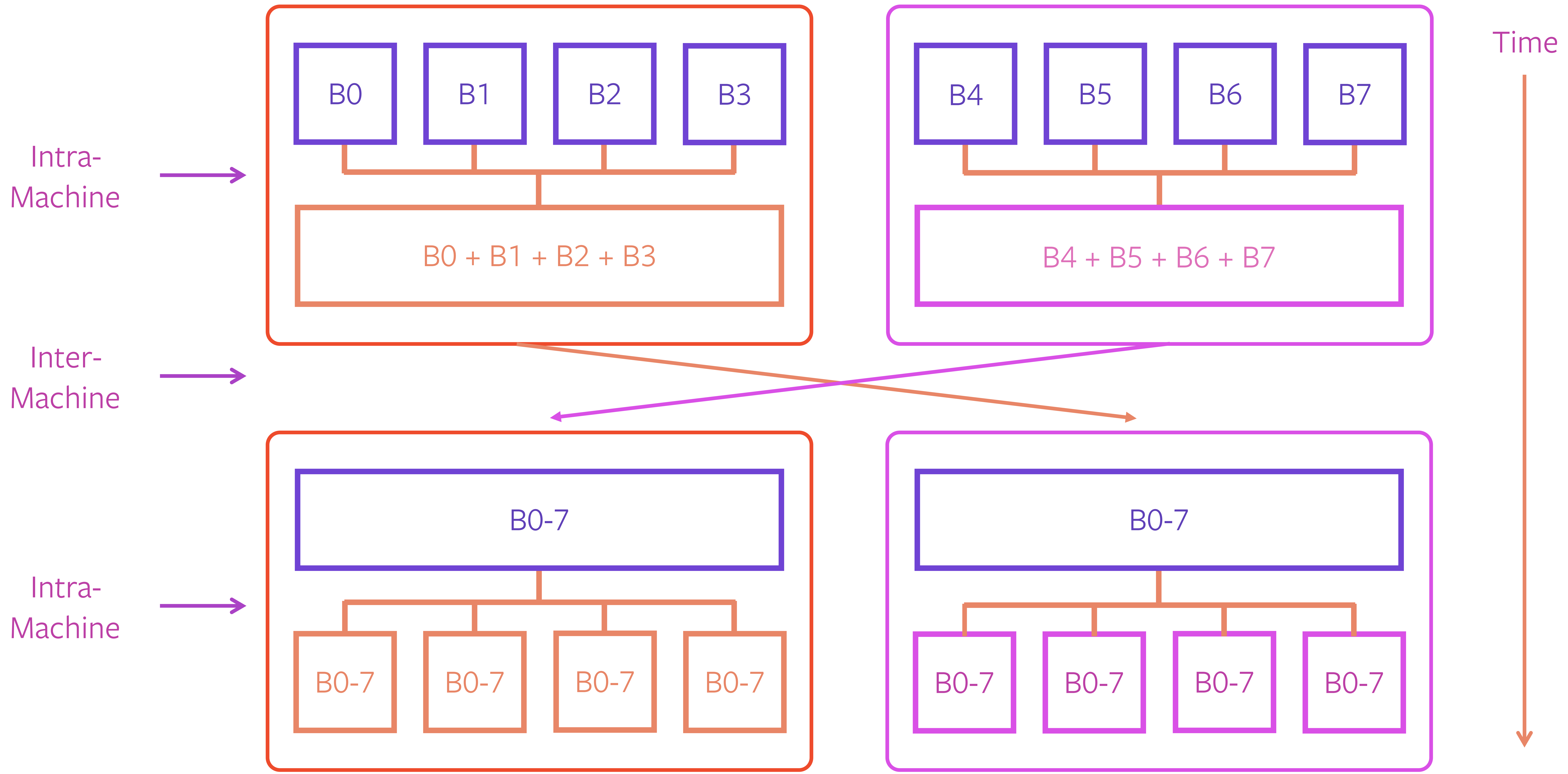
OVERLAPPING BACKWARD WITH REDUCE



TENSOR COALESCING/ BUCKETING



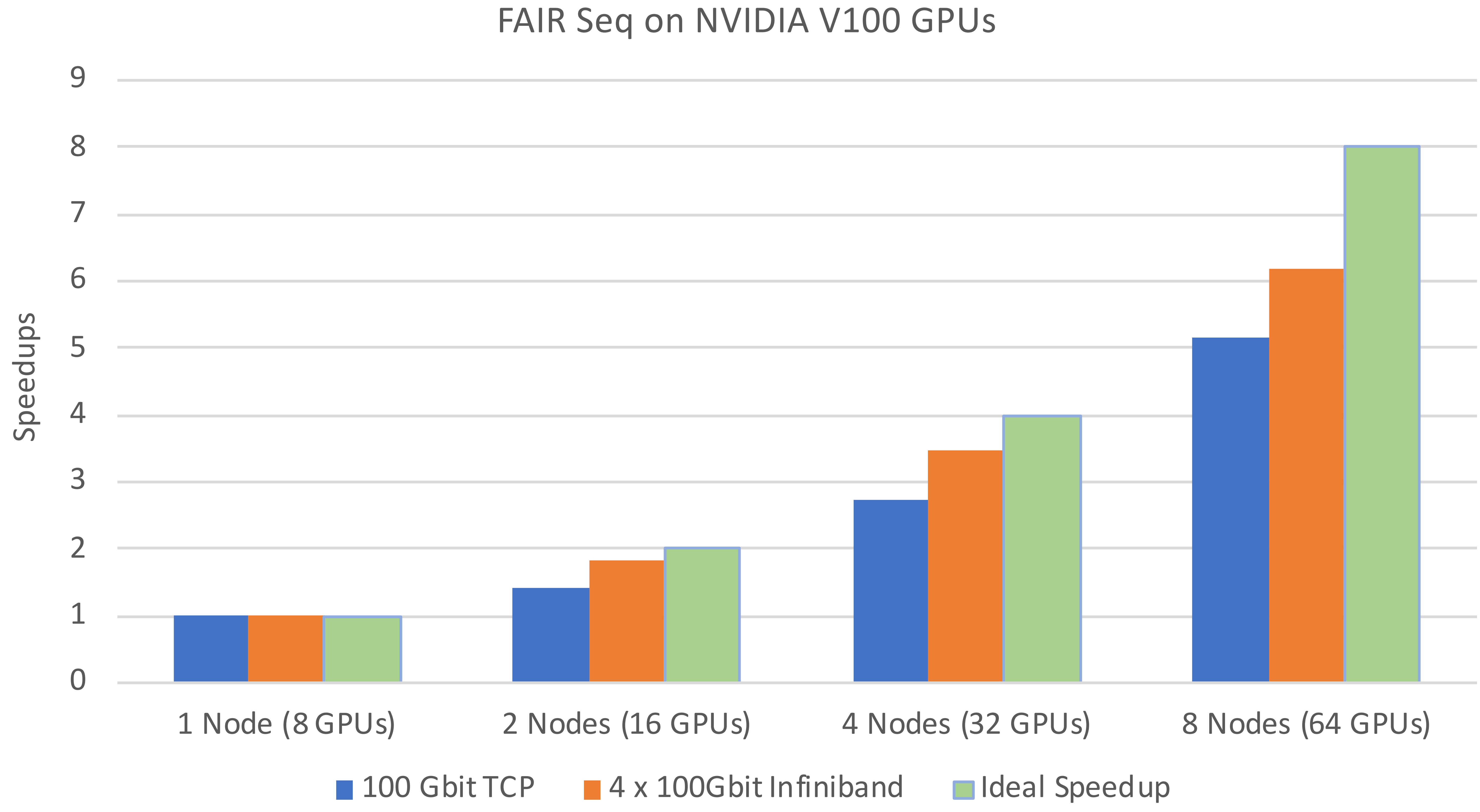


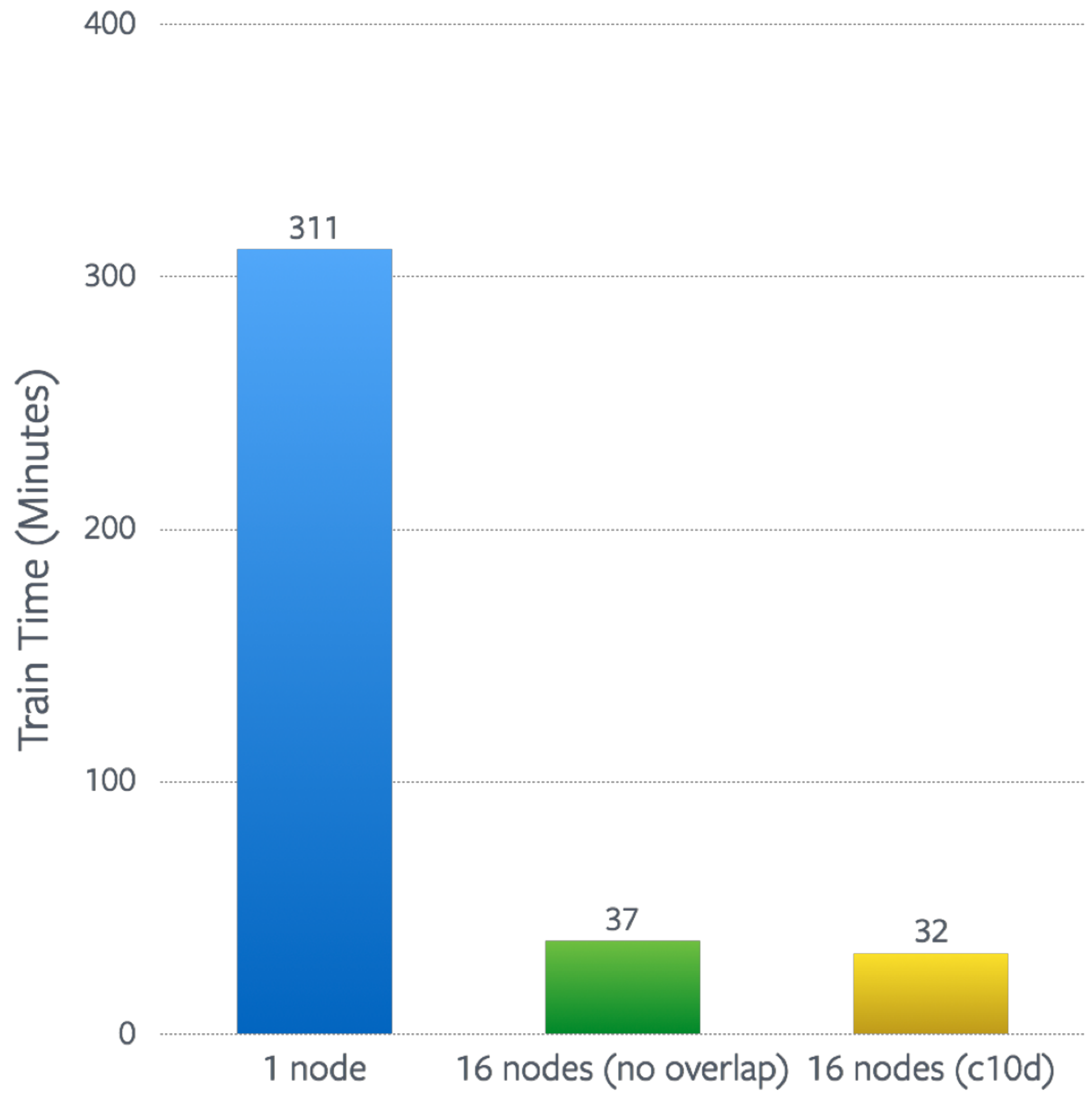


Intra-Machine
Inter-Machine
Intra-Machine
B0
B1
B2
B3
B4
B5
B6
B7
B0 + B1 + B2 + B3
B4 + B5 + B6 + B7
B0-7
B0-7
B0-7
B0-7
B0-7
B0-7
B0-7
B0-7
B0-7
B0-7
Time

Distributed Training Performance – FAIR Seq

Bonjour à tous ! → Hello everybody!





- 311 minutes – 32 minutes, by going from 1 to 16 NVIDIA DGX-1 nodes (8 to 128 NVIDIA V100 GPUs)
- 19% performance gain (1.53M – 1.82M Words Per Second on 16 nodes), thanks to c10d DDP overlapping

C++
FRONTEND

The aesthetics of imperative PyTorch for high performance, pure C++ research environments

MISSION

The aesthetics of PyTorch in pure C++

MOTIVATION

Enable research in environments that are . . .

MISSION

The aesthetics of PyTorch in pure C++

VALUES

Enable research in environments that are . . .

LOW LATENCY

BARE METAL

MULTITHREADED

ALREADY C++

torch::nn

NEURAL NETWORKS

torch::optim

OPTIMIZERS

torch::data

DATASETS &
DATA LOADERS

torch::serialize

SERIALIZATION

torch::python

PYTHON INTER-OP

torch::jit

TORCH SCRIPT
INTER-OP

```cpp
#include <torch/torch.h>

struct Net : torch::nn::Module {
  Net() : fc1(8, 64), fc2(64, 1) {
    register_module("fc1", fc1);
    register_module("fc2", fc2);
  }

  torch::Tensor forward(torch::Tensor x) {
    x = torch::relu(fc1->forward(x));
    x = torch::dropout(x, /*p=*/0.5);
    x = torch::sigmoid(fc2->forward(x));
    return x;
  }

  torch::nn::Linear fc1, fc2;
};
```

C++

```python
import torch

class Net(torch.nn.Module):
    def __init__(self):
        self.fc1 = torch.nn.Linear(8, 64)
        self.fc2 = torch.nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1.forward(x))
        x = torch.dropout(x, p=0.5)
        x = torch.sigmoid(self.fc2.forward(x))
        return x
```

PYTHON

```cpp
Net net;

auto data_loader = torch::data::data_loader(
  torch::data::datasets::MNIST("./data"));

torch::optim::SGD optimizer(net->parameters());

for (size_t epoch = 1; epoch <= 10; ++epoch) {
  for (auto batch : data_loader) {
    optimizer.zero_grad();
    auto prediction = net->forward(batch.data);
    auto loss = torch::nll_loss(prediction,
                                batch.label);
    loss.backward();
    optimizer.step();
  }
  if (epoch % 2 == 0)
    torch::save(net, "net.pt");
}
```

C++

```python
net = Net()

data_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data'))

optimizer = torch.optim.SGD(net.parameters())

for epoch in range(1, 11):
    for data, target in data_loader:
        optimizer.zero_grad()
        prediction = net.forward(data)
        loss = F.nll_loss(prediction, target)
        loss.backward()
        optimizer.step()
    if epoch % 2 == 0:
        torch.save(net, "net.pt")
```

PYTHON

Thank You