PyTorch

PyTorch PyTorch PyTorch PyTorch PyTorch PyTorch

Usable while performant: the challenges building PyTorch

Soumith Chintala

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

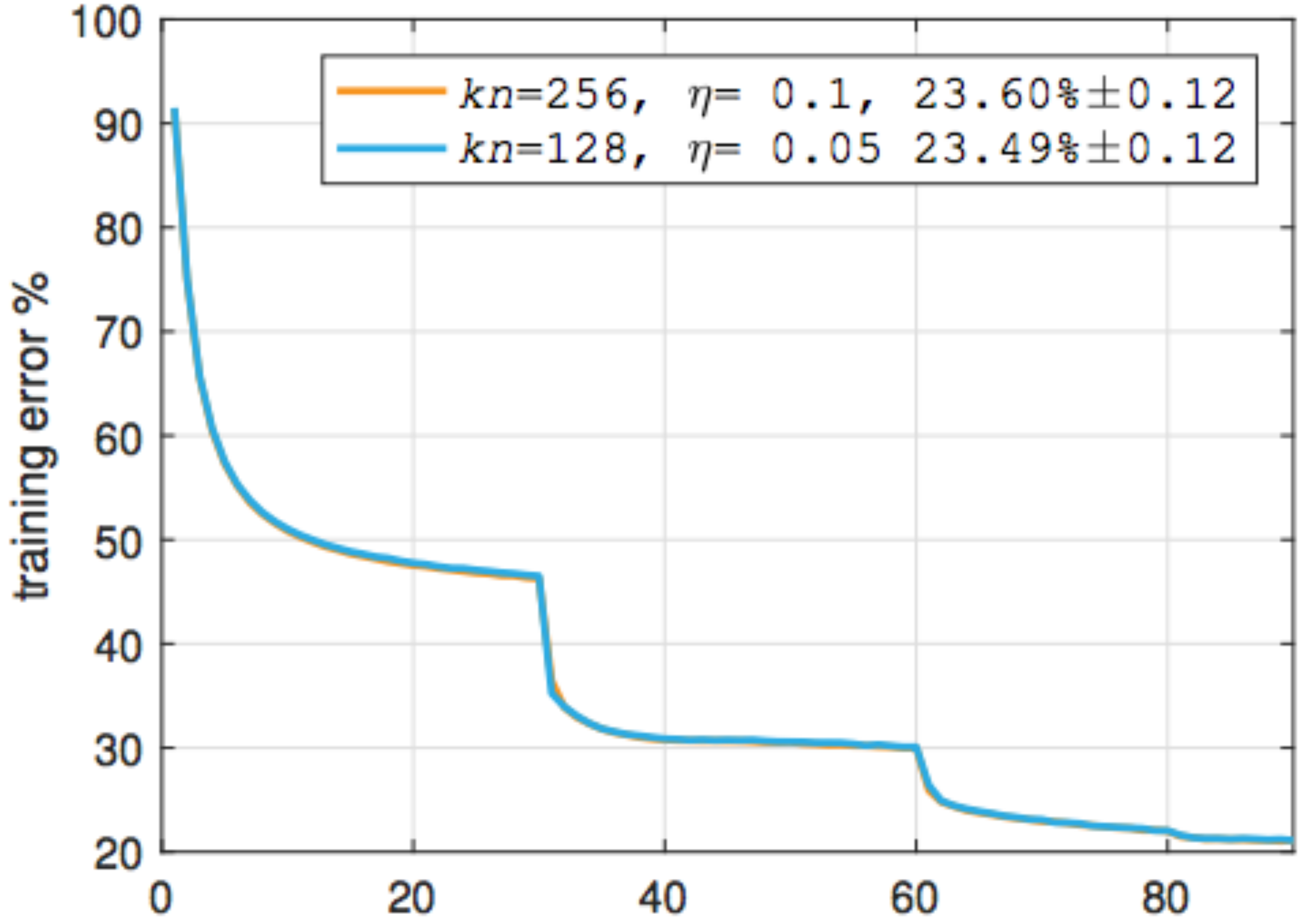
update weights using the computed gradients

```
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):

                                               get)
```



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

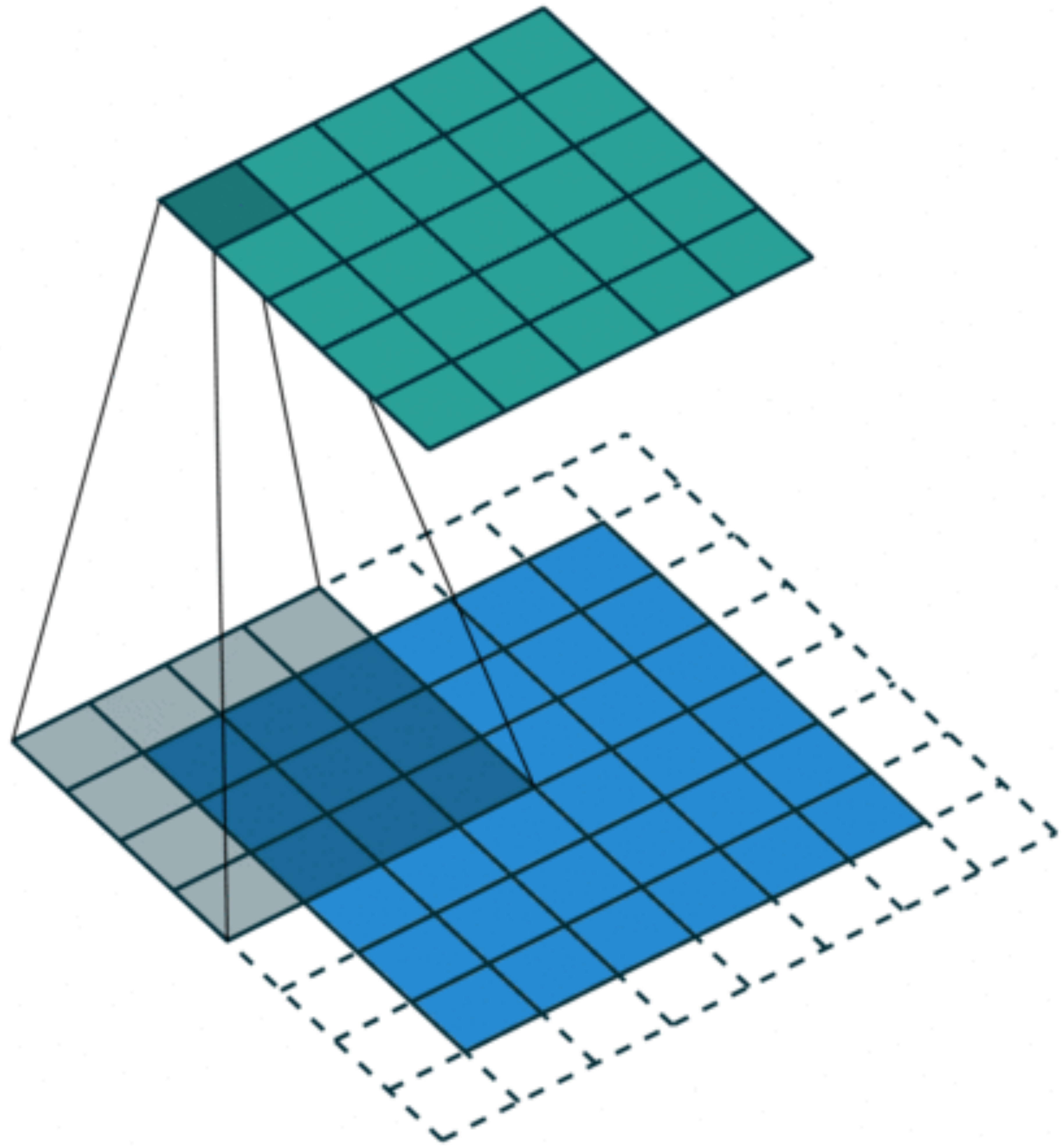
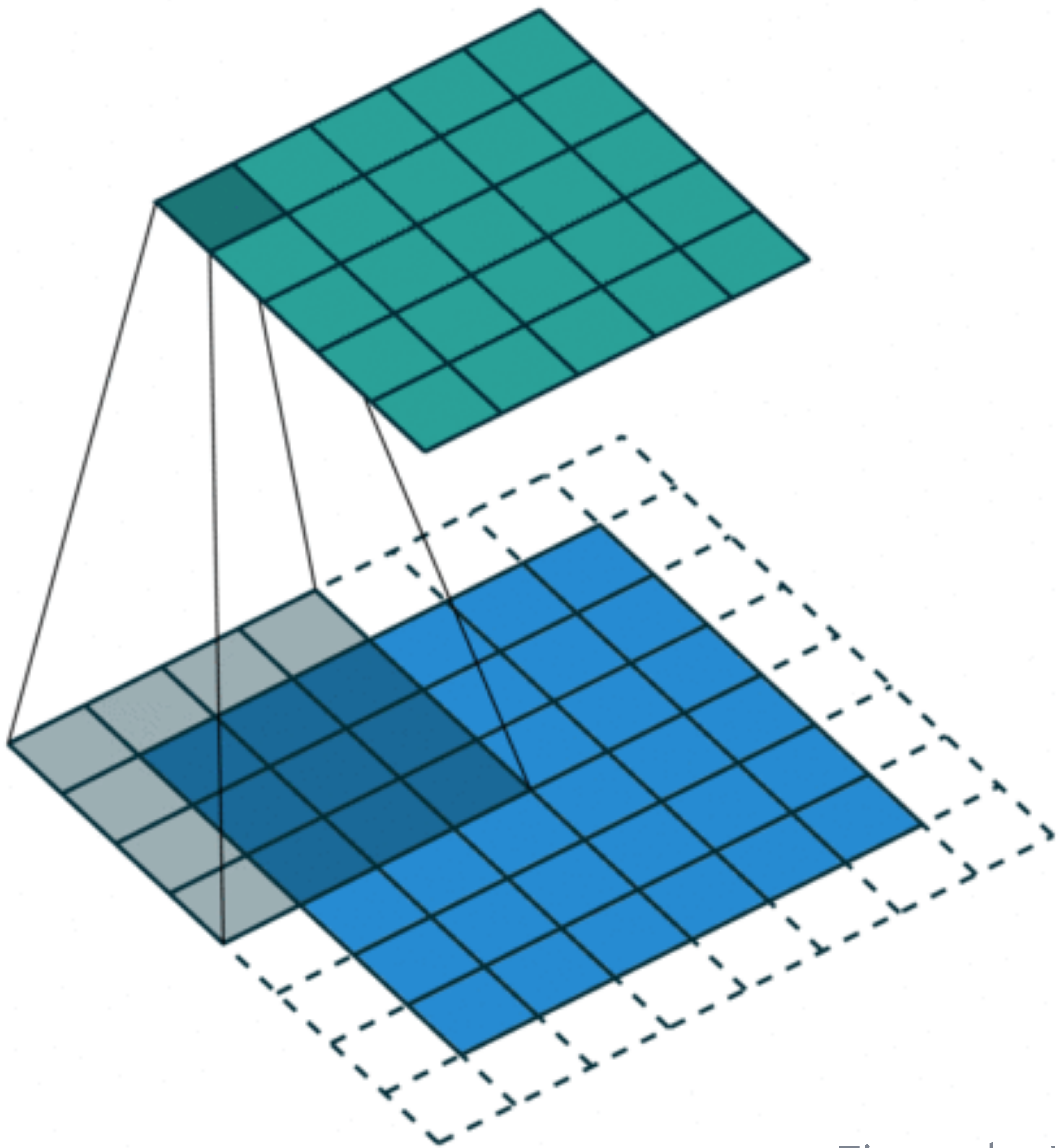


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Convolution



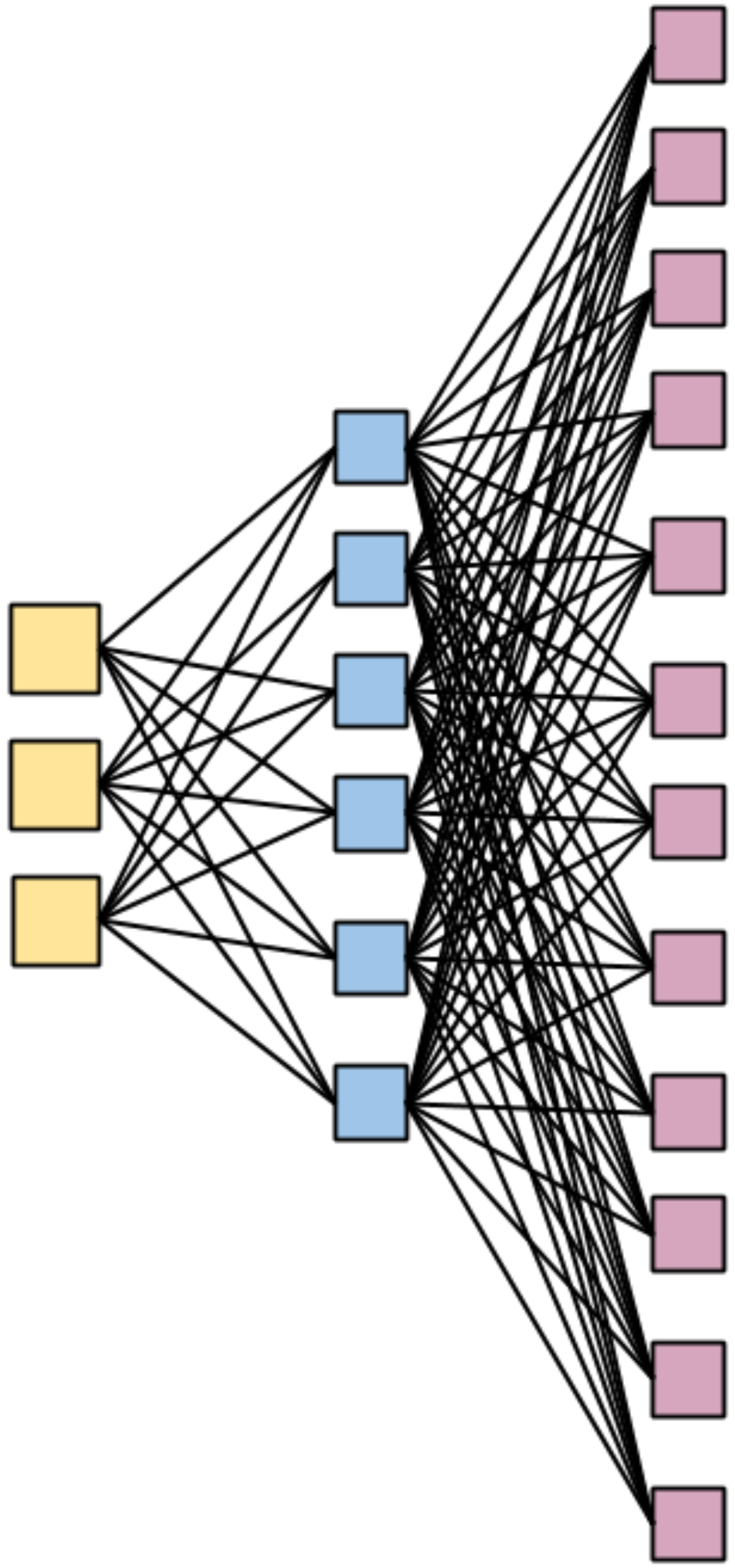
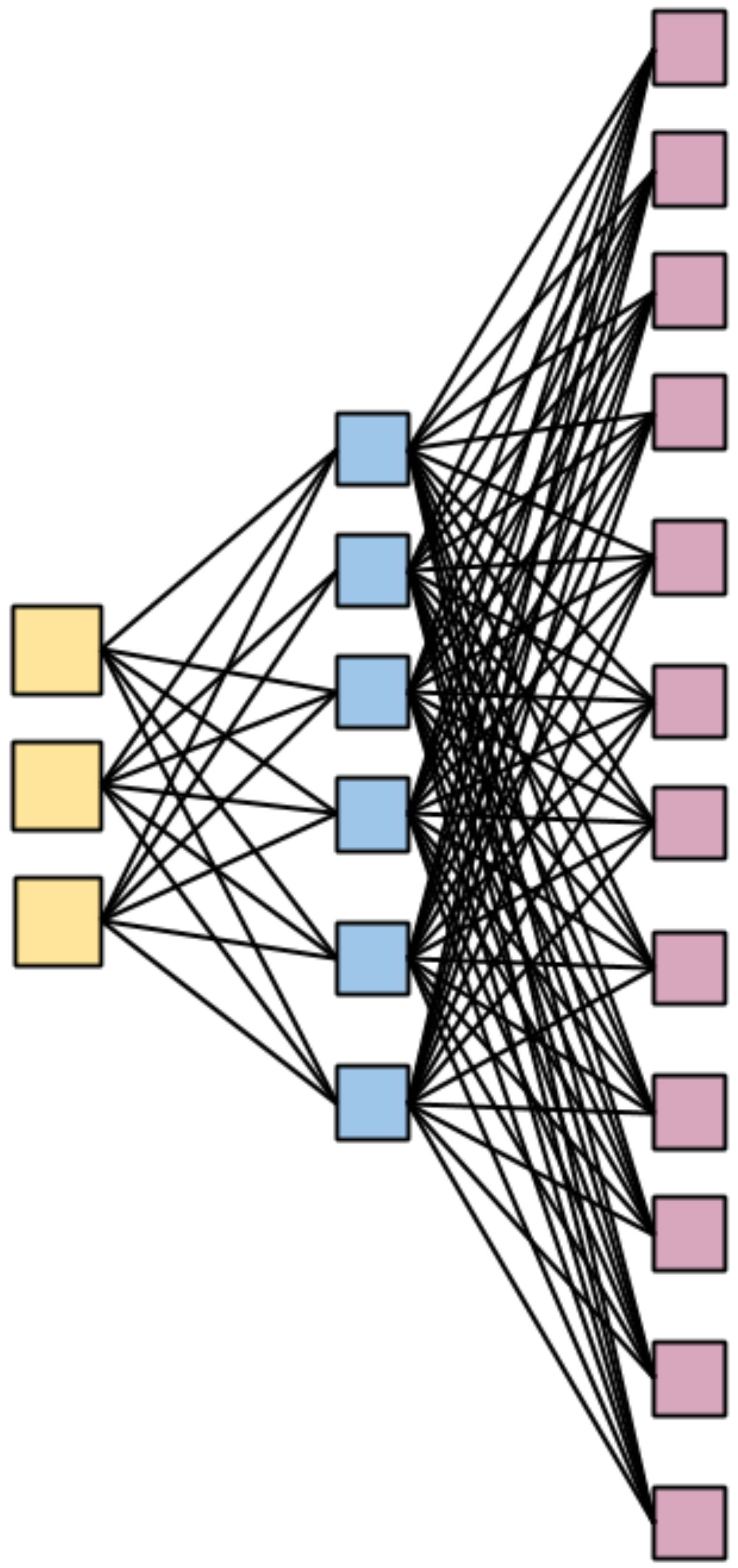


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

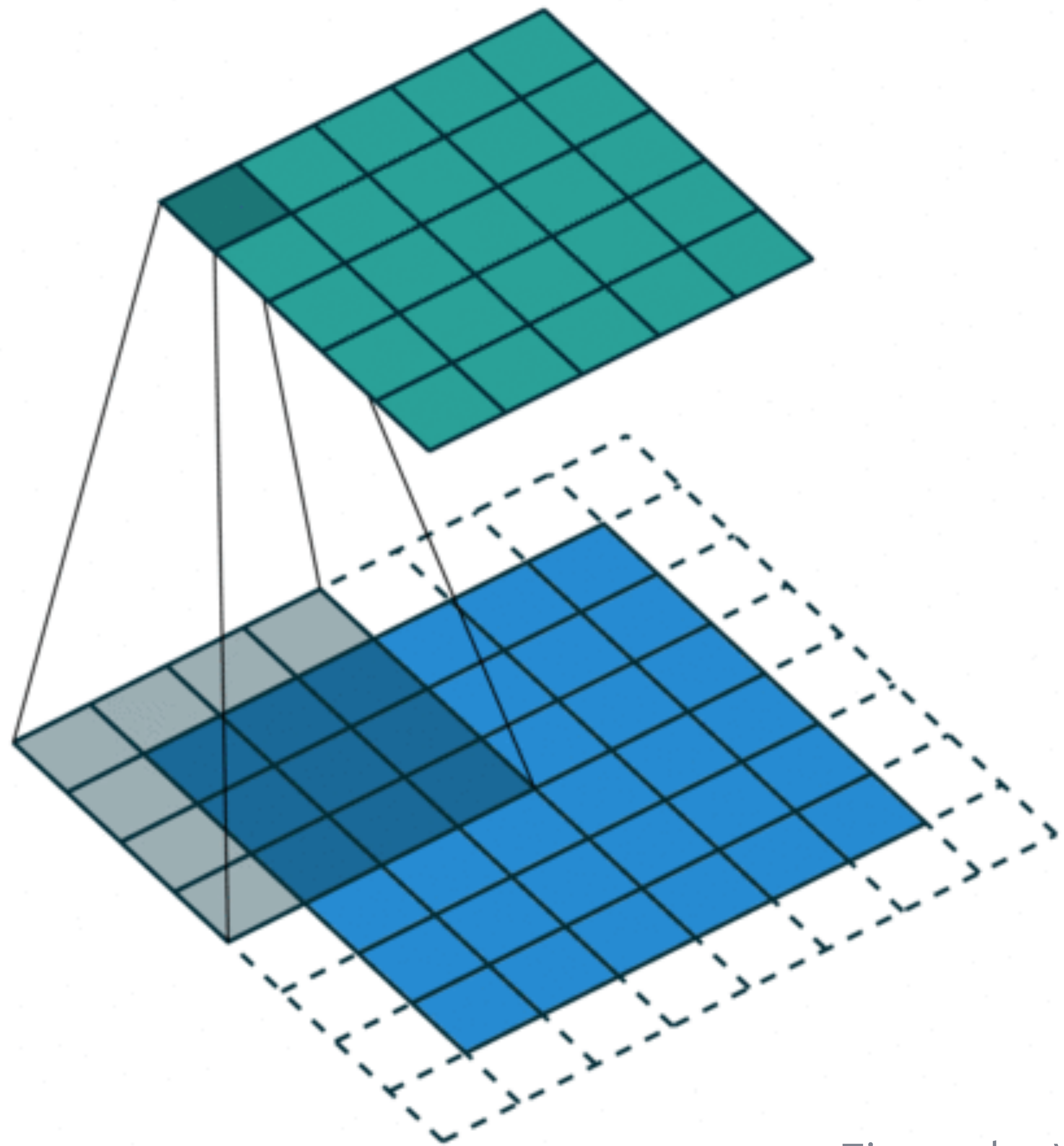
Convolution



```
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



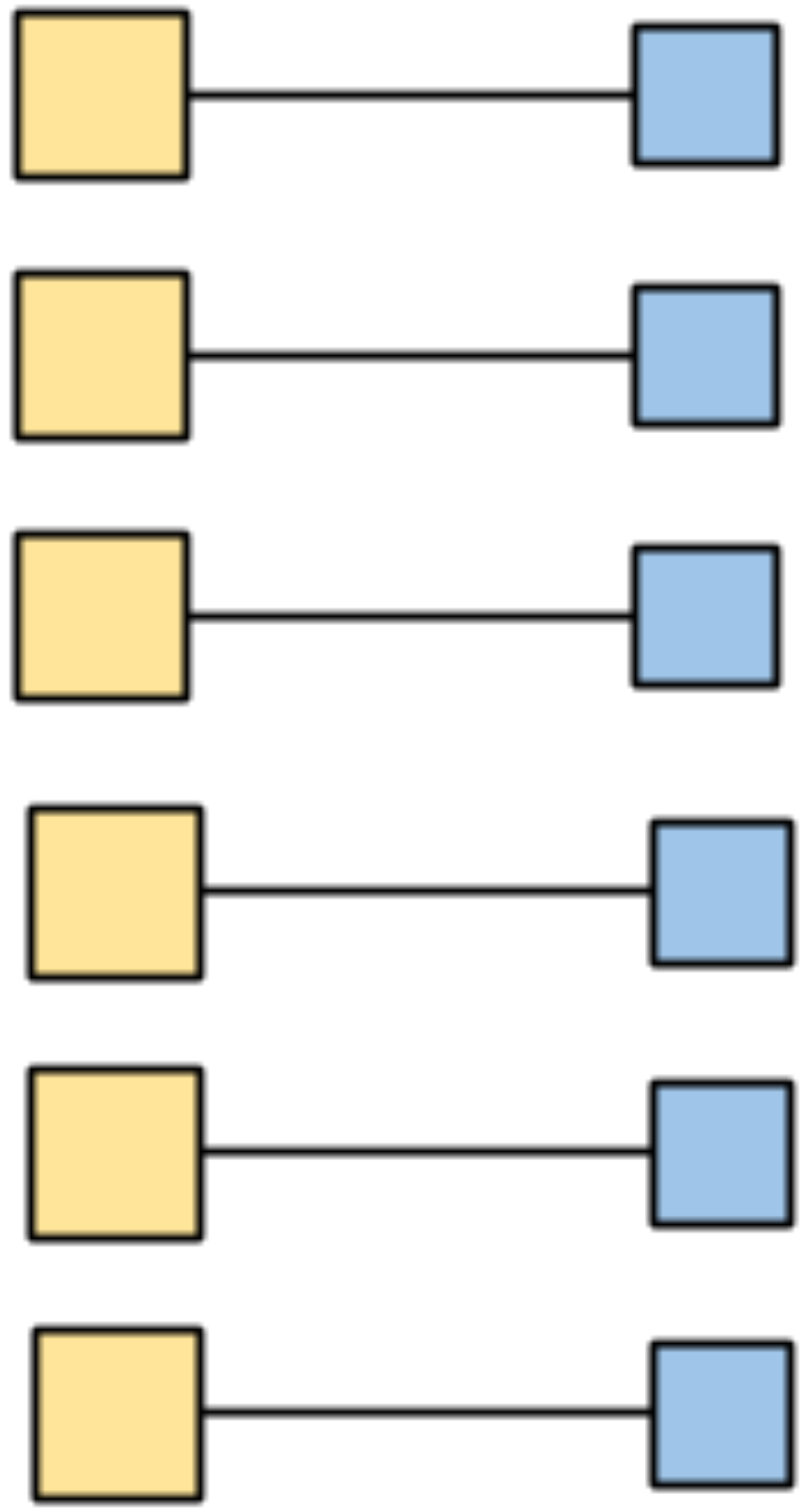


Figure by Vincent Dumolin: https://github.com/vdumoulin/conv_arithmetic

Types of typical operators

Matrix Multiply

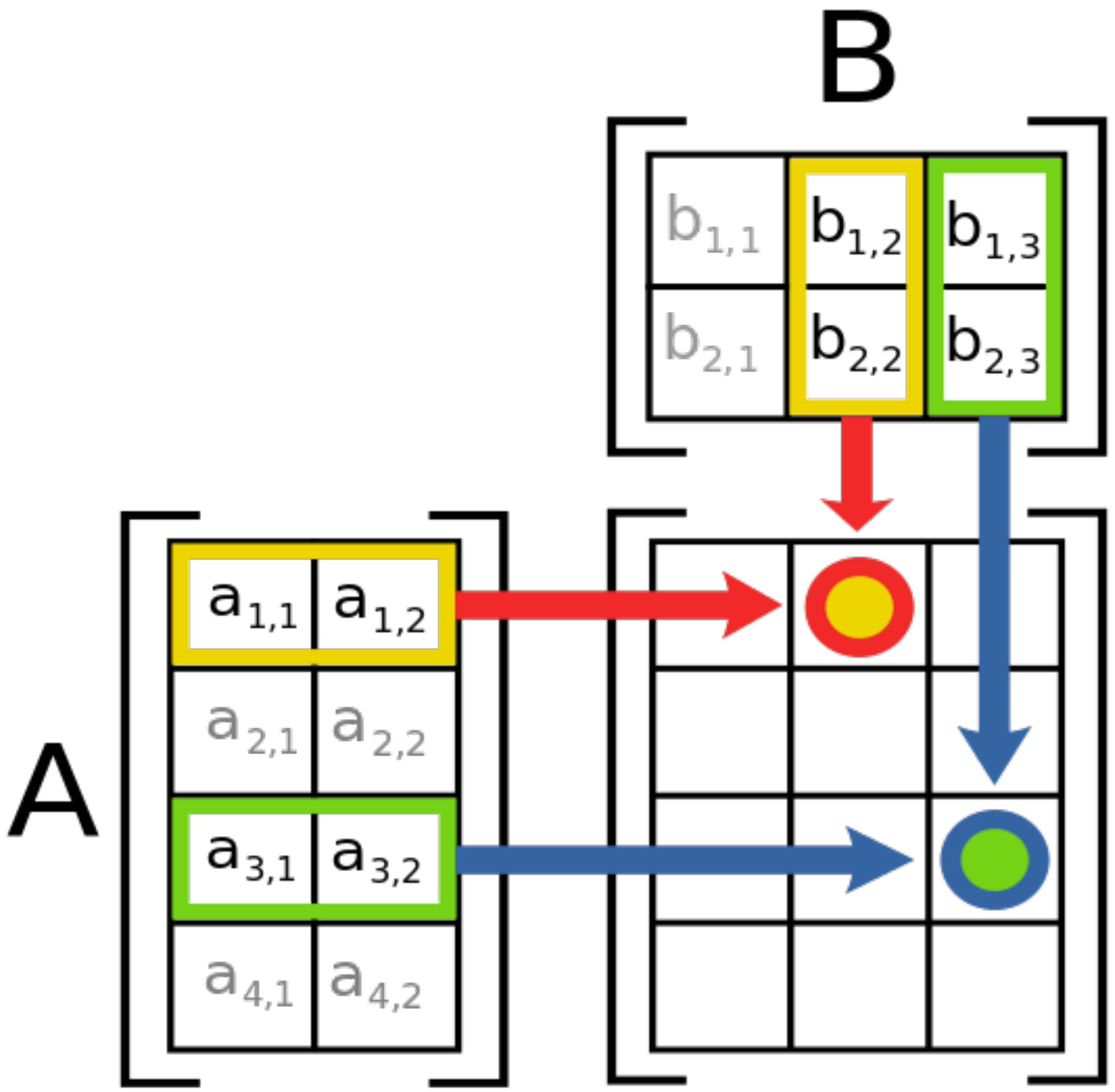


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

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

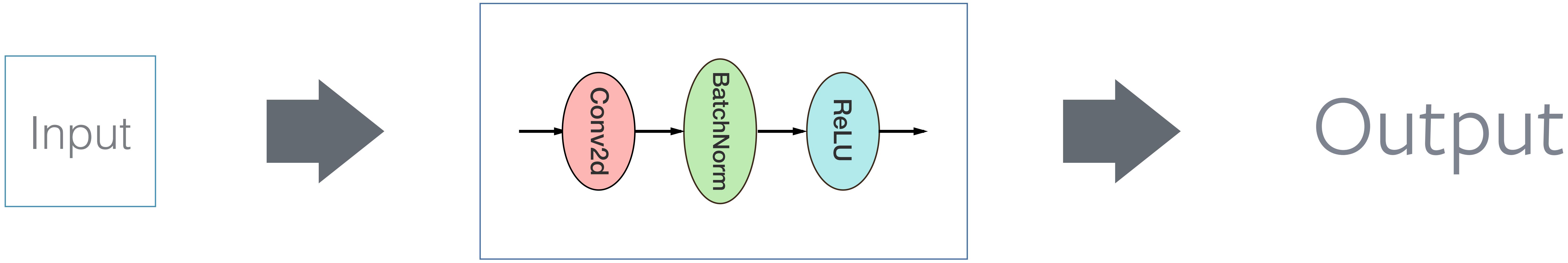


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

Chained Together

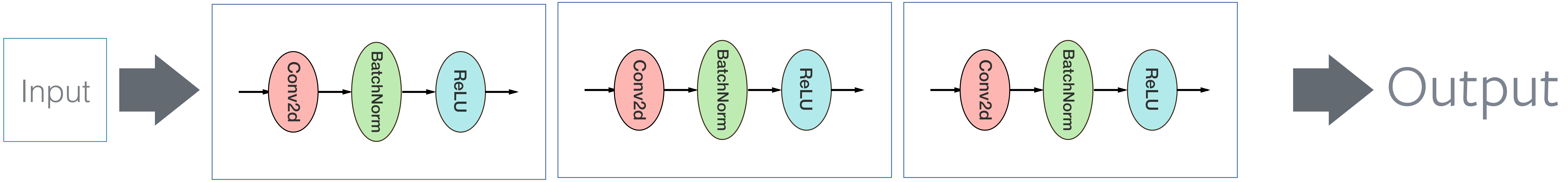


Figure by Wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication

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()
```

Pre-PyTorch

Caffe theano torch

meta programming meta programming imperative

Caffe

deploy.prototxt	[examples] switch examples + models to Input layers	3 years ago
readme.md	BVLC -> BAIR	2 years ago
solver.prototxt	Renaming CaffeNet model prototxts and unignoring models/*	4 years ago
train_val.prototxt	Upgrade existing nets using upgrade_net_proto_text tool	4 years ago

readme.md

name	caffemodel	caffemodel_url	license	
BAIR/BVLC AlexNet Model	bvlc_alexnet.caffemodel	http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel	unrestricted	91

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
```

Caffe

deploy.prototxt	[examples] switch examples + models to Input layers	3 years ago
readme.md	BVLC -> BAIR	2 years ago
solver.prototxt	Renaming CaffeNet model prototxts and unignoring models/*	4 years ago
train_val.prototxt		

readme.md

name	caffemodel	caffemodel_url	license	
BAIR/BVLC AlexNet Model	bvlc_alexnet.caffemodel	http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel	unrestricted	91

define protobuf, run via command-line utility

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
```

Caffe

deploy.prototxt	[examples] switch examples + models to Input layers	3 years ago
readme.md	BVLC -> BAIR	2 years ago
solver.prototxt	Renaming CaffeNet model prototxts and unignoring models/*	4 years ago
train_val.prototxt		

readme.md

name	caffemodel	caffemodel_url	license
BAIR/BVLC AlexNet Model			

define protobuf, run via command-line utility

small, efficient library. Could do convents well.

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
```

Theano

```python
# #################### BUILD NETWORK ##########################
# allocate symbolic variables for the data
# 'rand' is a random array used for random cropping/mirroring of data
x = T.ftensor4('x')
y = T.lvector('y')
rand = T.fvector('rand')

print '... building the model'
self.layers = []
params = []
weight_types = []

if flag_datalayer:
    data_layer = DataLayer(input=x, image_shape=(3, 256, 256,
                                                 batch_size),
                           cropsize=227, rand=rand, mirror=True,
                           flag_rand=config['rand_crop'])

    layer1_input = data_layer.output
else:
    layer1_input = x

convpool_layer1 = ConvPoolLayer(input=layer1_input,
                                image_shape=(3, 227, 227, batch_size),
                                filter_shape=(3, 11, 11, 96),
                                convstride=4, padsize=0, group=1,
                                poolsize=3, poolstride=2,
                                bias_init=0.0, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer1)
params += convpool_layer1.params
weight_types += convpool_layer1.weight_type

convpool_layer2 = ConvPoolLayer(input=convpool_layer1.output,
                                image_shape=(96, 27, 27, batch_size),
                                filter_shape=(96, 5, 5, 256),
                                convstride=1, padsize=2, group=2,
                                poolsize=3, poolstride=2,
                                bias_init=0.1, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer2)
params += convpool_layer2.params
weight_types += convpool_layer2.weight_type
```

```python
def compile_models(model, config, flag_top_5=False):

    x = model.x
    y = model.y
    rand = model.rand
    weight_types = model.weight_types

    cost = model.cost
    params = model.params
    errors = model.errors
    errors_top_5 = model.errors_top_5
    batch_size = model.batch_size

    mu = config['momentum']
    eta = config['weight_decay']

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)
    updates = []

    learning_rate = theano.shared(np.float32(config['learning_rate']))
    lr = T.scalar('lr')  # symbolic learning rate

    if config['use_data_layer']:
        raw_size = 256
    else:
        raw_size = 227

    shared_x = theano.shared(np.zeros((3, raw_size, raw_size,
                                       batch_size),
                                      dtype=theano.config.floatX),
                             borrow=True)
    shared_y = theano.shared(np.zeros((batch_size,), dtype=int),
                             borrow=True)

    rand_arr = theano.shared(np.zeros(3, dtype=theano.config.floatX),
                             borrow=True)

    vels = [theano.shared(param_i.get_value() * 0.)
            for param_i in params]

    if config['use_momentum']:

        assert len(weight_types) == len(params)

        for param_i, grad_i, vel_i, weight_type in \
```

```python
# Define Theano Functions

train_model = theano.function([], cost, updates=updates,
                              givens=[(x, shared_x), (y, shared_y),
                                      (lr, learning_rate),
                                      (rand, rand_arr)])

validate_outputs = [cost, errors]
if flag_top_5:
    validate_outputs.append(errors_top_5)

validate_model = theano.function([], validate_outputs,
                                 givens=[(x, shared_x), (y, shared_y),
                                         (rand, rand_arr)])

train_error = theano.function(
    [], errors, givens=[(x, shared_x), (y, shared_y), (rand, rand_arr)])

return (train_model, validate_model, train_error,
        learning_rate, shared_x, shared_y, rand_arr, vels)
```

Theano

```python
# #################### BUILD NETWORK ##########################
# allocate symbolic variables for the data
# 'rand' is a random array used for random cropping/mirroring of data
x = T.ftensor4('x')
y = T.lvector('y')
rand = T.fvector('rand')

print '... building the model'
self.layers = []
params = []
weight_types = []

if flag_datalayer:
    data_layer = DataLayer(input=x, image_shape=(3, 256, 256,
                                                 batch_size),
                           cropsize=227, rand=rand, mirror=True,

    layer1_input = c
else:
    layer1_input = x

convpool_layer1 = ConvPoolLayer(input=layer1_input,
                                image_shape=(3, 227, 227, batch_size),
                                filter_shape=(3, 11, 11, 96),
                                convstride=4, padsize=0, group=1,
                                poolsize=3, poolstride=2,
                                bias_init=0.0, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer1)
params += convpool_layer1.params
weight_types += convpool_layer1.weight_type

convpool_layer2 = ConvPoolLayer(input=convpool_layer1.output,
                                image_shape=(96, 27, 27, batch_size),
                                filter_shape=(96, 5, 5, 256),
                                convstride=1, padsize=2, group=2,
                                poolsize=3, poolstride=2,
                                bias_init=0.1, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer2)
params += convpool_layer2.params
weight_types += convpool_layer2.weight_type
```

```python
def compile_models(model, config, flag_top_5=False):

    x = model.x
    y = model.y
    rand = model.rand
    weight_types = model.weight_types

    cost = model.cost
    params = model.params
    errors = model.errors
    errors_top_5 = model.errors_top_5
    batch_size = model.batch_size

    mu = config['momentum']
    eta = config['weight_decay']

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)
    updates = []

    if config['use_data_layer']:
        raw_size = 256
    else:
        raw_size = 227

    shared_x = theano.shared(np.zeros((3, raw_size, raw_size,
                                       batch_size),
                                      dtype=theano.config.floatX),
                             borrow=True)
    shared_y = theano.shared(np.zeros((batch_size,), dtype=int),
                             borrow=True)

    rand_arr = theano.shared(np.zeros(3, dtype=theano.config.floatX),
                             borrow=True)

    vels = [theano.shared(param_i.get_value() * 0.)
            for param_i in params]

    if config['use_momentum']:

        assert len(weight_types) == len(params)

        for param_i, grad_i, vel_i, weight_type in \
```

meta-program Theano VM via Python API

```python
# Define Theano Functions

train_model = theano.function([], cost, updates=updates,
                              givens=[(x, shared_x), (y, shared_y),
                                      (lr, learning_rate),
                                      (rand, rand_arr)])

validate_outputs = [cost, errors]
if flag_top_5:
                                  s_top_5)

                                  [], validate_outputs,
                                  givens=[(x, shared_x), (y, shared_y),
                                          (rand, rand_arr)])

train_error = theano.function(
    [], errors, givens=[(x, shared_x), (y, shared_y), (rand, rand_arr)])

return (train_model, validate_model, train_error,
        learning_rate, shared_x, shared_y, rand_arr, vels)
```

Theano

```
# #################### BUILD NETWORK ########################
# allocate symbolic variables for the data
# 'rand' is a random array used for random cropping/mirroring of data
x = T.ftensor4('x')
y = T.lvector('y')
rand = T.fvector('rand')

print '... building the model'
self.layers = []
params = []
weight_types = []

if flag_datalayer:
    data_layer = DataLayer(input=x, image_shape=(3, 256, 256,
                                                 batch_size),
                           cropsize=227, rand=rand, mirror=True,


    layer1_input = c
else:
    layer1_input = x

convpool_layer1 =

                                filter_shape=(3, 11, 11, 96),
                                convstride=4, padsize=0, group=1,
                                poolsize=3, poolstride=2,
                                bias_init=0.0, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer1)
params += convpool_layer1.params
weight_types += convpool_layer1.weight_type

convpool_layer2 = ConvPoolLayer(input=convpool_layer1.output,
                                image_shape=(96, 27, 27, batch_size),
                                filter_shape=(96, 5, 5, 256),
                                convstride=1, padsize=2, group=2,
                                poolsize=3, poolstride=2,
                                bias_init=0.1, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer2)
params += convpool_layer2.params
weight_types += convpool_layer2.weight_type
```

```
def compile_models(model, config, flag_top_5=False):

    x = model.x
    y = model.y
    rand = model.rand
    weight_types = model.weight_types

    cost = model.cost
    params = model.params
    errors = model.errors
    errors_top_5 = model.errors_top_5
    batch_size = model.batch_size

    mu = config['momentum']
    eta = config['weight_decay']

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)
    updates = []



    shared_x = theano.shared(np.zeros((3, raw_size, raw_size,
                                       batch_size),
                                      dtype=theano.config.floatX),
                             borrow=True)
    shared_y = theano.shared(np.zeros((batch_size,), dtype=int),
                             borrow=True)

    rand_arr = theano.shared(np.zeros(3, dtype=theano.config.floatX),
                             borrow=True)

    vels = [theano.shared(param_i.get_value() * 0.)
            for param_i in params]

    if config['use_momentum']:

        assert len(weight_types) == len(params)

        for param_i, grad_i, vel_i, weight_type in \
```

```
    # Define Theano Functions

    train_model = theano.function([], cost, updates=updates,
                                  givens=[(x, shared_x), (y, shared_y),
                                          (lr, learning_rate),
                                          (rand, rand_arr)])

    validate_outputs = [cost, errors]
    if flag_top_5:
                                            s_top_5)

                                            [], validate_outputs,
                                            givens=[(x, shared_x), (y, shared_y),
                                                    (rand, rand_arr)])


                                            , (y, shared_y), (rand, rand_arr)])

    return (train_model, validate_model, train_error,
            learning_rate, shared_x, shared_y, rand_arr, vels)
```

meta-program Theano VM via Python API

whole program optimizations, graph fusion

Theano

meta-program Theano VM via Python API

whole program optimizations, graph fusion

graphs took minutes to hours to compile and start

```
# #################### BUILD NETWORK #########################
# allocate symbolic variables for the data
# 'rand' is a random array used for random cropping/mirroring of data
x = T.ftensor4('x')
y = T.lvector('y')
rand = T.fvector('rand')

print '... building the model'
self.layers = []
params = []
weight_types = []

if flag_datalayer:
    data_layer = DataLayer(input=x, image_shape=(3, 256, 256,
                                                 batch_size),
                           cropsize=227, rand=rand, mirror=True,

    layer1_input = c
else:
    layer1_input = x

convpool_layer1 =

                filter_shape=(3, 11, 11, 96),

                )
self.layers.append(convpool_layer1)
params += convpool_layer1.params
weight_types += convpool_layer1.weight_type

convpool_layer2 = ConvPoolLayer(input=convpool_layer1.output,
                                image_shape=(96, 27, 27, batch_size),
                                filter_shape=(96, 5, 5, 256),
                                convstride=1, padsize=2, group=2,
                                poolsize=3, poolstride=2,
                                bias_init=0.1, lrn=True,
                                lib_conv=lib_conv,
                                )
self.layers.append(convpool_layer2)
params += convpool_layer2.params
weight_types += convpool_layer2.weight_type
```

```
def compile_models(model, config, flag_top_5=False):

    x = model.x
    y = model.y
    rand = model.rand
    weight_types = model.weight_types

    cost = model.cost
    params = model.params
    errors = model.errors
    errors_top_5 = model.errors_top_5
    batch_size = model.batch_size

    mu = config['momentum']
    eta = config['weight_decay']

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)
    updates = []

    shared_y = theano.shared(np.zeros((batch_size,), dtype=int),
                             borrow=True)

    rand_arr = theano.shared(np.zeros(3, dtype=theano.config.floatX),
                             borrow=True)

    vels = [theano.shared(param_i.get_value() * 0.)
        for param_i in params]

    if config['use_momentum']:

        assert len(weight_types) == len(params)

        for param_i, grad_i, vel_i, weight_type in \
```

```
# Define Theano Functions

train_model = theano.function([], cost, updates=updates,
                              givens=[(x, shared_x), (y, shared_y),
                                      (lr, learning_rate),
                                      (rand, rand_arr)])

validate_outputs = [cost, errors]
if flag_top_5:
                                                   s_top_5)

                                          [], validate_outputs,
                                          givens=[(x, shared_x), (y, shared_y),
                                                  (rand, rand_arr)])

), (y, shared_y), (rand, rand_arr)])

, vels)
```

Torch-7

```lua
function alexnet(lib)
   local SpatialConvolution = lib[1]
   local SpatialMaxPooling = lib[2]
   local ReLU = lib[3]
   local SpatialZeroPadding = nn.SpatialZeroPadding
   local padding = true
   local stride1only = false

   -- from https://code.google.com/p/cuda-convnet2/source/browse/layers/layers-imagenet-1gpu.cfg
   -- this is AlexNet that was presented in the One Weird Trick paper. http://arxiv.org/abs/1404.5997
   local features = nn.Sequential()
   features:add(SpatialConvolution(3,64,11,11,4,4,2,2))       -- 224 -> 55
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   -- 55 ->  27
   features:add(SpatialConvolution(64,192,5,5,1,1,2,2))       --  27 -> 27
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   --  27 ->  13
   features:add(SpatialConvolution(192,384,3,3,1,1,1,1))      --  13 ->  13
   features:add(ReLU(true))
   features:add(SpatialConvolution(384,256,3,3,1,1,1,1))      --  13 ->  13
   features:add(ReLU(true))
   features:add(SpatialConvolution(256,256,3,3,1,1,1,1))      --  13 ->  13
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   -- 13 -> 6

   local classifier = nn.Sequential()
   classifier:add(nn.View(256*6*6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(256*6*6, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(4096, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   classifier:add(nn.Linear(4096, 1000))
   -- classifier:add(nn.LogSoftMax())

   features:get(1).gradInput = nil

   local model = nn.Sequential()
   model:add(features):add(classifier)

   return model,'AlexNet',{128,3,224,224}
end

return alexnet
```

```lua
-- 4. trainBatch - Used by train() to train a single batch after the data is loaded.
function trainBatch(inputsCPU, labelsCPU)
   cutorch.synchronize()
   collectgarbage()
   local dataLoadingTime = dataTimer:time().real
   timer:reset()

   -- transfer over to GPU
   inputs:resize(inputsCPU:size()):copy(inputsCPU)
   labels:resize(labelsCPU:size()):copy(labelsCPU)

   local err, outputs
   feval = function(x)
      model:zeroGradParameters()
      outputs = model:forward(inputs)
      err = criterion:forward(outputs, labels)
      local gradOutputs = criterion:backward(outputs, labels)
      model:backward(inputs, gradOutputs)
      return err, gradParameters
   end
   optim.sgd(feval, parameters, optimState)

   cutorch.synchronize()
   batchNumber = batchNumber + 1
   loss_epoch = loss_epoch + err
   -- top-1 error
   local top1 = 0
   do
      local _,prediction_sorted = outputs:float():sort(2, true) -- descending
      for i=1,opt.batchSize do
        if prediction_sorted[i][1] == labelsCPU[i] then
           top1_epoch = top1_epoch + 1;
           top1 = top1 + 1
        end
      end
      top1 = top1 * 100 / opt.batchSize;
   end
   -- Calculate top-1 error, and print information
   print(('Epoch: [%d][%d/%d]\tTime %.3f Err %.4f Top1-%%: %.2f LR %.0e DataLoadingTime %.3f'):format(
          epoch, batchNumber, opt.epochSize, timer:time().real, err, top1,
          optimState.learningRate, dataLoadingTime))

   dataTimer:reset()
end
```

Torch-7

```lua
function alexnet(lib)
   local SpatialConvolution = lib[1]
   local SpatialMaxPooling = lib[2]
   local ReLU = lib[3]
   local SpatialZeroPadding = nn.SpatialZeroPadding
   local padding = true
   local stride1only = false

   -- from https://code.google.com/p/cuda-convnet2/source/browse/layers/layers-imagenet-1gpu.cfg
   -- this is AlexNet that was presented in the One Weird Trick paper. http://arxiv.org/abs/1404.5997
   local features = nn.Sequential()
   features:add(SpatialConvolution(3,64,11,11,4,4,2,2))       -- 224 -> 55
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   -- 55 ->  27
   features:add(SpatialConvolution(64,192,5,5,1,1,2,2))       --  27 -> 27
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   --  27 ->  13
   features:add(SpatialConvolution(
   features:add(ReLU(true))
   features:add(SpatialConvolution(
   features:add(ReLU(true))
   features:add(SpatialConvolution(
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3

   local classifier = nn.Sequential()
   classifier:add(nn.View(256*6*6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(256*6*6, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(4096, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   classifier:add(nn.Linear(4096, 1000))
   -- classifier:add(nn.LogSoftMax())

   features:get(1).gradInput = nil

   local model = nn.Sequential()
   model:add(features):add(classifier)

   return model,'AlexNet',{128,3,224,224}
end

return alexnet
```

imperative programming in Lua

```lua
-- 4. trainBatch - Used by train() to train a single batch after the data is loaded.
function trainBatch(inputsCPU, labelsCPU)
   cutorch.synchronize()
   collectgarbage()
   local dataLoadingTime = dataTimer:time().real
   timer:reset()

   -- transfer over to GPU
   inputs:resize(inputsCPU:size()):copy(inputsCPU)
   labels:resize(labelsCPU:size()):copy(labelsCPU)

   local err, outputs
   feval = function(x)
      model:zeroGradParameters()
      outputs = model:forward(inputs)
      err = criterion:forward(outputs, labels)
      ...bels)

   loss_epoch = loss_epoch + err
   -- top-1 error
   local top1 = 0
   do
      local _,prediction_sorted = outputs:float():sort(2, true) -- descending
      for i=1,opt.batchSize do
         if prediction_sorted[i][1] == labelsCPU[i] then
            top1_epoch = top1_epoch + 1;
            top1 = top1 + 1
         end
      end
      top1 = top1 * 100 / opt.batchSize;
   end
   -- Calculate top-1 error, and print information
   print(('Epoch: [%d][%d/%d]\tTime %.3f Err %.4f Top1-%%: %.2f LR %.0e DataLoadingTime %.3f'):format(
         epoch, batchNumber, opt.epochSize, timer:time().real, err, top1,
         optimState.learningRate, dataLoadingTime))

   dataTimer:reset()
end
```

Torch-7

```lua
function alexnet(lib)
   local SpatialConvolution = lib[1]
   local SpatialMaxPooling = lib[2]
   local ReLU = lib[3]
   local SpatialZeroPadding = nn.SpatialZeroPadding
   local padding = true
   local stride1only = false

   -- from https://code.google.com/p/cuda-convnet2/source/browse/layers/layers-imagenet-1gpu.cfg
   -- this is AlexNet that was presented in the One Weird Trick paper. http://arxiv.org/abs/1404.5997
   local features = nn.Sequential()
   features:add(SpatialConvolution(3,64,11,11,4,4,2,2))       -- 224 -> 55
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   -- 55 ->  27
   features:add(SpatialConvolution(64,192,5,5,1,1,2,2))       --  27 -> 27
```

imperative programming in Lua

tied closely to underlying C89 implementations

```lua
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(256*6*6, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(4096, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   classifier:add(nn.Linear(4096, 1000))
   -- classifier:add(nn.LogSoftMax())

   features:get(1).gradInput = nil

   local model = nn.Sequential()
   model:add(features):add(classifier)

   return model,'AlexNet',{128,3,224,224}
end

return alexnet
```

```lua
-- 4. trainBatch - Used by train() to train a single batch after the data is loaded.
function trainBatch(inputsCPU, labelsCPU)
   cutorch.synchronize()
   collectgarbage()
   local dataLoadingTime = dataTimer:time().real
   timer:reset()

   -- transfer over to GPU
   inputs:resize(inputsCPU:size()):copy(inputsCPU)
   labels:resize(labelsCPU:size()):copy(labelsCPU)

   local err, outputs
   feval = function(x)
      model:zeroGradParameters()
```

```lua
   local top1 = 0
   do
      local _,prediction_sorted = outputs:float():sort(2, true) -- descending
      for i=1,opt.batchSize do
        if prediction_sorted[i][1] == labelsCPU[i] then
           top1_epoch = top1_epoch + 1;
           top1 = top1 + 1
        end
      end
      top1 = top1 * 100 / opt.batchSize;
   end
   -- Calculate top-1 error, and print information
   print(('Epoch: [%d][%d/%d]\tTime %.3f Err %.4f Top1-%%: %.2f LR %.0e DataLoadingTime %.3f'):format(
         epoch, batchNumber, opt.epochSize, timer:time().real, err, top1,
         optimState.learningRate, dataLoadingTime))

   dataTimer:reset()
end
```

Torch-7

```lua
function alexnet(lib)
   local SpatialConvolution = lib[1]
   local SpatialMaxPooling = lib[2]
   local ReLU = lib[3]
   local SpatialZeroPadding = nn.SpatialZeroPadding
   local padding = true
   local stride1only = false

   -- from https://code.google.com/p/cuda-convnet2/source/browse/layers/layers-imagenet-1gpu.cfg
   -- this is AlexNet that was presented in the One Weird Trick paper. http://arxiv.org/abs/1404.5997
   local features = nn.Sequential()
   features:add(SpatialConvolution(3,64,11,11,4,4,2,2))       -- 224 -> 55
   features:add(ReLU(true))
   features:add(SpatialMaxPooling(3,3,2,2))                   -- 55 ->  27
   features:add(SpatialConvolution(64,192,5,5,1,1,2,2))       --  27 -> 27
```

imperative programming in Lua

tied closely to underlying C89 implementations

Lua lacked good tooling and ecosystem

```lua
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(256*6*6, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   -- classifier:add(nn.Dropout(0.5))
   classifier:add(nn.Linear(4096, 4096))
   classifier:add(nn.Threshold(0, 1e-6))
   classifier:add(nn.Linear(4096, 1000))
   -- classifier:add(nn.LogSoftMax())

   features:get(1).gradInput = nil

   local model = nn.Sequential()
   model:add(features):add(classifier)

   return model,'AlexNet',{128,3,224,224}
end

return alexnet
```

```lua
-- 4. trainBatch - Used by train() to train a single batch after the data is loaded.
function trainBatch(inputsCPU, labelsCPU)
   cutorch.synchronize()
   collectgarbage()
   local dataLoadingTime = dataTimer:time().real
   timer:reset()

   -- transfer over to GPU
   inputs:resize(inputsCPU:size()):copy(inputsCPU)
   labels:resize(labelsCPU:size()):copy(labelsCPU)

   local err, outputs
   feval = function(x)
      model:zeroGradParameters()
```

```lua
   local top1 = 0
   do
      local _,prediction_sorted = outputs:float():sort(2, true) -- descending
      for i=1,opt.batchSize do
        if prediction_sorted[i][1] == labelsCPU[i] then
           top1_epoch = top1_epoch + 1;
           top1 = top1 + 1
        end
      end
      top1 = top1 * 100 / opt.batchSize;
   end
   -- Calculate top-1 error, and print information
   print(('Epoch: [%d][%d/%d]\tTime %.3f Err %.4f Top1-%%: %.2f LR %.0e DataLoadingTime %.3f'):format(
          epoch, batchNumber, opt.epochSize, timer:time().real, err, top1,
          optimState.learningRate, dataLoadingTime))

   dataTimer:reset()
end
```

What is PyTorch?

automatic differentiation engine

Ndarray library with GPU support

gradient based optimization package

Utilities (data loading, etc.)

Deep Learning

Reinforcement Learning

Numpy-alternative

ndarray library

- np.ndarray <-> torch.Tensor
- 200+ operations, similar to numpy
- very fast acceleration on NVIDIA GPUs

Numpy

```python
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch

```python
import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

ndarray / Tensor library

Tensors are similar to numpy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
from __future__ import print_function
import torch
```

Construct a 5x3 matrix, uninitialized:

```
x = torch.Tensor(5, 3)
print(x)
```

Out:

```
1.00000e-25 *
  0.4136  0.0000  0.0000
  0.0000  1.6519  0.0000
  1.6518  0.0000  1.6519
  0.0000  1.6518  0.0000
  1.6520  0.0000  1.6519
[torch.FloatTensor of size 5x3]
```

ndarray / Tensor library

Construct a randomly initialized matrix

```
x = torch.rand(5, 3)
print(x)
```

Out:

```
0.2598  0.7231  0.8534
 0.3928  0.1244  0.5110
 0.5476  0.2700  0.5856
 0.7288  0.9455  0.8749
 0.6663  0.8230  0.2713
[torch.FloatTensor of size 5x3]
```

Get its size

```
print(x.size())
```

Out:

```
torch.Size([5, 3])
```

ndarray / Tensor library

You can use standard numpy-like indexing with all bells and whistles!

```
print(x[:, 1])
```

Out:

```
0.7231
 0.1244
 0.2700
 0.9455
 0.8230
[torch.FloatTensor of size 5]
```

ndarray / Tensor library

```
y = torch.rand(5, 3)
print(x + y)
```

Out:

```
0.7931  1.1872  1.6143
 1.1946  0.4669  0.9639
 0.7576  0.8136  1.1897
 0.7431  1.8579  1.3400
 0.8188  1.1041  0.8914
[torch.FloatTensor of size 5x3]
```

NumPy bridge

Converting torch Tensor to numpy Array

```
a = torch.ones(5)
print(a)
```

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

```
b = a.numpy()
print(b)
```

Out:

```
[ 1.  1.  1.  1.  1.]
```

NumPy bridge

Converting torch Tensor to numpy Array

```
a = torch.ones(5)
print(a)
```

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

Zero memory-copy
very efficient

```
b = a.numpy()
print(b)
```

Out:

```
[ 1.  1.  1.  1.  1.]
```

NumPy bridge

See how the numpy array changed in value.

```
a.add_(1)
print(a)
print(b)
```

Out:

```
2
 2
 2
 2
 2
[torch.FloatTensor of size 5]

[ 2.  2.  2.  2.  2.]
```

NumPy bridge

Converting numpy Array to torch Tensor

See how changing the np array changed the torch Tensor automatically

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[ 2.  2.  2.  2.  2.]

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

Seamless GPU Tensors

CUDA Tensors

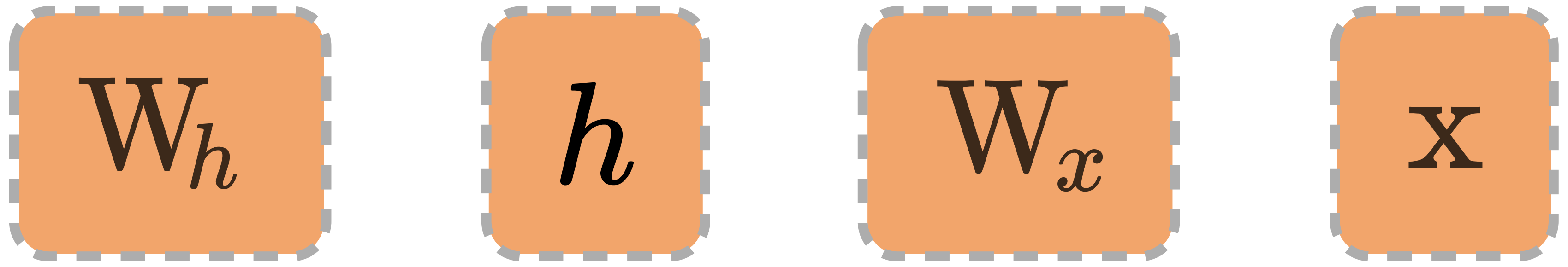
Tensors can be moved onto GPU using the `.cuda` function.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

automatic differentiation engine

for deep learning and reinforcement learning

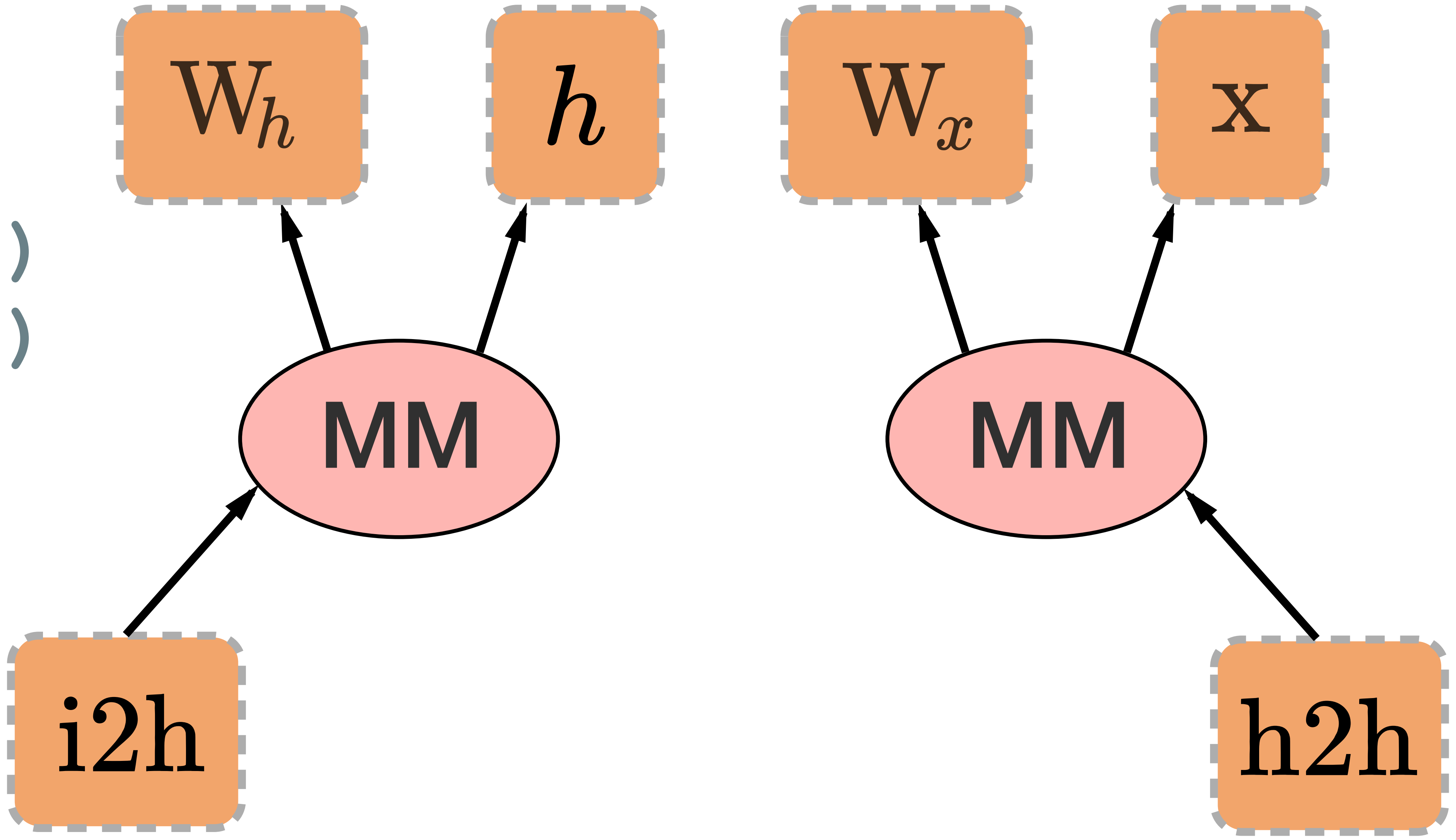
PyTorch Autograd




```
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)
```

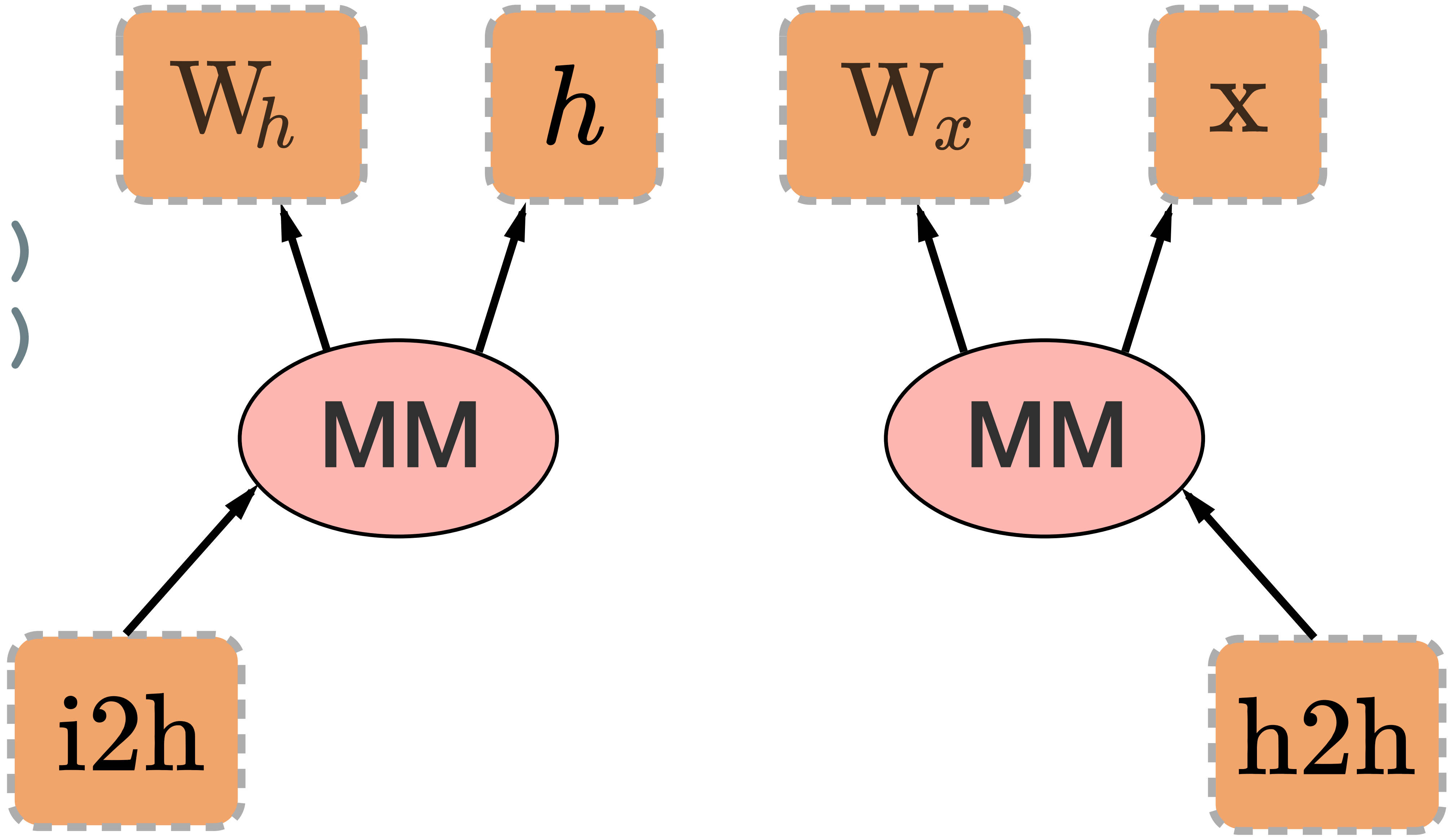
PyTorch Autograd

```
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())
```



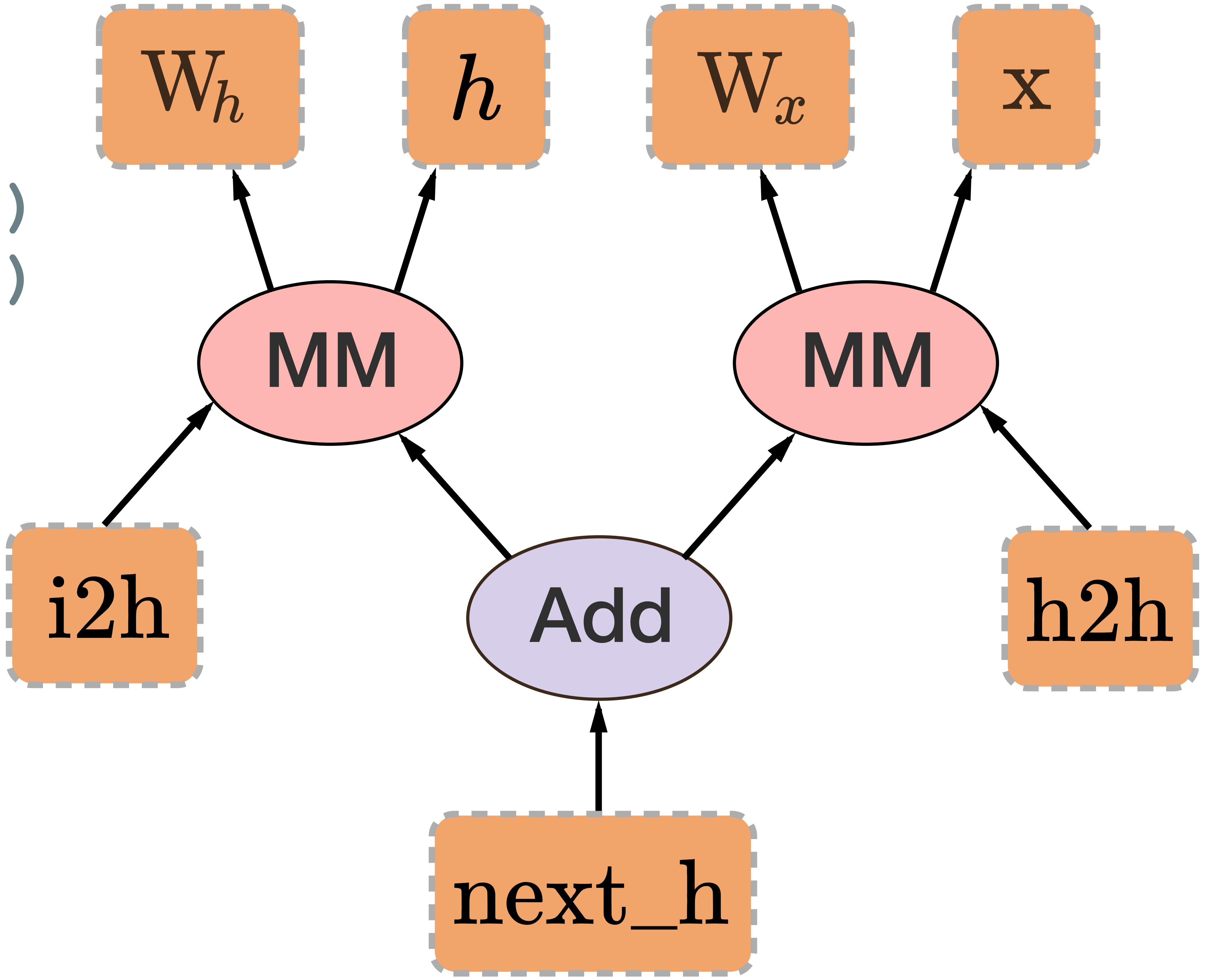
PyTorch Autograd

```
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
```



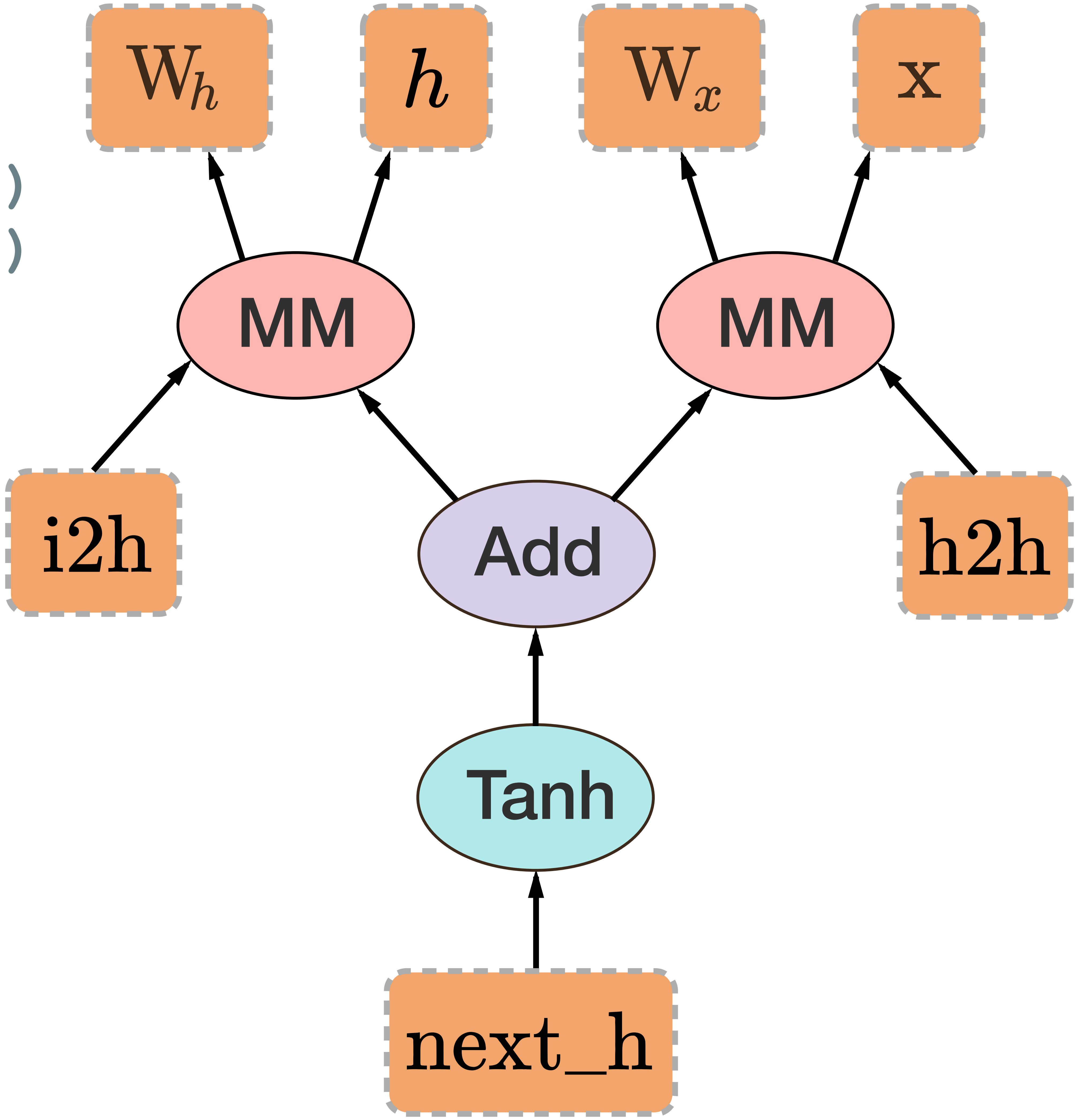
PyTorch Autograd

```
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
```



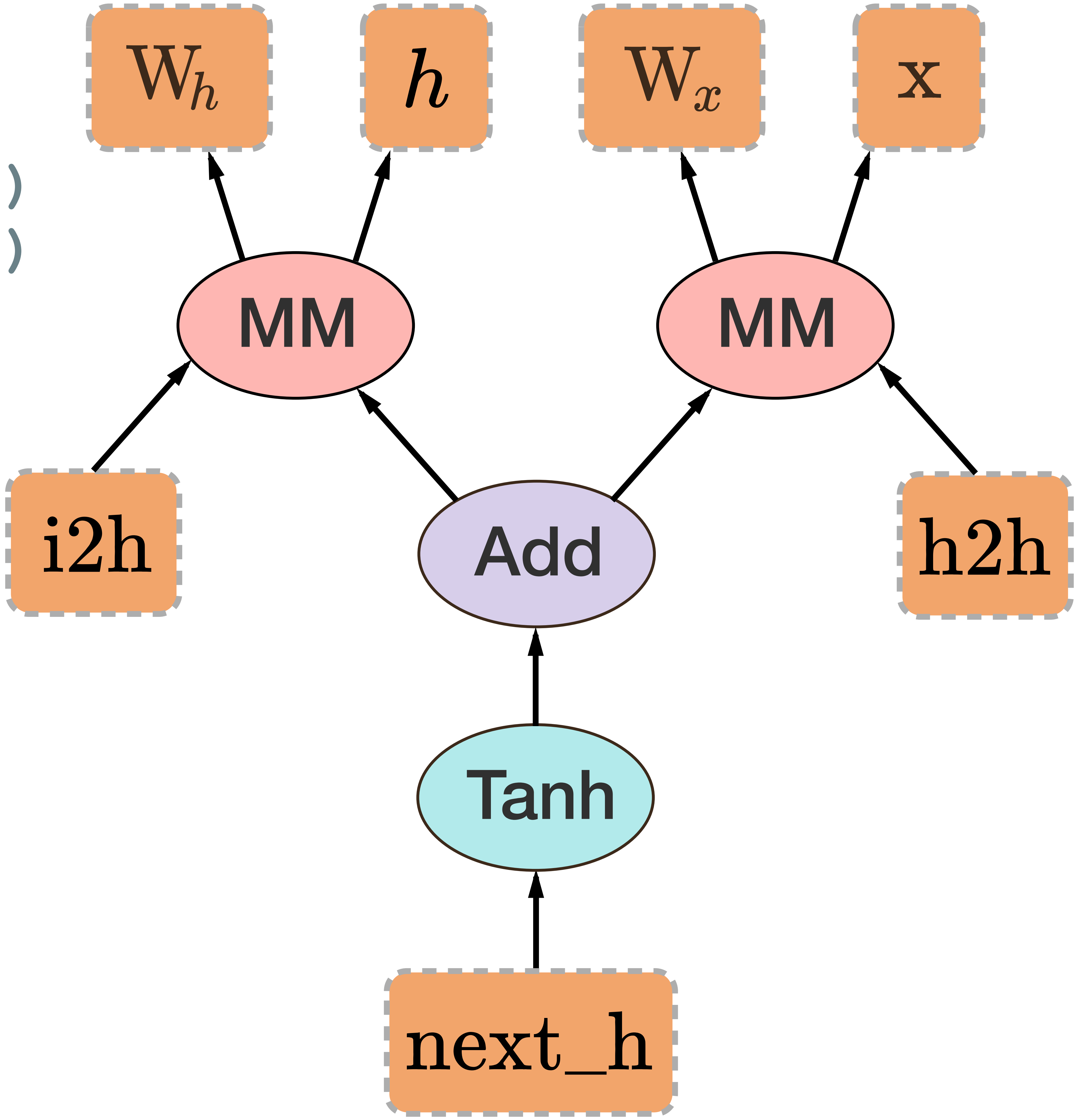
PyTorch Autograd

```
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))
```



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

```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

```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()
```

Bootstrapping

Writing Dataset loaders	Building models	Implementing Training loop	Checkpointing models

Python + PyTorch - an environment to do all of this

Interfacing with environments	Building optimizers	Dealing with GPUs	Building Baselines

Bootstrapping

Writing Dataset loaders	Building models	Implementing Training loop	Checkpointing models

bootstrapping the Python tooling stack for good UX

Interfacing with environments	Building optimizers	Dealing with GPUs	Building Baselines

Python is slow, interpreted

- Global interpreter-lock
- application logic is order of magnitude slower than C++
- moved autograd engine to C++
- moved everything to ATen
  - Side-effect, a clean C++ API