Introduction to PyTorch

Deep Learning Frameworks

torch
NYU

Caffe
UC Berkeley

theano
Montreal

PyTorch
Meta

TensorFlow
Google

mxnet
Amazon

ONNX
Microsoft

DL4J
Eclipse (IBM)

JAX
Google

Slide Idea: Justin Johnson

Why PyTorch

- Implements many necessary modules for DL

- Quick prototyping

- Easy debugging

- "Pythonic" programming style

Today

- Tensors   ~50%

- nn Module   ~20%

- Autograd   ~30%
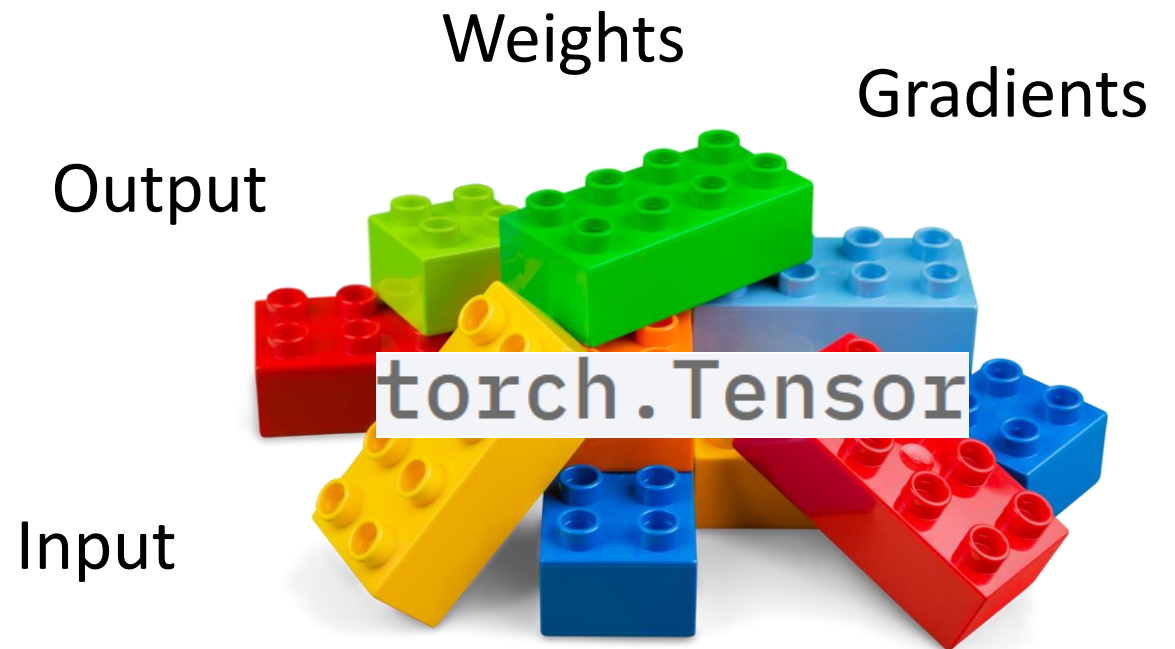
Tensors

```python
# creation from data
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

tensor([[1, 2],
        [3, 4]])
```

```python
# creation from numpy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)

tensor([[1, 2],
        [3, 4]])
```

Weights

Gradients

Output

Input

torch.Tensor

Based on Pytorch tutorials

Tensor Attributes

$$x\_data = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

torch.Tensor

shape

```
x_data.shape
```

```
torch.Size([2, 2])
```

dtype

```
x_data.dtype
```

```
torch.int64
```

device

```
x_data.device
```

```
device(type='cpu')
```

Managing Device

Do we have GPU resources available?

```
torch.cuda.is_available()
```

```
True
```

How many GPU resources are available?

```
torch.cuda.device_count()
```

```
1
```

Moving a Tensor from one device to another.

```
print(x_data.device)
x_gpu = x_data.to('cuda')
print(x_gpu.device)
```

```
cpu
cuda:0
```

Based on Pytorch tutorials

Tensor Operations

```python
# initialization
a = torch.tensor([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
b = torch.eye(3)


print(f"a: {a}")
print(f"b: {b}")
```

```
a: tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
b: tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

Basic Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Just like NumPy!

```
a + b
```
```
tensor([[ 2.,  2.,  3.],
        [ 4.,  6.,  6.],
        [ 7.,  8., 10.]])
```

```
a / b
```
```
tensor([[1., inf, inf],
        [inf, 5., inf],
        [inf, inf, 9.]])
```

```
a - b
```
```
tensor([[0., 2., 3.],
        [4., 4., 6.],
        [7., 8., 8.]])
```

```
a ** b
```
```
tensor([[1., 1., 1.],
        [1., 5., 1.],
        [1., 1., 9.]])
```

```
a * b
```
```
tensor([[1., 0., 0.],
        [0., 5., 0.],
        [0., 0., 9.]])
```

```
a @ b
```
```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

In-place Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Just like NumPy!

```
a + b
a
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a += b
a
```

```
tensor([[ 2.,  2.,  3.],
        [ 4.,  6.,  6.],
        [ 7.,  8., 10.]])
```

```
a -= b
a
```

```
tensor([[0., 2., 3.],
        [4., 4., 6.],
        [7., 8., 8.]])
```

```
a.clamp(2, 5)
a
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a.clamp_(2, 5)
a
```

```
tensor([[2., 2., 3.],
        [4., 5., 5.],
        [5., 5., 5.]])
```

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Indexing and slicing – just like NumPy!

Indexing

```
a[0,0]
```

```
tensor(1.)
```

```
a[1,2]
```

```
tensor(6.)
```

Slicing

```
a[0,:] 
```
row

```
tensor([1., 2., 3.])
```

```
a[0,:].shape
```

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

```
a[:,0] 
```
column

```
tensor([1., 4., 7.])
```

```
a[:,0].shape
```

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

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Concatenation, splitting, stacking, etc. – just like NumPy!

Concatenation

```
torch.cat((a, b))
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.],
        [1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

```
torch.cat((a, b)).shape
```

```
torch.Size([6, 3])
```

```
torch.cat((a, b), dim=1)
```

```
tensor([[1., 2., 3., 1., 0., 0.],
        [4., 5., 6., 0., 1., 0.],
        [7., 8., 9., 0., 0., 1.]])
```

```
torch.cat((a, b), dim=1).shape
```

```
torch.Size([3, 6])
```

dim → axis
Pytorch → Numpy

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

reshape – just like NumPy!

```
a.shape

torch.Size([3, 3])

a.reshape((1,9))

tensor([[1., 2., 3., 4., 5., 6., 7., 8., 9.]])

a.reshape((1,9)).shape

torch.Size([1, 9])
```

squeeze & unsqueeze

```
a_unsqueeze = a.unsqueeze(dim=0)

a_unsqueeze.shape

torch.Size([1, 3, 3])

a_unsqueeze.squeeze_()

tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])

a_unsqueeze.shape

torch.Size([3, 3])
```
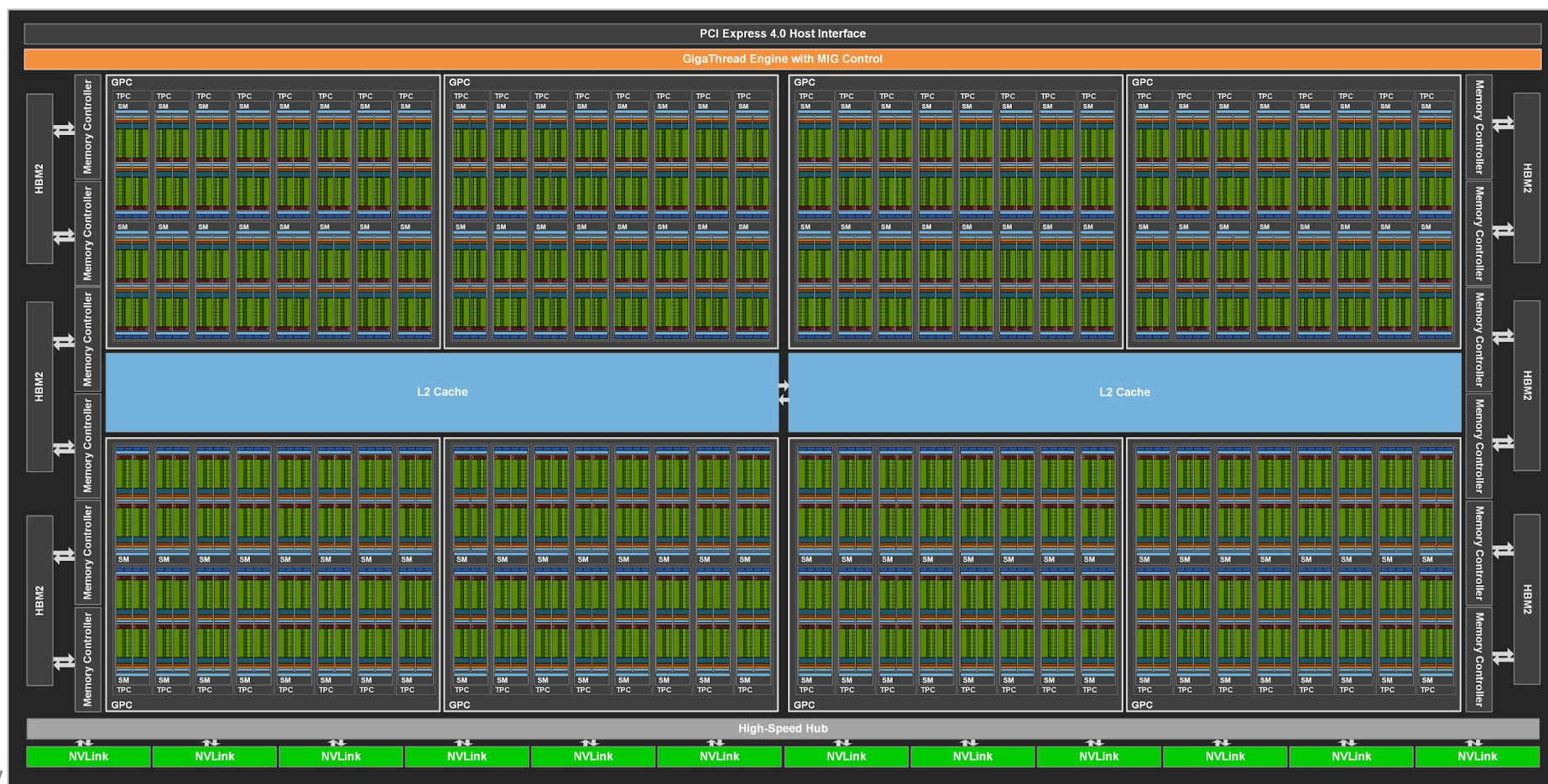
Vectorization & GPU

- GPUs have many small "cores" => many small operations can run in parallel

NVIDIA TESLA A100

Case Study: Matrix Multiplication

Case 1: Nested for loops

```
res = torch.zeros((size, size))
for i i  nge(size):
  for j           ):
    ro
    col = y[:, j]
    res[i, j] = torch.sum(      col)
```

18.8 seconds

Case 3: Vectorized Operation

```
tor
```

25.8 milliseconds

Case 2: Single for loop

```
res =    ch.zeros((size, size))
for i           :
                          dim=0)
    res[i, :]
```

132 milliseconds

Case 4: using a GPU

```
x_g         to('cuda')
y_
torch.mm(x_g
```

0.57 milliseconds

DON'T Do It Yourself!

- Use built-in vectorized operations instead of implementing them yourself!
- Use GPU resources when possible.

Your Implementation

Pytorch's Implementation

Pytorch's Implementation on GPU

Elementwise Operations

- +, -, *, /, min, max, ...

```
c = a + b
d = a - b
...
```

Linear Algebra

- `torch.matmul / @`

```
mat = torch.rand(size=(N, M))        # N×M
vec = torch.rand(size=(M,))          # M
out = mat @ vec                      # N
```

```
mat1 = torch.rand(size=(N, K))       # N×K
mat2 = torch.rand(size=(K, M))       # K×M
out = mat1 @ mat2                     # N×M
```

```
bmat1 = torch.rand(size=(B, N, K))   # B×N×K
bmat2 = torch.rand(size=(B, K, M))   # B×K×M
out = bmat1 @ bmat2                   # B×N×M
```

Pro Tip: Broadcasting

- Fully matching dimensions may be redundant:
  - Add a single row to each row in a matrix.

```
a = torch.rand(size=(N, 1))          # N×1
b = torch.rand(size=(1, M))          # 1×M
out = a + b                          # N×M
```

  - Multiply a batch of matrices by a single matrix.

```
bmat1 = torch.rand(size=(B, N, K))   # B×N×K
mat2 = torch.rand(size=(K, M))       # K×M
out = bmat1 @ mat2                   # B×N×M
```

Advanced Tensor Multiplication

- Einstein summation convention (`torch.einsum`):
  - Each dimension in each operand has a letter.
  - Multiply over dimensions with the same letter.
  - Sum over dimensions which letters are not in the output.

```
mat = torch.rand(size=(N, M))              # N×M
vec = torch.rand(size=(M,))                # M

out = torch.einsum(          , mat, vec)   # N
```

Advanced Tensor Multiplication

- Batch matrix-multiplication with `torch.einsum`:

$$\textbf{out}[b, n, m] = \sum_k \textbf{mat1}[b, n, k] \cdot \textbf{mat2}[b, k, m]$$

```
mat1 = torch.rand(size=(B, N, K))                      # B×N×K
mat2 = torch.rand(size=(B, K, M))                      # B×K×M

out = torch.einsum(                    , mat1, mat2)   # B×N×M
```
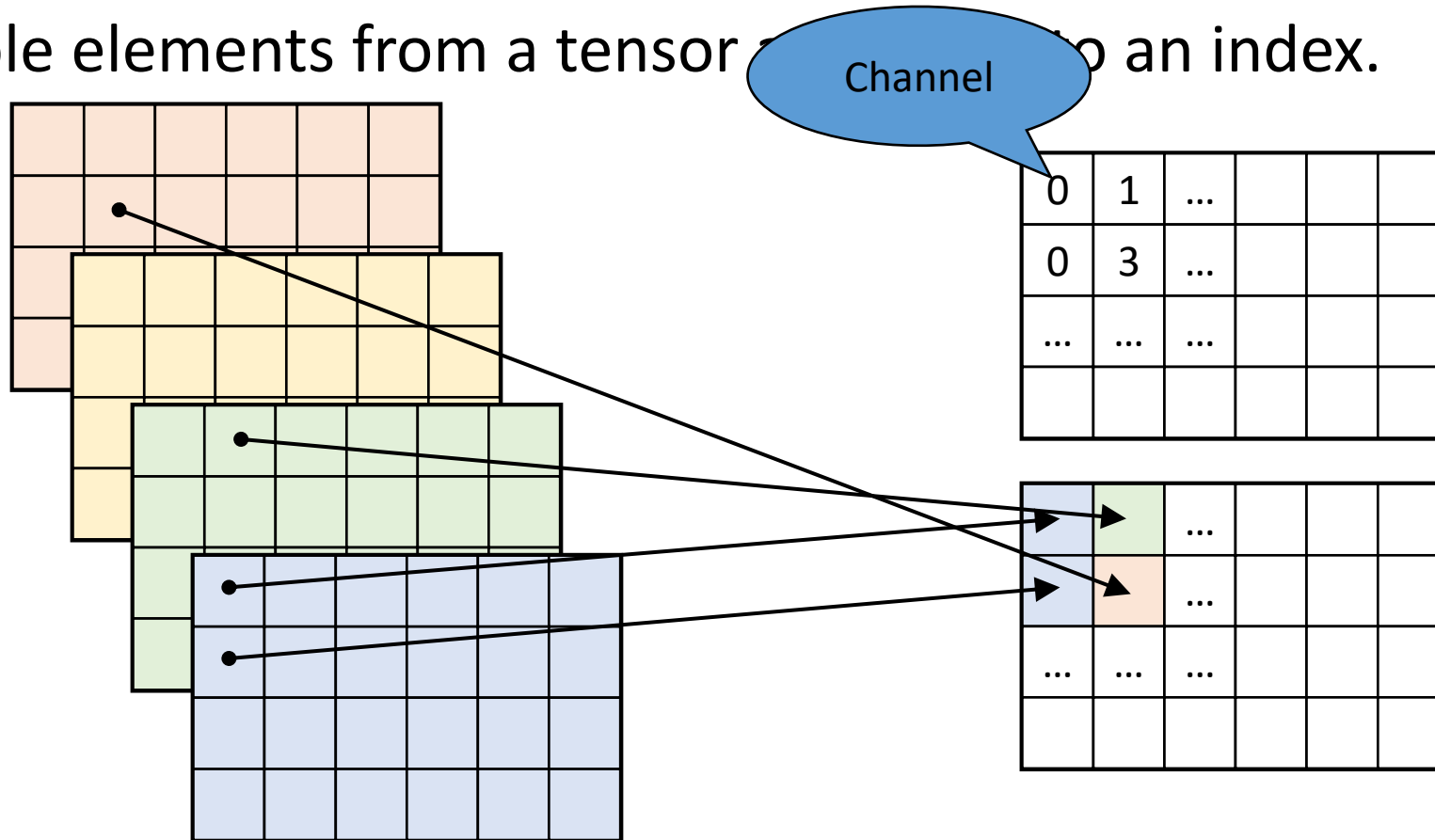
Advanced Tensor Multiplication

- More complex example:

$$\mathbf{out}[i,j,k] = \sum_{l}\sum_{m} \mathbf{x}[i,k,m,l,j] \cdot \mathbf{y}[i,l,j,k] \cdot \mathbf{z}[i]$$

```
x = torch.rand(size=(I, K, M, L, J))
y = torch.rand(size=(I, L, J, K))
z = torch.rand(size=(I,))

out = torch.einsum("ikmlj,iljk,i->ijk", x, y, z)
```

Gather

- Sample elements from a tensor according to an index.

Gather

- Sample elements from a tensor according to an index.

```
dim = 0
src = torch.rand(size=(4, 4, 6)) # C, H, W
index = torch.randint(low=0, high=4, size=(1, 4, 6))
#index = torch.randint(low=0, high=src.size(dim), size=(num_samples, src.size(1), src.size(2))

# naive
out = src.zeros_like(size=(1, 4, 6))
for i in range(1):
  for j in range(4):
    for k in range(6):
      out[i, j, k] = src[index[i, j, k], j, k]

# vectorized
out = torch.gather(src, dim, index)
```
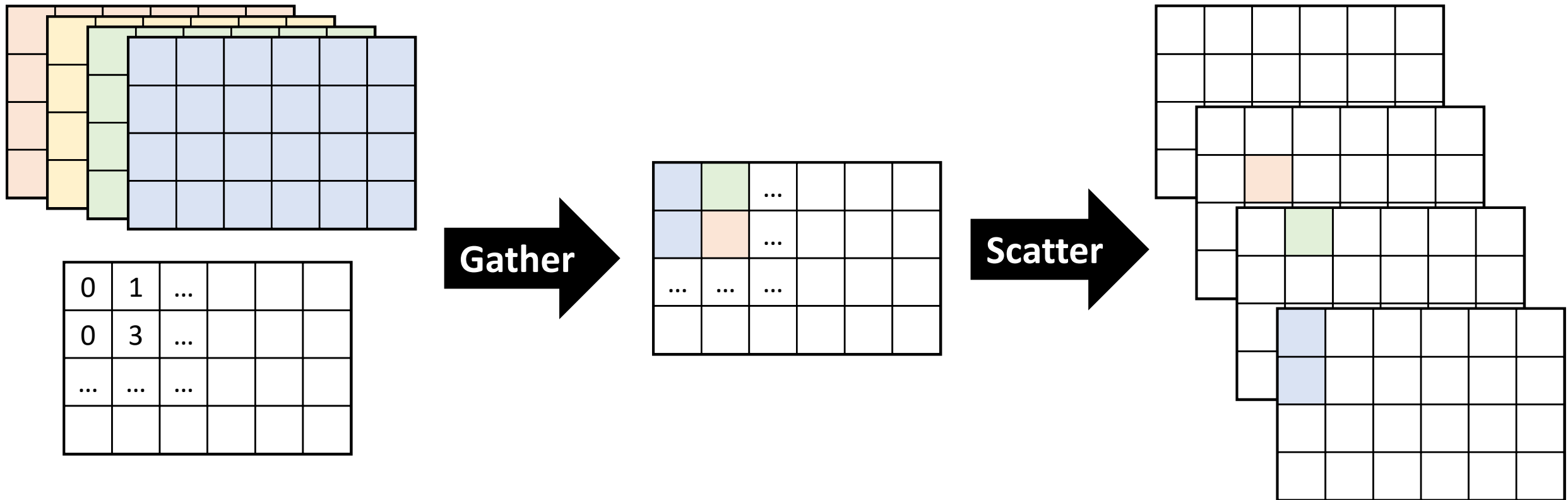
Gather

- Sample elements from a tensor according to an index.

```
dim = 1
src = torch.rand(size=(4, 4, 6))
index = torch.randint(low=0, high=4, size=(4, 1, 6))
#index = torch.randint(low=0, high=src.size(dim), size=(src.size(0), num_samples, src.size(2))

# naive
out = src.zeros_like(size=(4, 1, 6))
for i in range(4):
  for j in range(1):
    for k in range(6):
      out[i, j, k] = src[i, index[i, j, k], k]

# vectorized
out = torch.gather(src, dim, index)
```
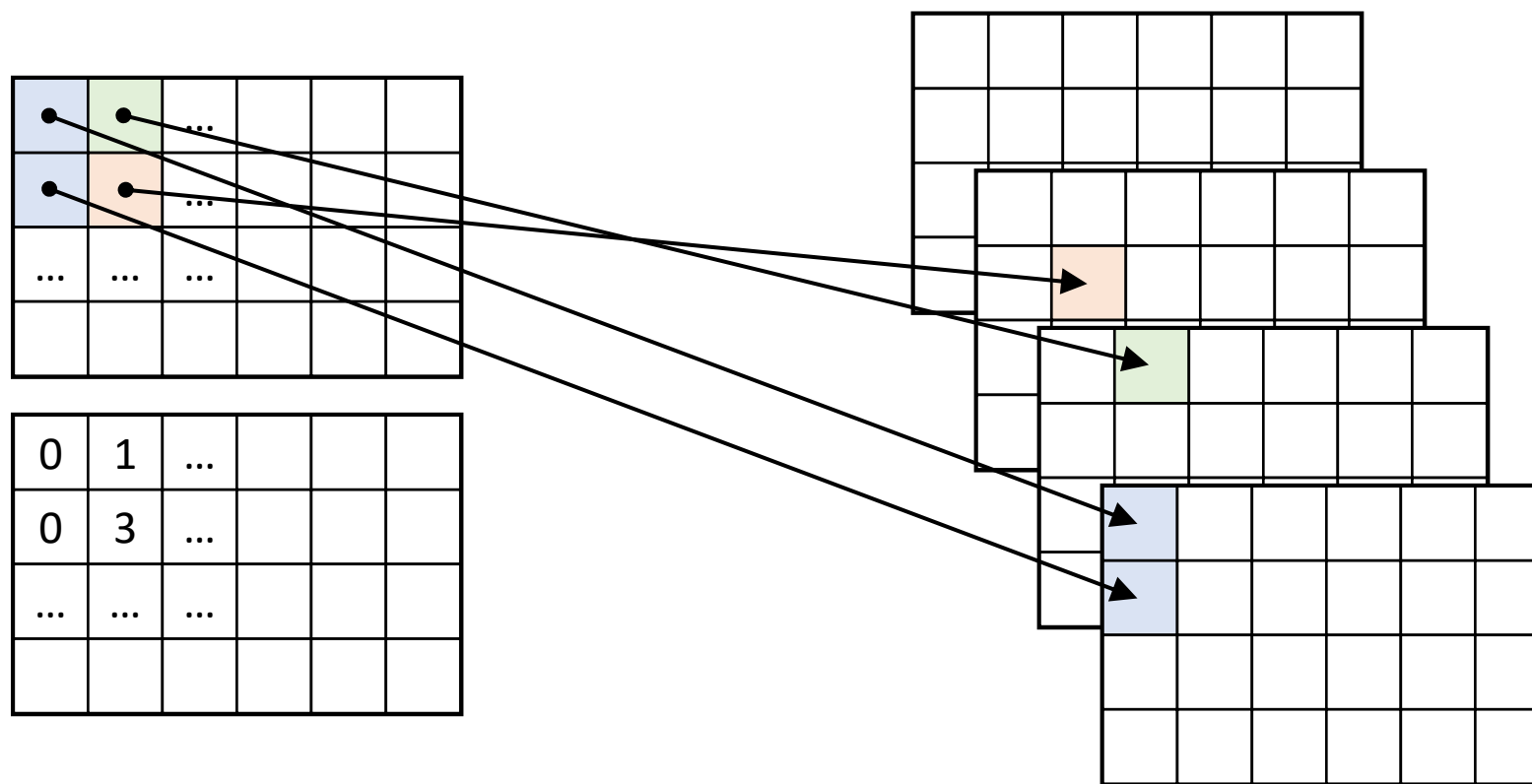
Gather and Scatter

Scatter (Add)

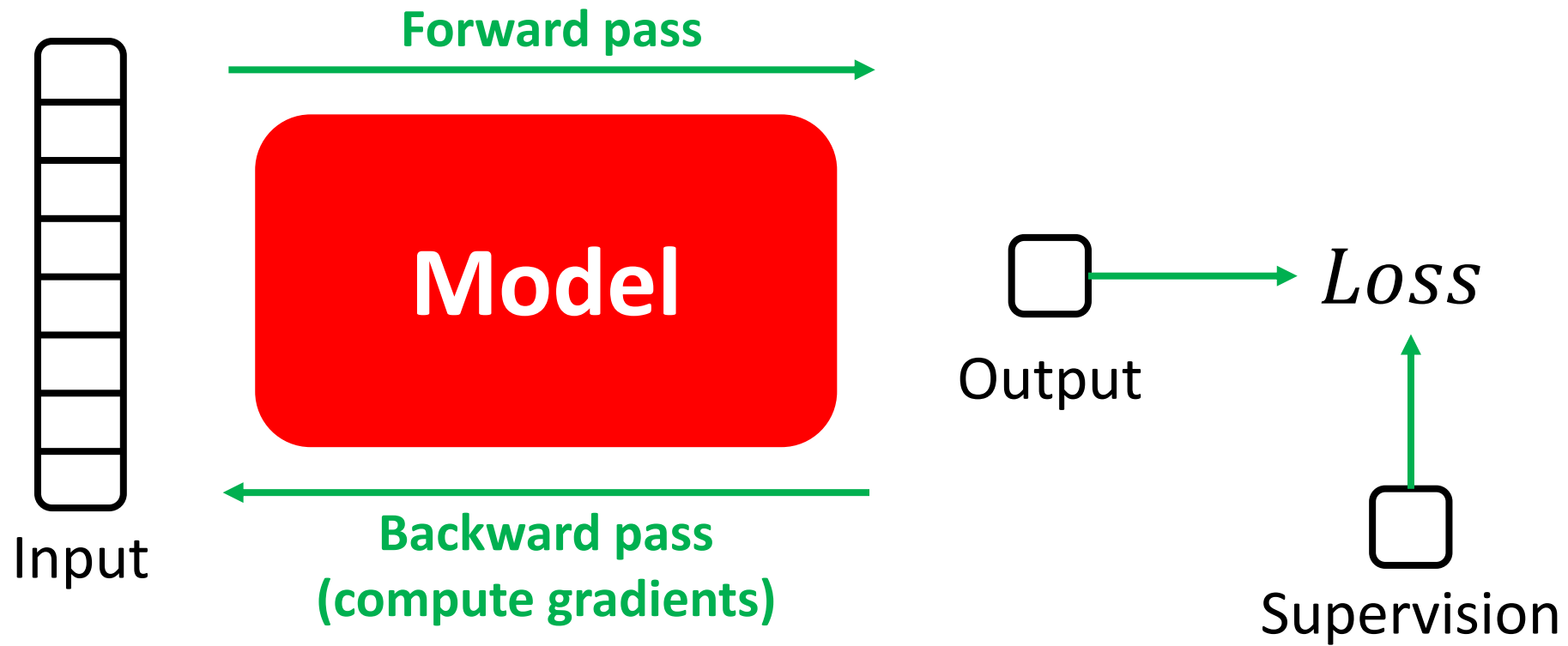- The opposite (backward) of *Gather*.

Scatter (Add)

- The opposite (backward) of *Gather*.

```python
dim = 0
src = torch.rand(size=(1, 4, 6))
index = torch.randint(low=0, high=4, size=(1, 4, 6))
#index = torch.randint(low=0, high=size, size=src.size())

# naive
out = src.zeros_like(size=(4, 4, 6))
for i in range(1):
  for j in range(4):
    for k in range(6):
      out[index[i, j, k], j, k] += src[i, j, k]

# vectorized in-place (assume out.shape == (4, 4, 6))
out.scatter_add_(dim, index, src)
```

Building a Neural Network

Forward pass

Model

Output

Loss

Backward pass
(compute gradients)

Input

Supervision

Building a Neural Network

nn.Module

__init__()

forward()

Building a Neural Network

```
INPUT_SIZE = 224
C = 10 # num classes


class MLP(nn.Module):

  def __init__(self):
    super(MLP, self).__init__()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(INPUT_SIZE*INPUT_SIZE, 256)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(256, 512)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(512, C)


  def forward(self, x):
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu1(x)
    x = self.fc2(x)
    x = self.relu2(x)
    x = self.fc3(x)
    return x
```

Initialize

```
B = 10 # batch size

input = torch.randn(B, 1, INPUT_SIZE, INPUT_SIZE)
model = MLP()
```

input shape: $(B, 1, H, W)$

output shape: (B, C)

Run with CPU

```
output = model(input)
```

7.96 milliseconds

Run with GPU

```
gpu_model = model.to('cuda')
gpu_input = input.to('cuda')

gpu_output = gpu_model(gpu_input)
```

0.58 milliseconds

Based on Pytorch tutorials

Building a Neural Network

```python
INPUT_SIZE = 224
C = 10 # num classes

class MLP(nn.Module):

  def __init__(self):
    super(MLP, self).__init__()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(INPUT_SIZE*INPUT_SIZE, 256)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(256, 512)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(512, C)

  def forward(self, x):
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu1(x)
    x = self.fc2(x)
    x = self.relu2(x)
    x = self.fc3(x)
    return x
```
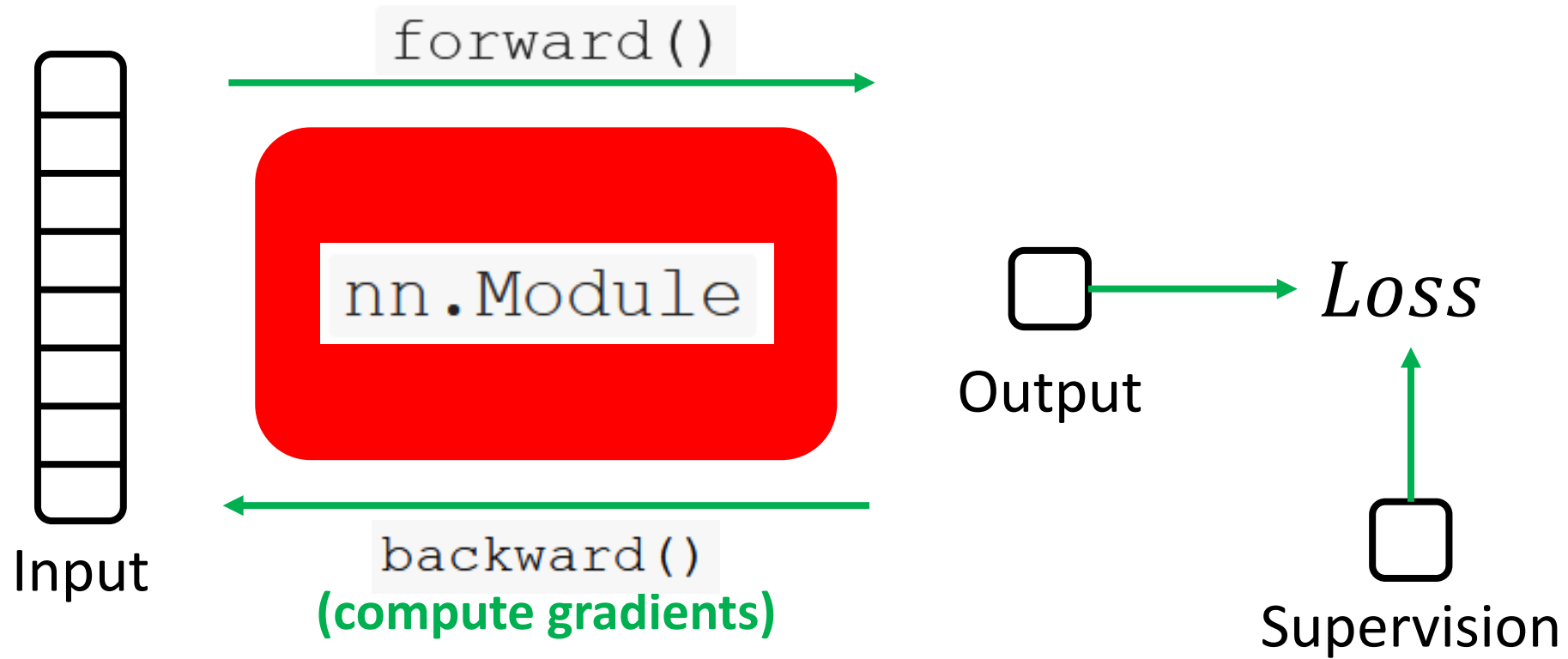
Using Sequential

```python
seq_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(INPUT_SIZE*INPUT_SIZE, 256),
    nn.ReLU(),
    nn.Linear(256, 512),
    nn.ReLU(),
    nn.Linear(512, 10),
)

output = seq_model(input)
```
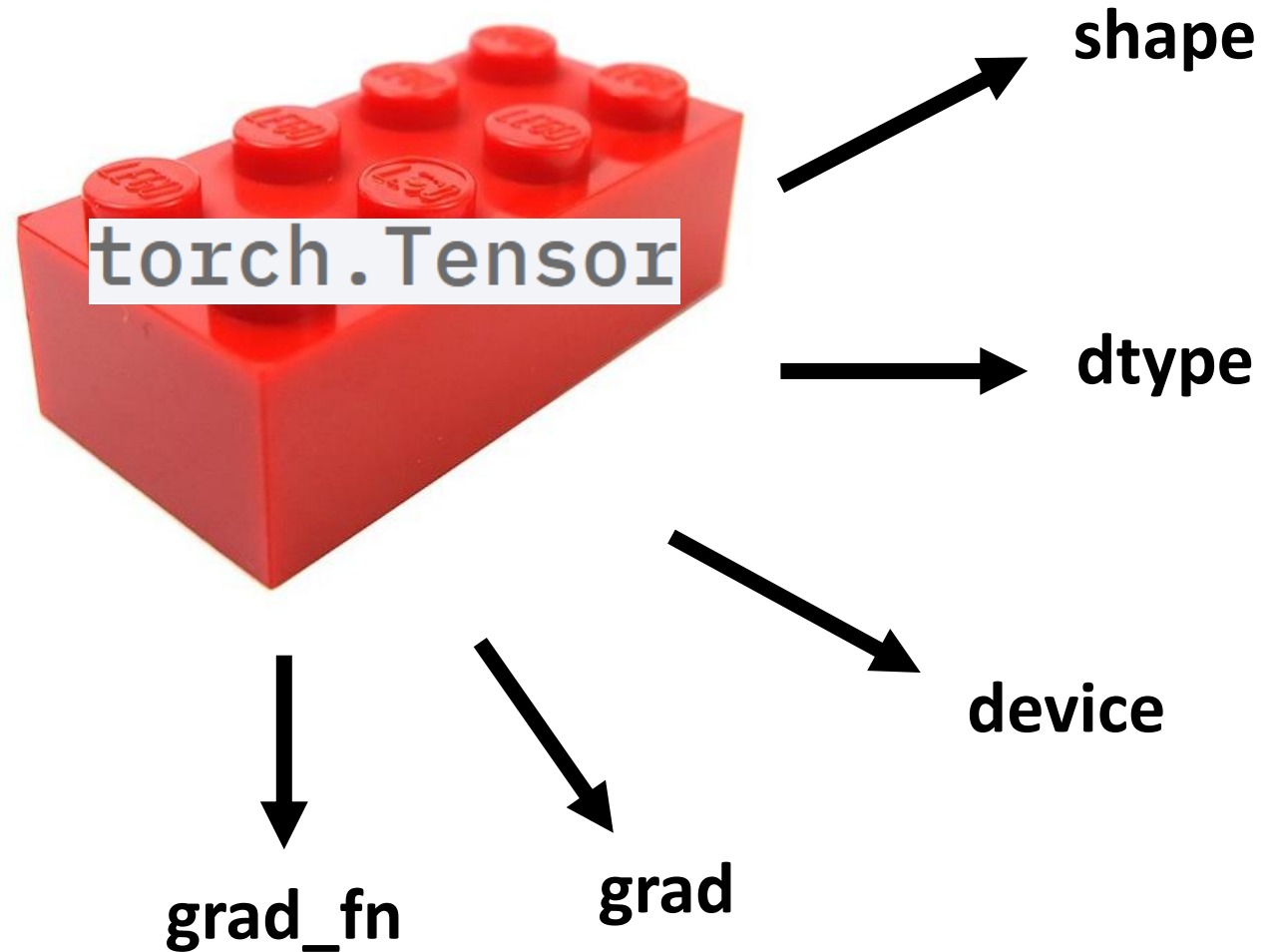
DL4CV@Weizmann Based on Pytorch tutorials

Backpropagation

forward()

nn.Module

Input

backward()
(compute gradients)

Output

Loss

Supervision

Reminder Example

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

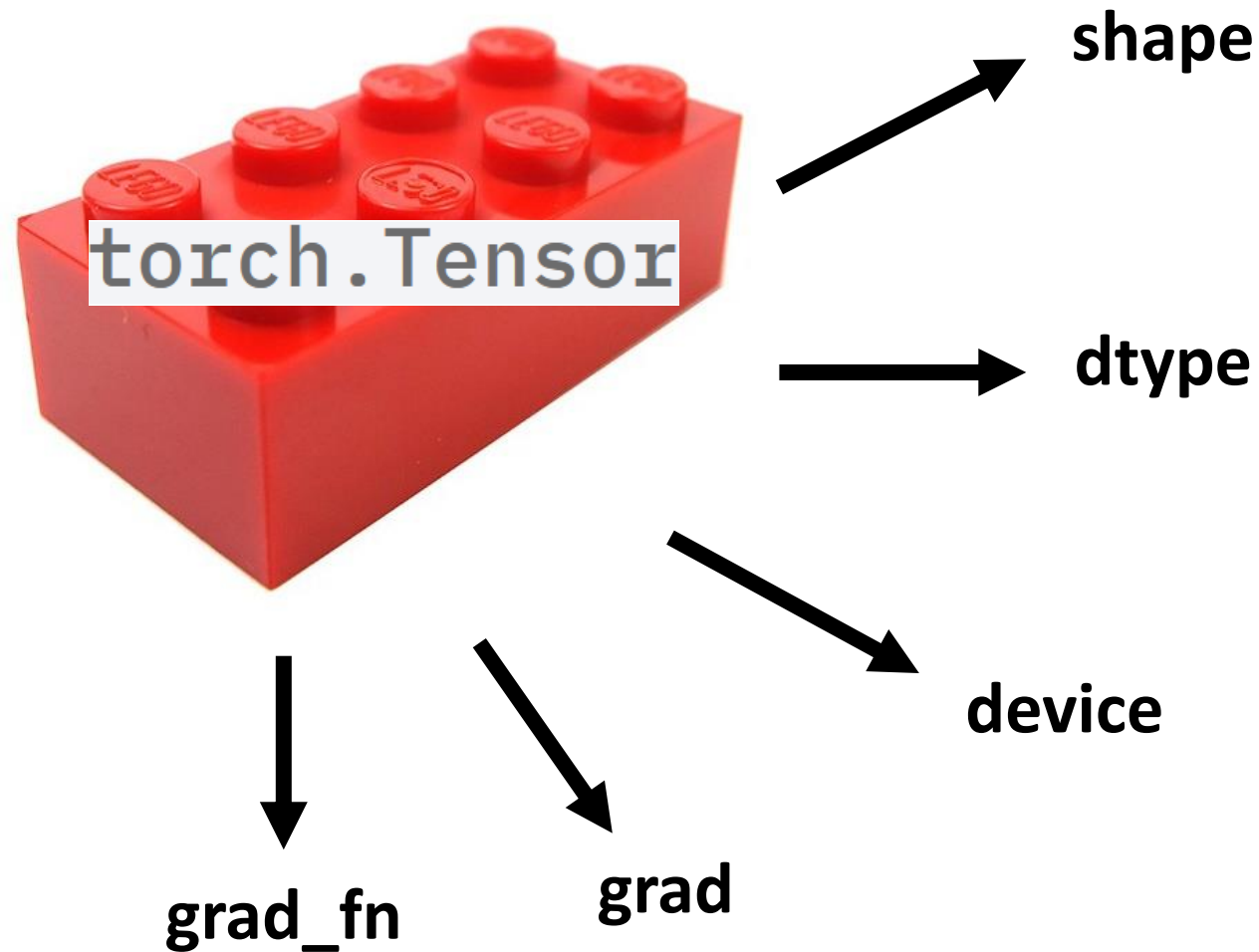- SGD: $w \leftarrow w - \alpha \cdot \boxed{\dfrac{\partial L}{\partial w}}$

Tensor Attributes - Autograd

torch.Tensor

shape

dtype

device

grad_fn

grad

grad - gradient for this tensor

grad_fn – reference for a function, used to obtain this tensor

Tensor Attributes - Autograd

shape

dtype

device

grad

grad_fn

torch.Tensor

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()

print(w.grad)
print(x.grad)
```
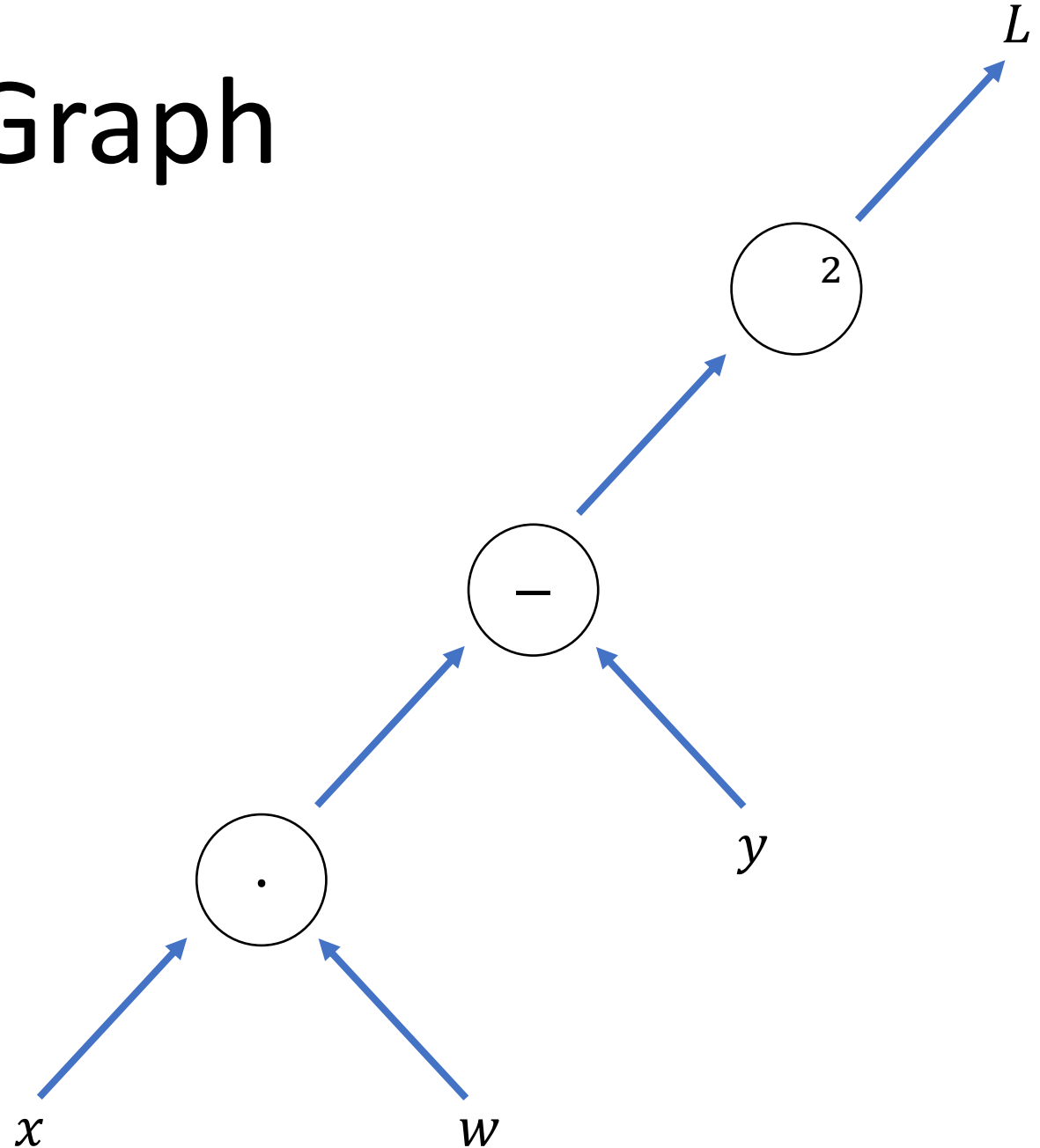
```
tensor([-4.])
None
```

Computational Graph

- Directed graph where the nodes correspond to operations or variables

- Can be used to differentiate via the *chain rule*

Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \frac{\partial L}{\partial w}$
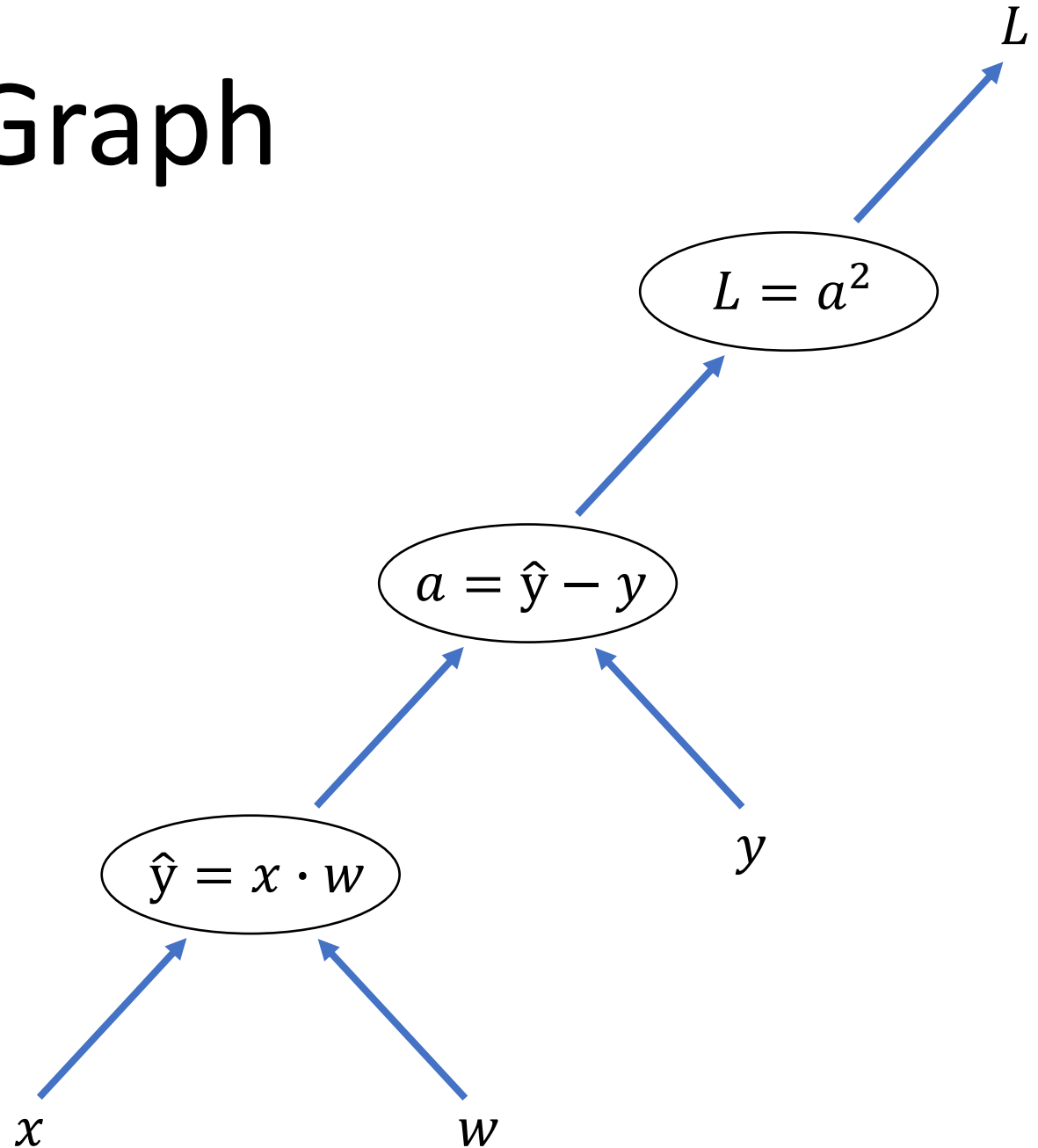
Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

L

$L = a^2$

$\dfrac{\partial L}{\partial a} = 2a$

$a = \hat{y} - y$

$\dfrac{\partial a}{\partial \hat{y}} = 1$

$\dfrac{\partial a}{\partial y} = -1$

$\hat{y} = x \cdot w$

y

$\dfrac{\partial \hat{y}}{\partial x} = w$

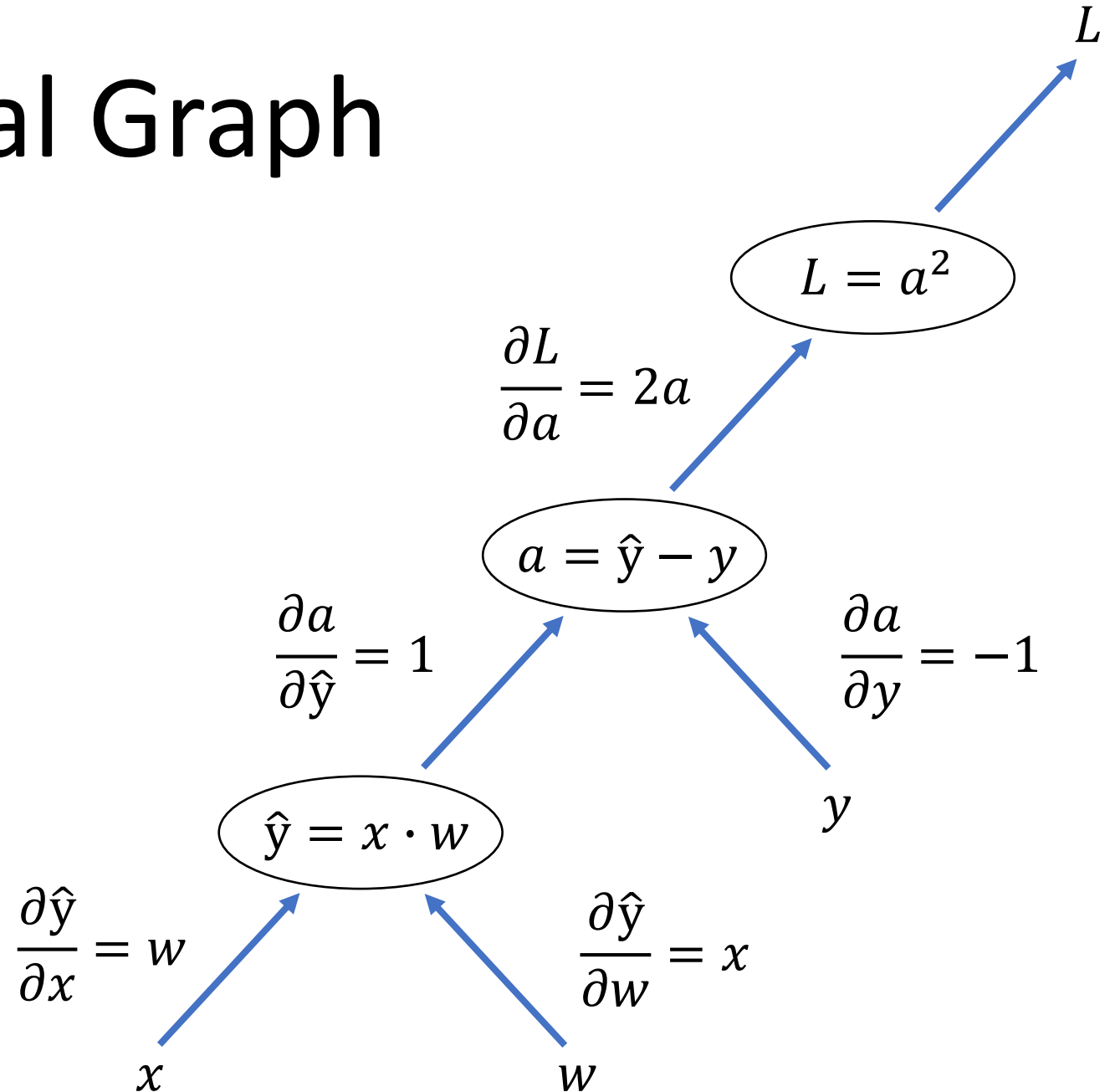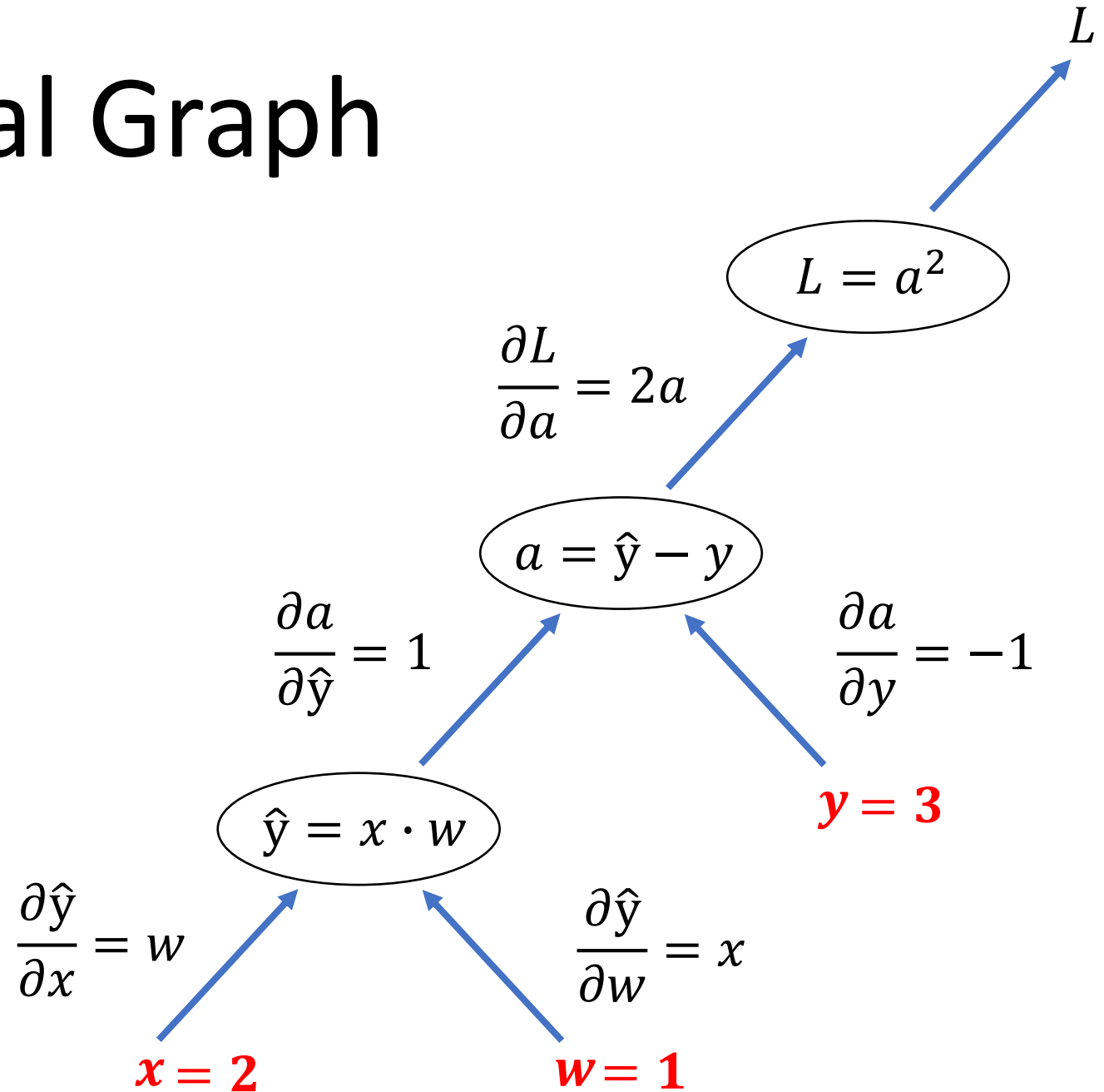$\dfrac{\partial \hat{y}}{\partial w} = x$

x

w

Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

L

$L = a^2$

$\dfrac{\partial L}{\partial a} = 2a$

$a = \hat{y} - y$

$\dfrac{\partial a}{\partial \hat{y}} = 1$ $\qquad \dfrac{\partial a}{\partial y} = -1$

$\hat{y} = x \cdot w$ $\qquad y = 3$

$\dfrac{\partial \hat{y}}{\partial x} = w$ $\qquad \dfrac{\partial \hat{y}}{\partial w} = x$
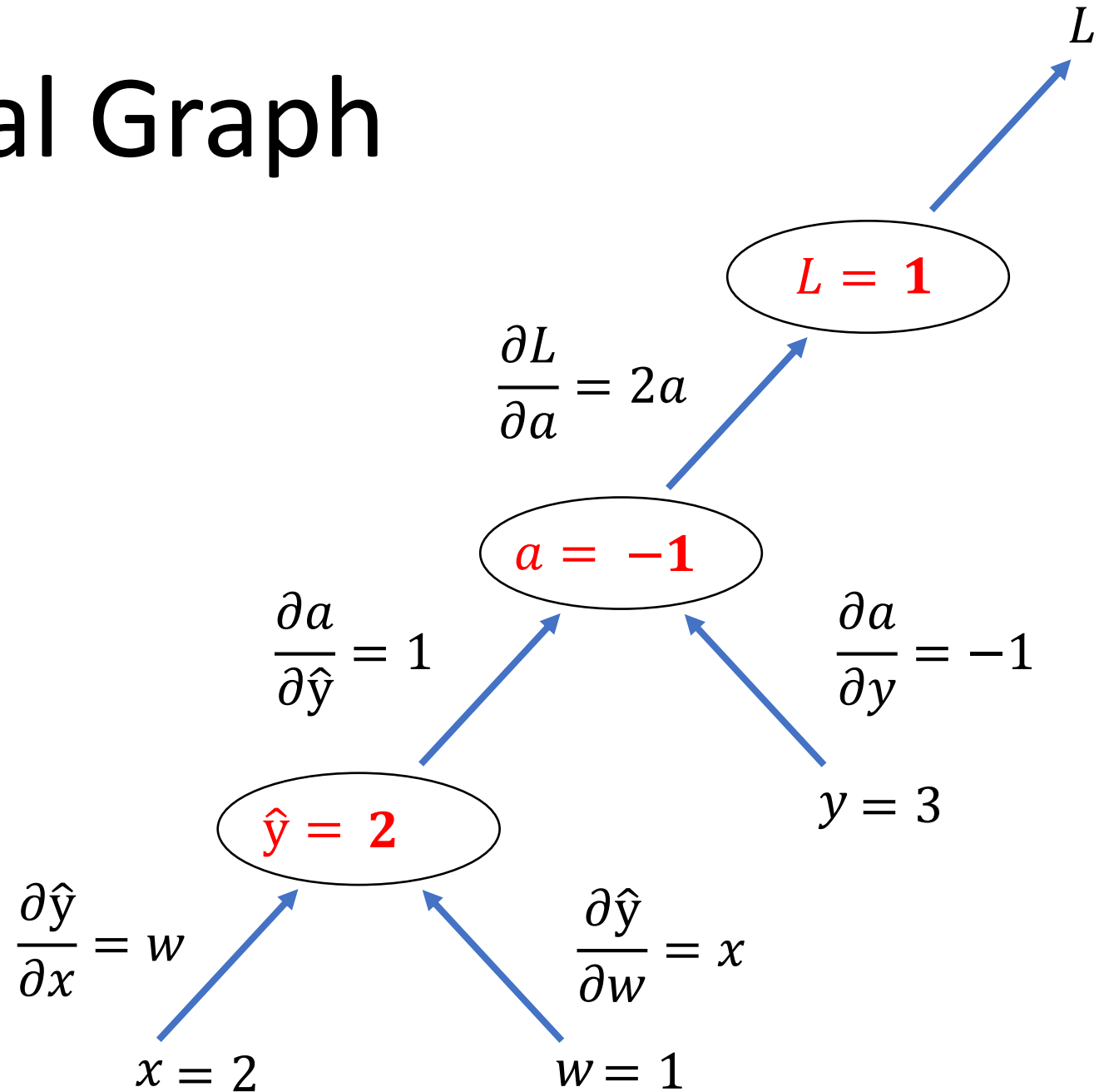
$x = 2$ $\qquad w = 1$

Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$
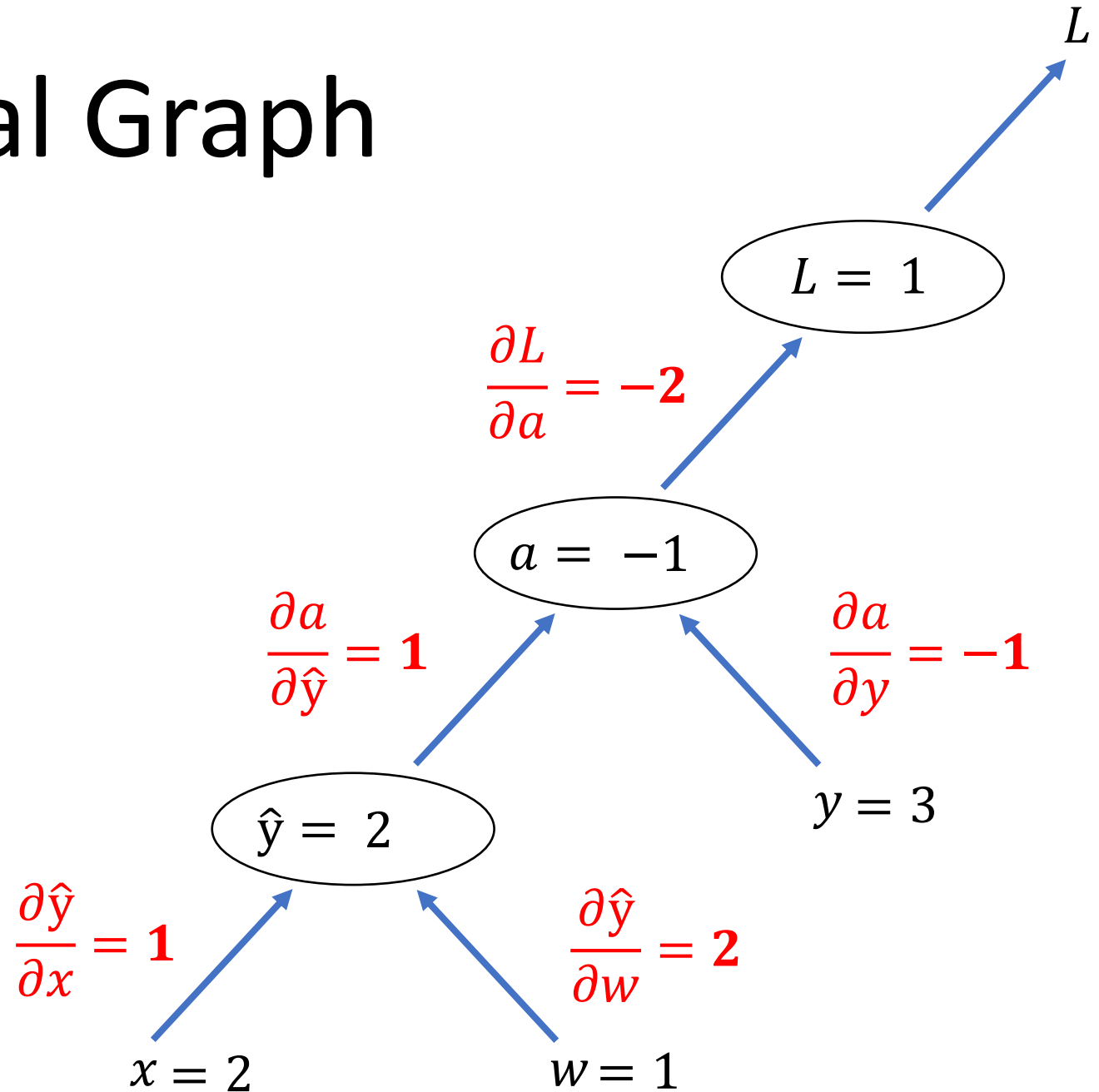
Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

$$L$$

$$L = 1$$

$$\frac{\partial L}{\partial a} = -2$$

$$a = -1$$

$$\frac{\partial a}{\partial \hat{y}} = 1 \qquad \frac{\partial a}{\partial y} = -1$$

$$\hat{y} = 2 \qquad y = 3$$

$$\frac{\partial \hat{y}}{\partial x} = 1 \qquad \frac{\partial \hat{y}}{\partial w} = 2$$
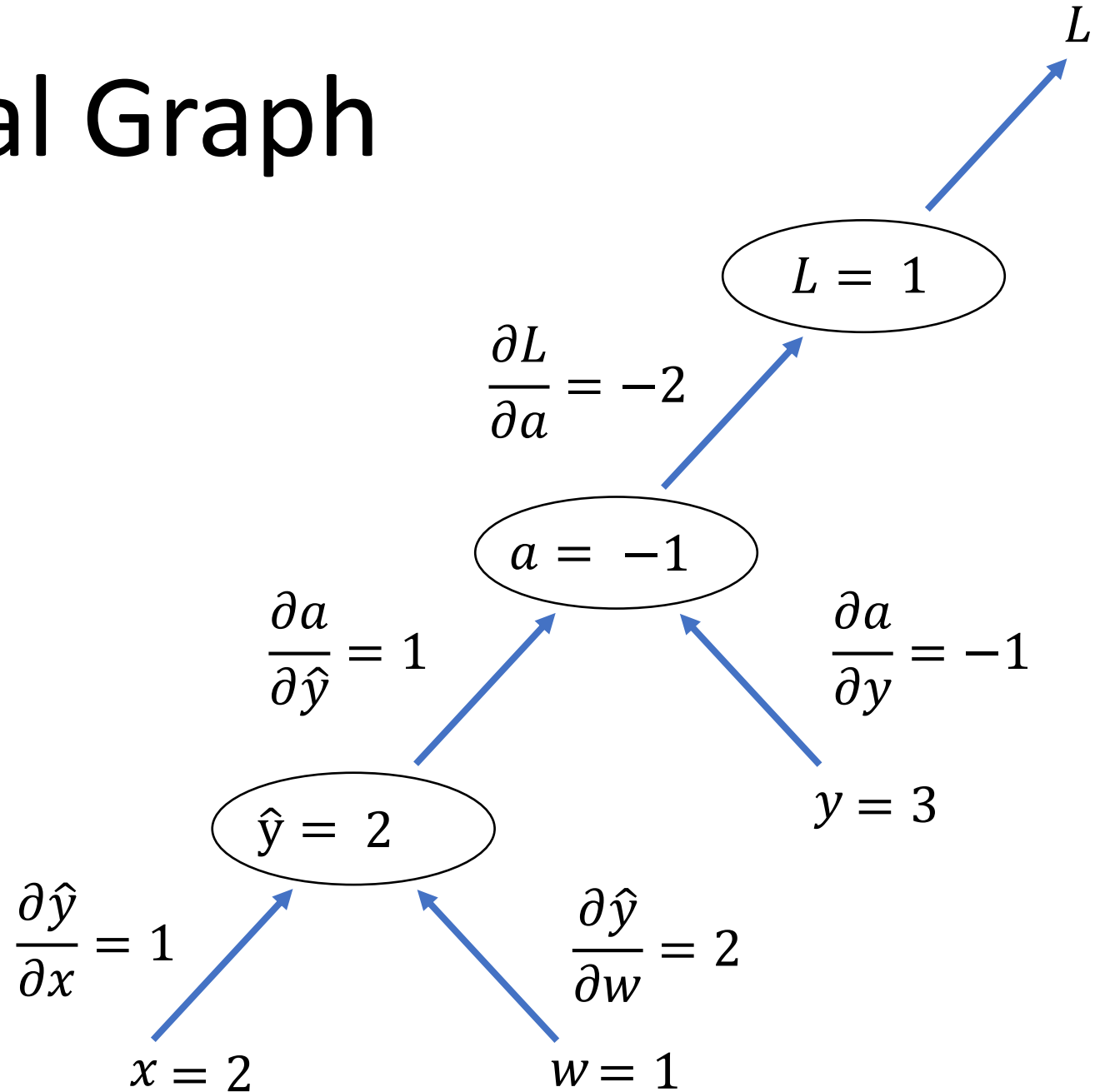
$$x = 2 \qquad w = 1$$

Computational Graph

- Data: $x, y \in \mathbb{R}$

- Hypothesis: $\hat{y} = x \cdot w$

- Loss: $L = (\hat{y} - y)^2$

- SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

Chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$
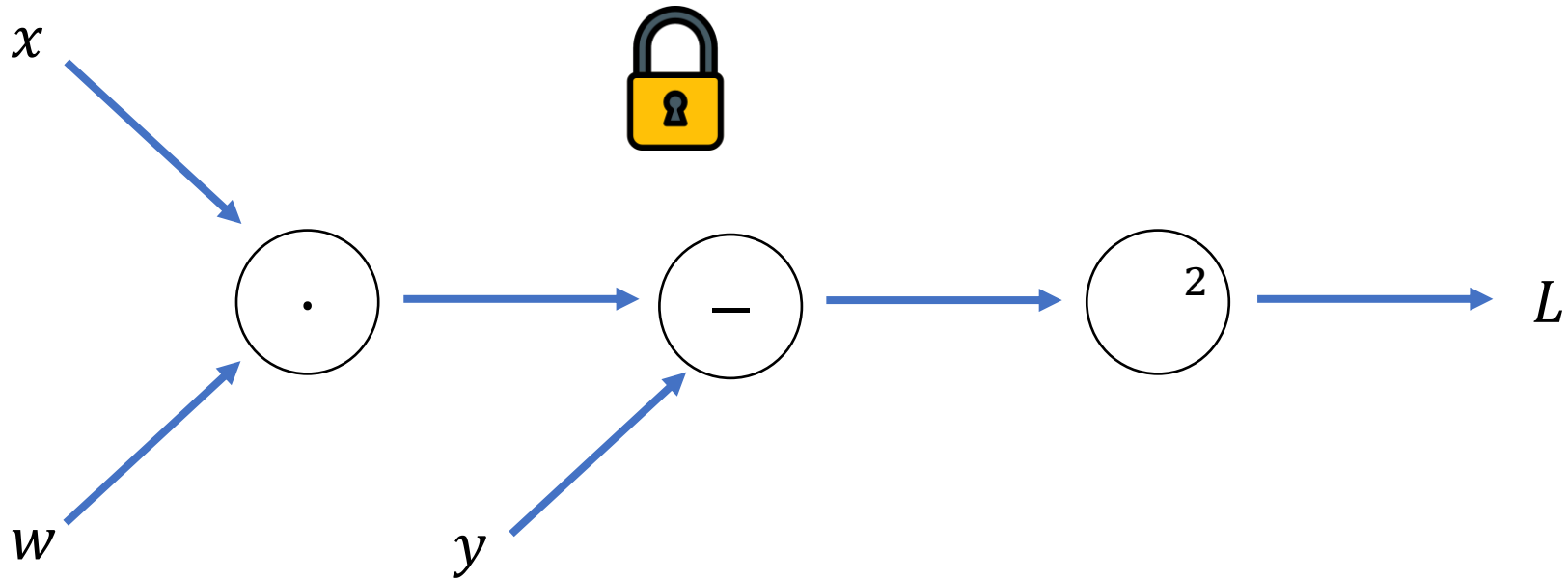
L

$L = 1$

$\dfrac{\partial L}{\partial a} = -2$

$a = -1$

$\dfrac{\partial a}{\partial \hat{y}} = 1$

$\dfrac{\partial a}{\partial y} = -1$

$\hat{y} = 2$

$y = 3$

$\dfrac{\partial \hat{y}}{\partial x} = 1$

$\dfrac{\partial \hat{y}}{\partial w} = 2$

$x = 2$

$w = 1$

Static VS Dynamic Graph

- Static graph: graph's structure is fixed, then inject data
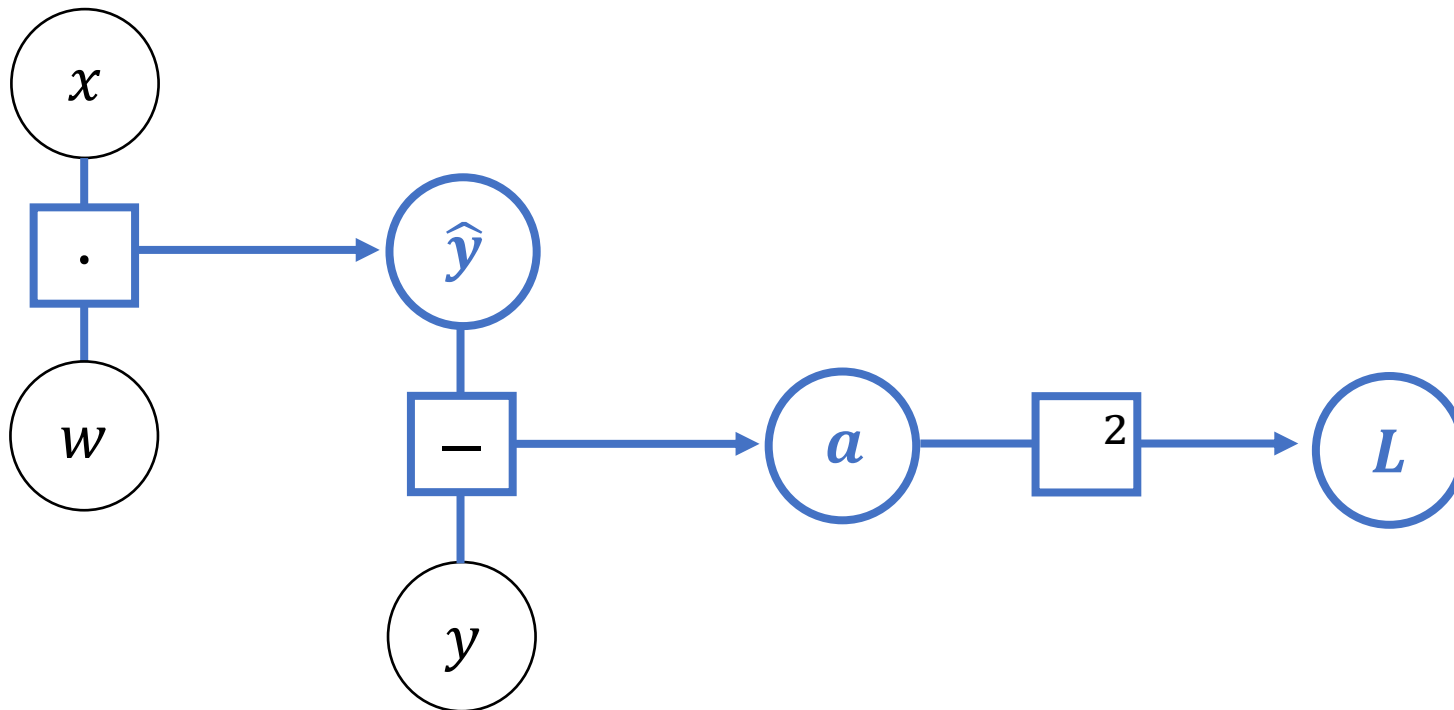- Dynamic graph: graph's structure is defined on the fly

Static Graph

- First define the graph, then inject data

x

w

y

L

2

Dynamic Graph

- Define variables and their relations. The graph is defined implicitly

Static VS Dynamic Graph

Static	Dynamic
Define-and-run	Define-by-run
Slightly Faster	Flexible

Computational Graph in PyTorch

- Dynamic

- Created in the forward pass (on the fly)
  - Unless torch.no_grad() was used

- Deleted in the backward pass

CG in PyTorch: forward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```

y=3
y.grad=None

x=2
x.grad=None

w=1
w.grad=None

CG in PyTorch: forward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```
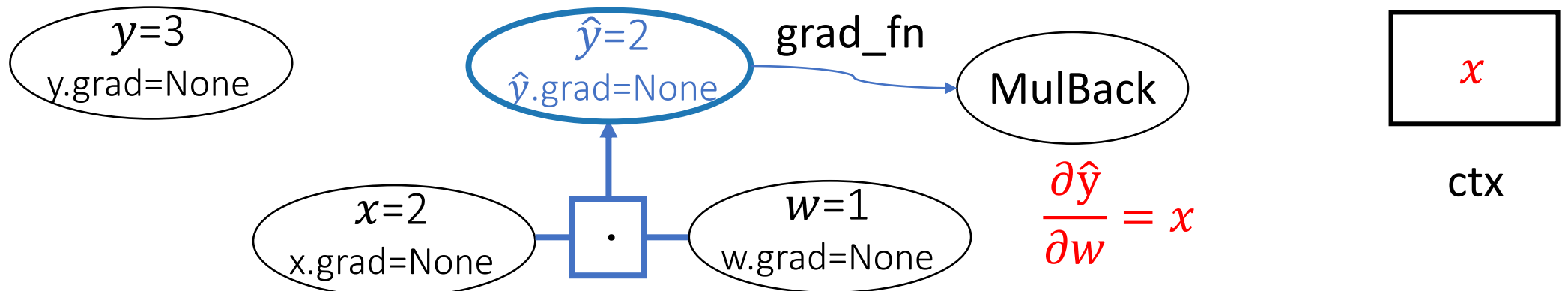
y=3
y.grad=None

$\hat{y}$=2
$\hat{y}$.grad=None

grad_fn

MulBack

x

ctx

x=2
x.grad=None

·

w=1
w.grad=None

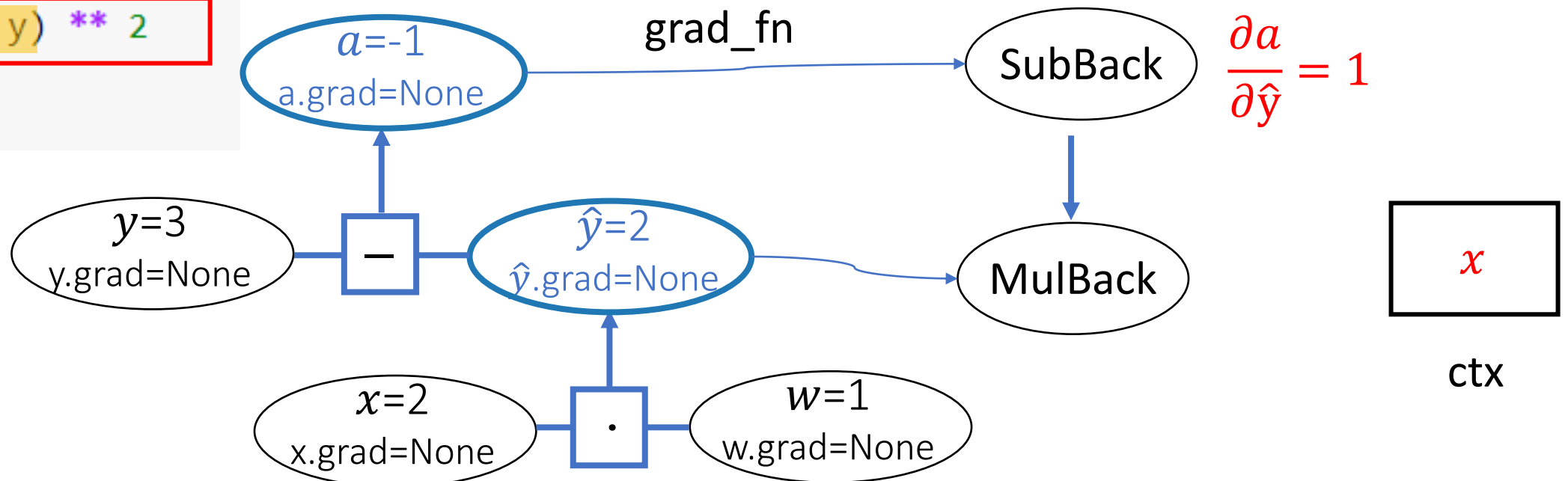$$\frac{\partial \hat{y}}{\partial w} = x$$

CG in PyTorch: forward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```


a=-1
a.grad=None

grad_fn

SubBack

$\dfrac{\partial a}{\partial \hat{y}} = 1$

y=3
y.grad=None

$\hat{y}$=2
$\hat{y}$.grad=None

MulBack

x=2
x.grad=None

w=1
w.grad=None

x

ctx

CG in PyTorch: forward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```
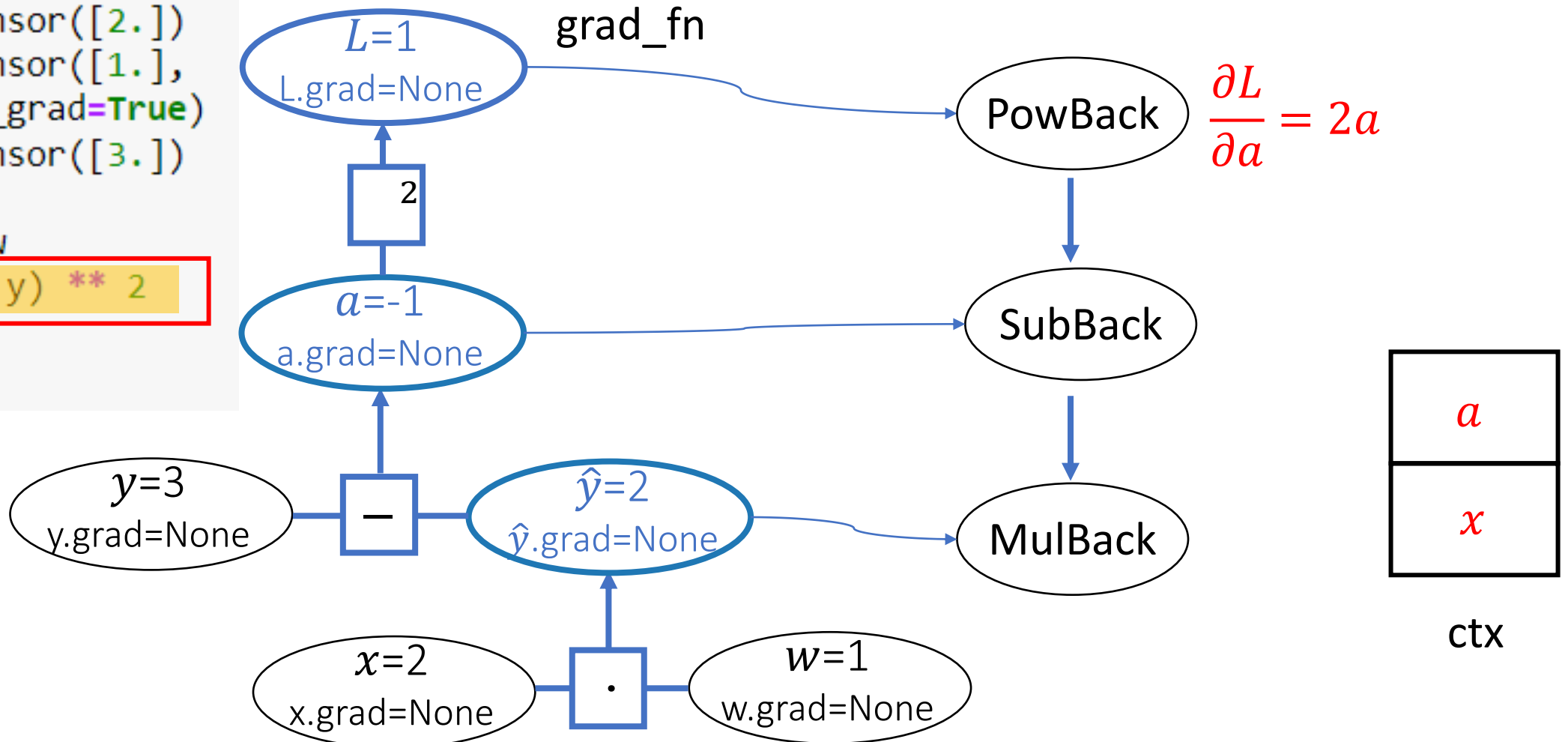

$L=1$
L.grad=None

grad_fn

PowBack

$$\frac{\partial L}{\partial a} = 2a$$

$a=-1$
a.grad=None

SubBack

$\hat{y}=2$
$\hat{y}$.grad=None

MulBack

$y=3$
y.grad=None

$x=2$
x.grad=None

$w=1$
w.grad=None

a

x

ctx

DL4CV Weizmann

CG in PyTorch: backward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```

CG in PyTorch: backward


```
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```
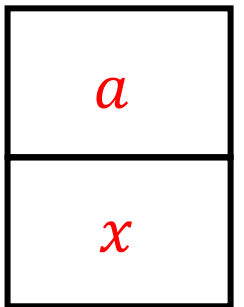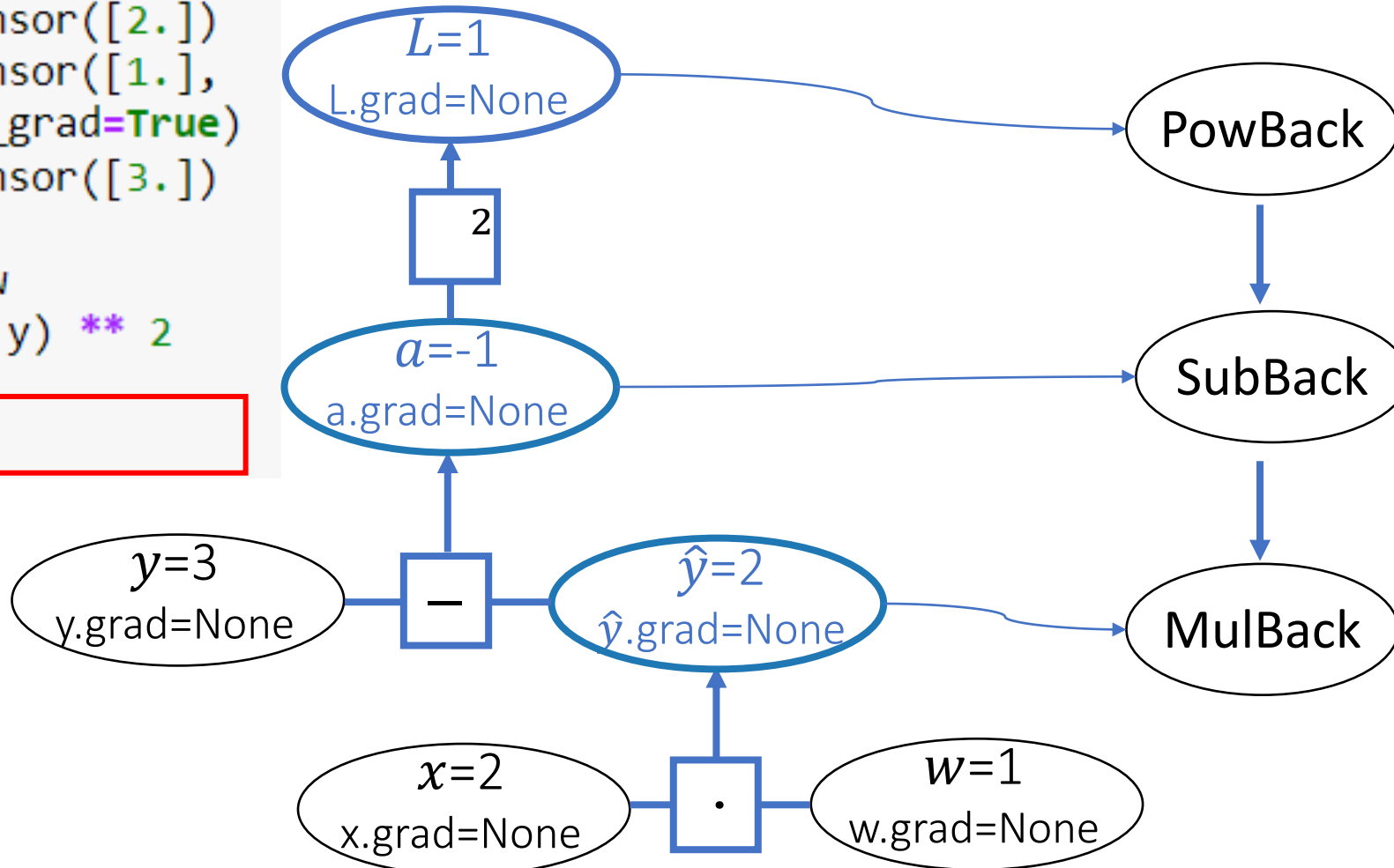
$L=1$
L.grad=None

2

$a=-1$
a.grad=None

$y=3$
y.grad=None

$\hat{y}=2$
$\hat{y}$.grad=None

$x=2$
x.grad=None

$w=1$
w.grad=None

PowBack

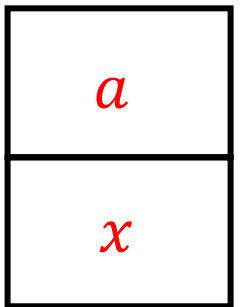$\dfrac{\partial L}{\partial a}$

SubBack

MulBack

a

x

ctx

CG in PyTorch: backward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```

$L=1$
L.grad=None

$a=-1$
a.grad=-2

$y=3$
y.grad=None

$\hat{y}=2$
$\hat{y}$.grad=None

$x=2$
x.grad=None

$w=1$
w.grad=None

PowBack $\quad \dfrac{\partial L}{\partial a} = 2a$

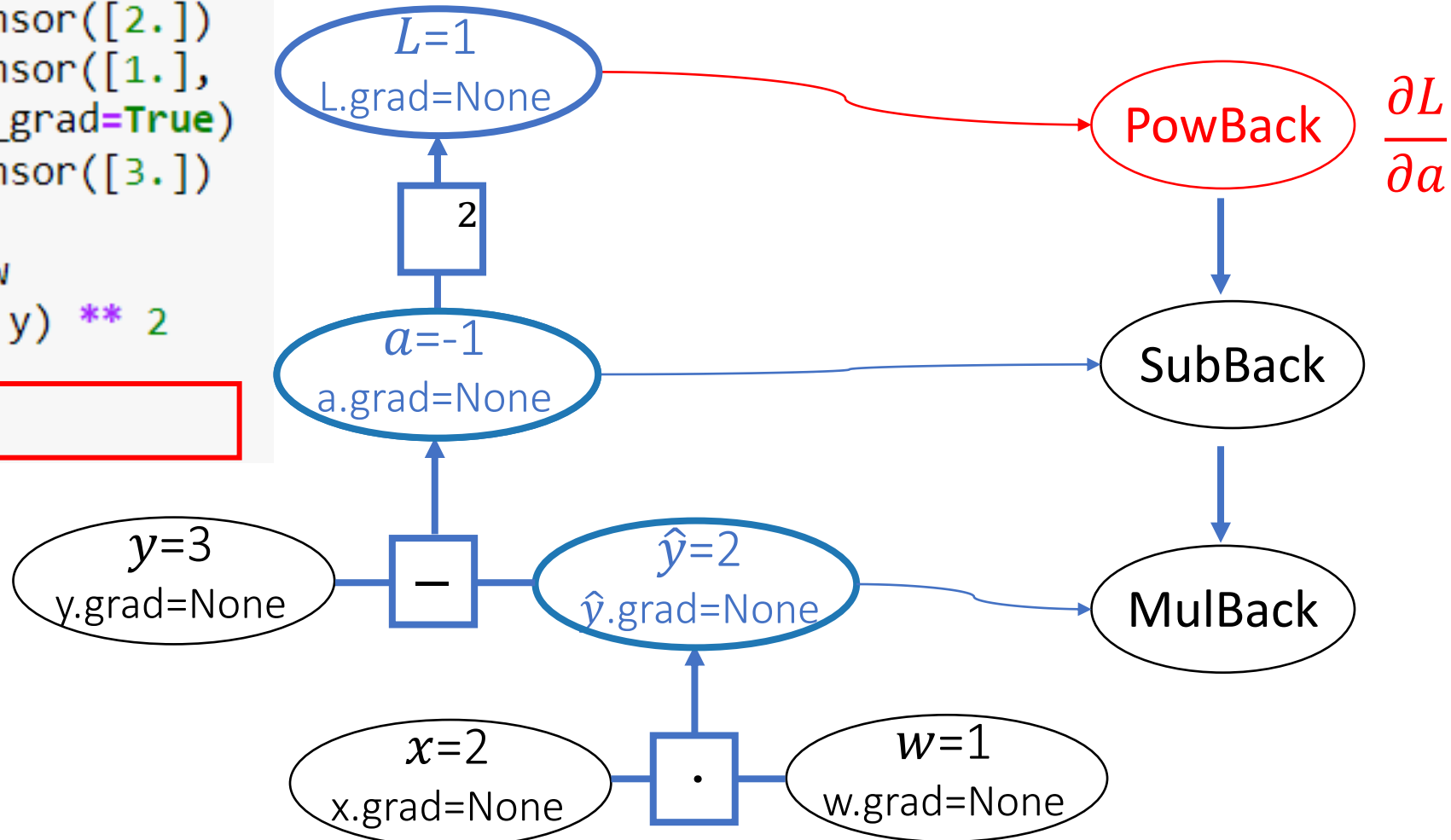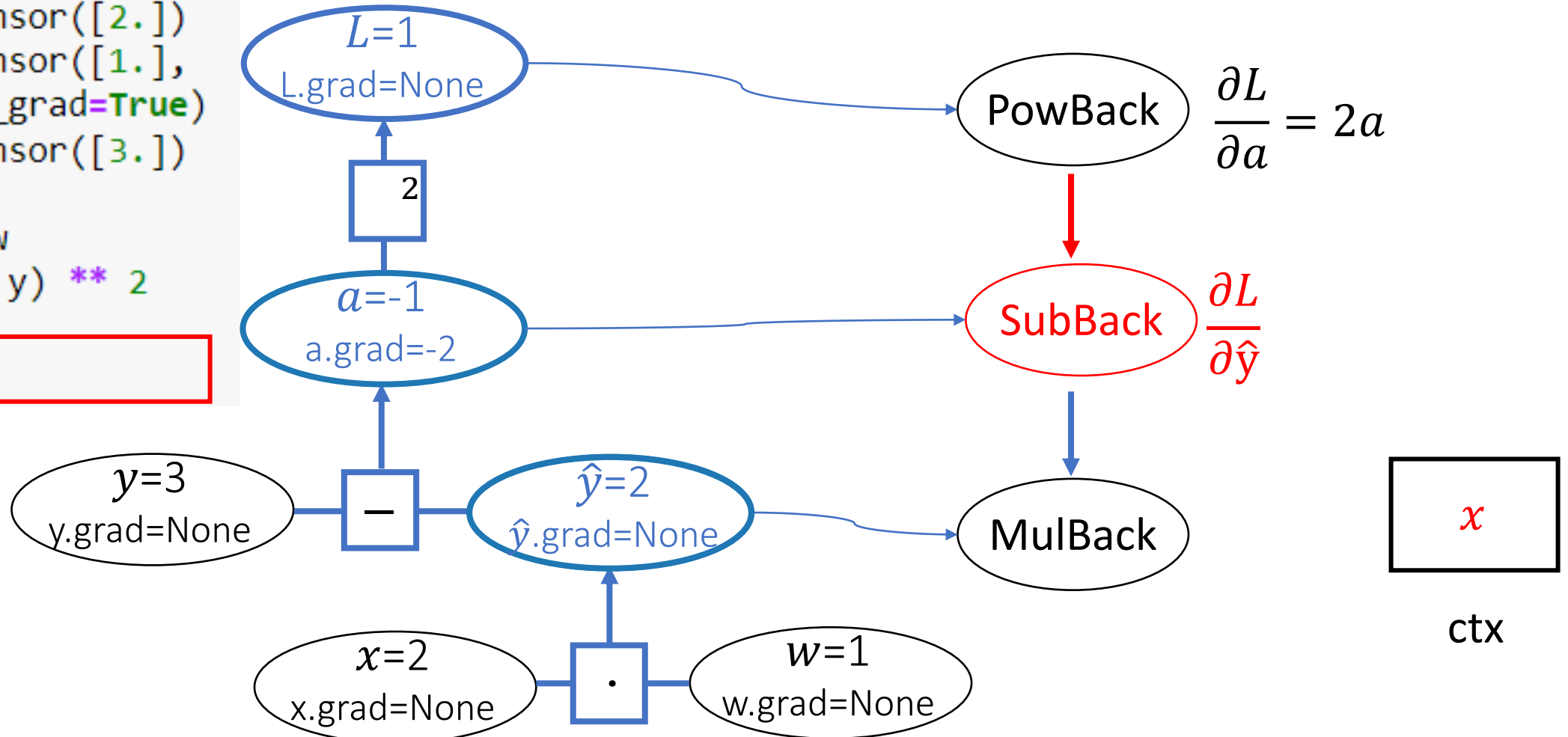SubBack $\dfrac{\partial L}{\partial \hat{y}}$

MulBack

x

ctx

CG in PyTorch: backward

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2
```

L.backward()

L=1
L.grad=None

a=-1
a.grad=-2

y=3
y.grad=None

$\hat{y}$=2
$\hat{y}$.grad=-2

x=2
x.grad=None

w=1
w.grad=None

PowBack $\quad \dfrac{\partial L}{\partial a} = 2a$

SubBack $\quad \dfrac{\partial L}{\partial \hat{y}} = \dfrac{\partial L}{\partial a}\dfrac{\partial a}{\partial \hat{y}} = 2a \cdot 1$

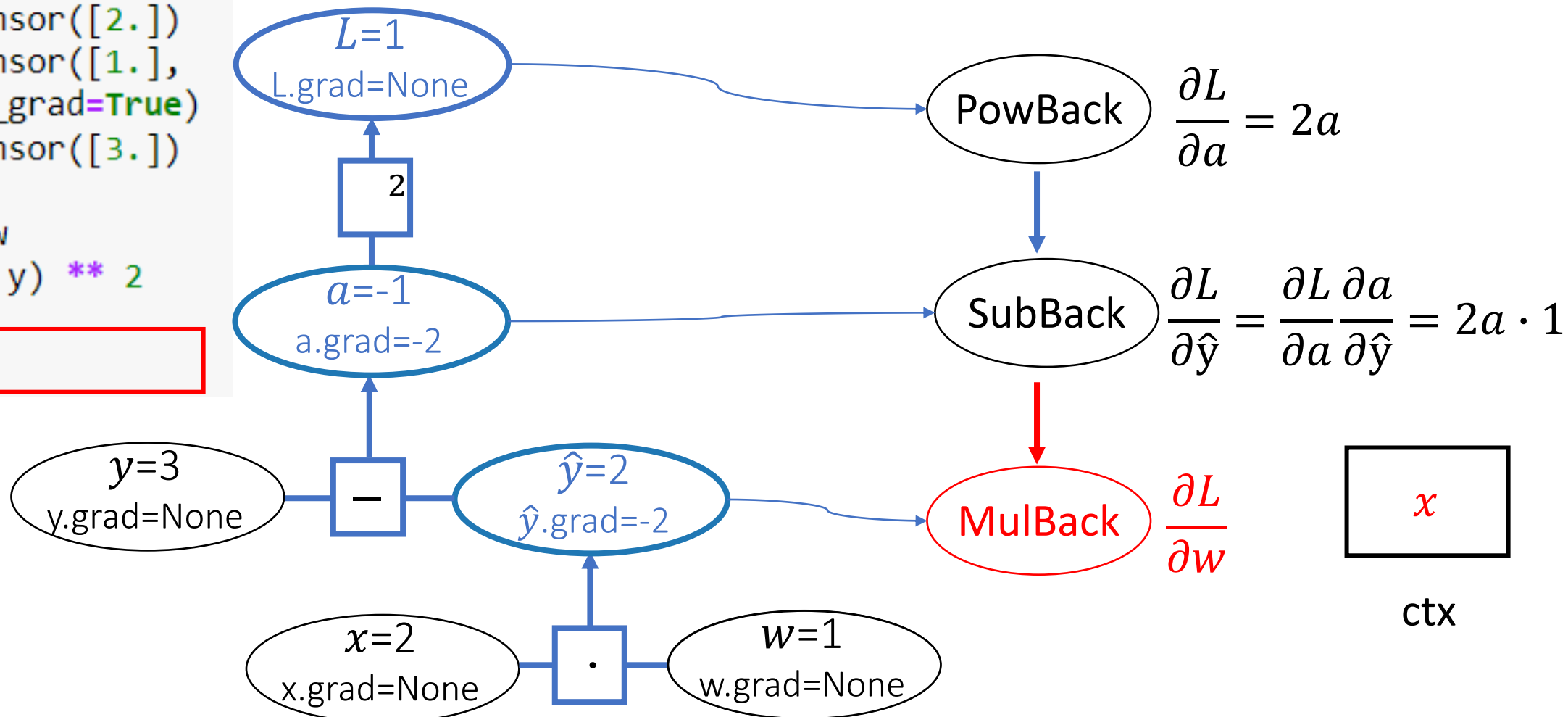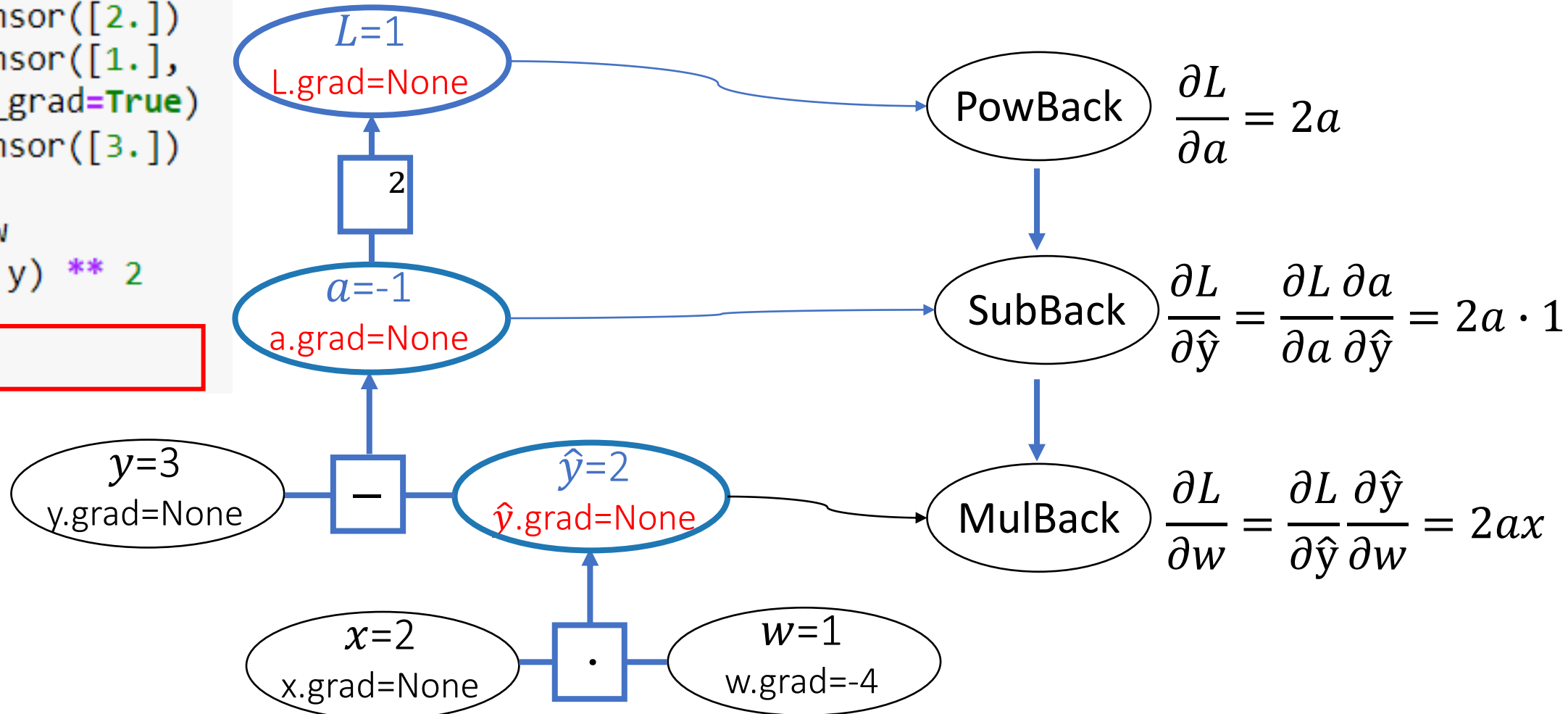MulBack $\quad \dfrac{\partial L}{\partial w}$

x

ctx

DL4CV Weizmann

CG in PyTorch: backward


```
x = torch.tensor([2.])
w = torch.tensor([1.],
      requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()
```

$L=1$
L.grad=None

$a=-1$
a.grad=None

$y=3$
y.grad=None

$\hat{y}=2$
$\hat{y}$.grad=None

$x=2$
x.grad=None

$w=1$
w.grad=-4

PowBack $\quad \dfrac{\partial L}{\partial a} = 2a$

SubBack $\quad \dfrac{\partial L}{\partial \hat{y}} = \dfrac{\partial L}{\partial a}\dfrac{\partial a}{\partial \hat{y}} = 2a \cdot 1$

MulBack $\quad \dfrac{\partial L}{\partial w} = \dfrac{\partial L}{\partial \hat{y}}\dfrac{\partial \hat{y}}{\partial w} = 2ax$

Computational Graph in PyTorch

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()

print(w.grad)
print(x.grad)
```

tensor([-4.])
None

x=2
x.grad=None

w=1
w.grad=-4

Computational Graph in PyTorch

```python
x = torch.tensor([2.])
w = torch.tensor([1.],
    requires_grad=True)
y = torch.tensor([3.])

y_hat = x * w
L = (y_hat - y) ** 2

L.backward()


L = (x * w - y) ** 2
L.backward()

print(w.grad)
```

$x=2$

x.grad=None

$w=1$

w.grad=-8

Autograd

Disable Gradients

```python
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```
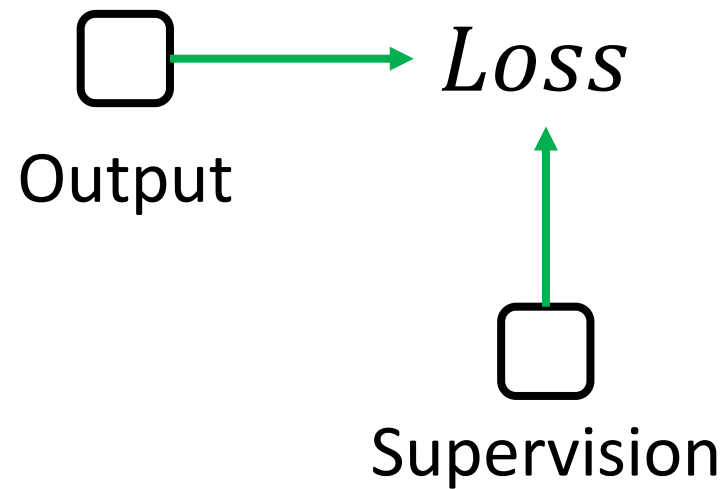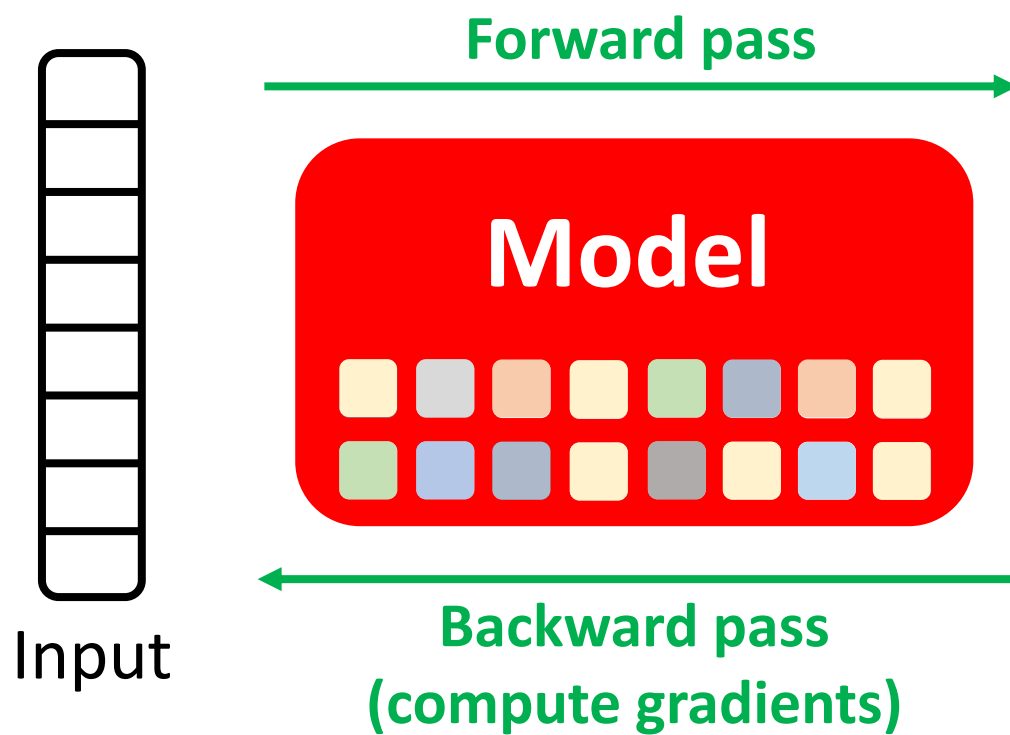
```
True
True
False
```
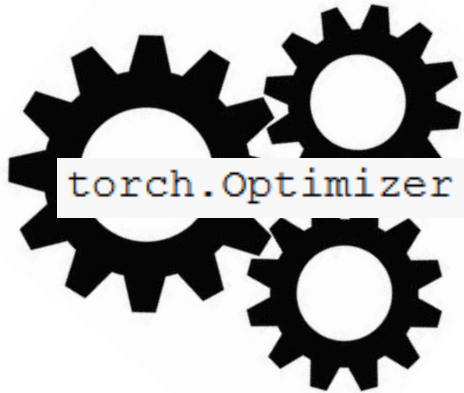
Back to NumPy

```python
np_out = out.detach().cpu().numpy()
```

Based on Pytorch tutorials

Backpropagation

Forward pass

Model

Input

Backward pass
(compute gradients)

torch.Optimizer

Loss

Output

Supervision

Optimizers

SGD: $w \leftarrow w - \alpha \cdot \dfrac{\partial L}{\partial w}$

torch.Optimizer

Create an SGD optimizer

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Reset Gradients

```
optimizer.zero_grad()
```

Perform an optimization step

```
optimizer.step()
```

Putting it All Together – Fashion MNIST Classification

Putting it All Together

1. Hyperparameters
2. Handling Data
3. Model, Loss, Optimizer
4. Training and Inference
5. Overall Training Process

Based on Pytorch tutorials

1. Hyperparameters

```python
# configurations
LR = 1e-3  # learning rate
B_SQRT = 8
B = B_SQRT**2  # batch size
EPOCHS = 100  # num epochs
INPUT_SIZE = 28  # input size

device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Based on Pytorch tutorials

2. Handling Data

```python
# handling data
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

val_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Downloading datasets

Setting dataset manager
- Automatic batching
- Customized data loading order
- …

3. Model, Loss, Optimizer

```python
# define model
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(INPUT_SIZE*INPUT_SIZE, 512),
  nn.ReLU(),
  nn.Linear(512, 512),
  nn.ReLU(),
  nn.Linear(512, 10),
)


# move to correct device
model.to(device)
```

```python
# define loss
criterion = nn.CrossEntropyLoss()
```

```python
# define optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=LR)
```

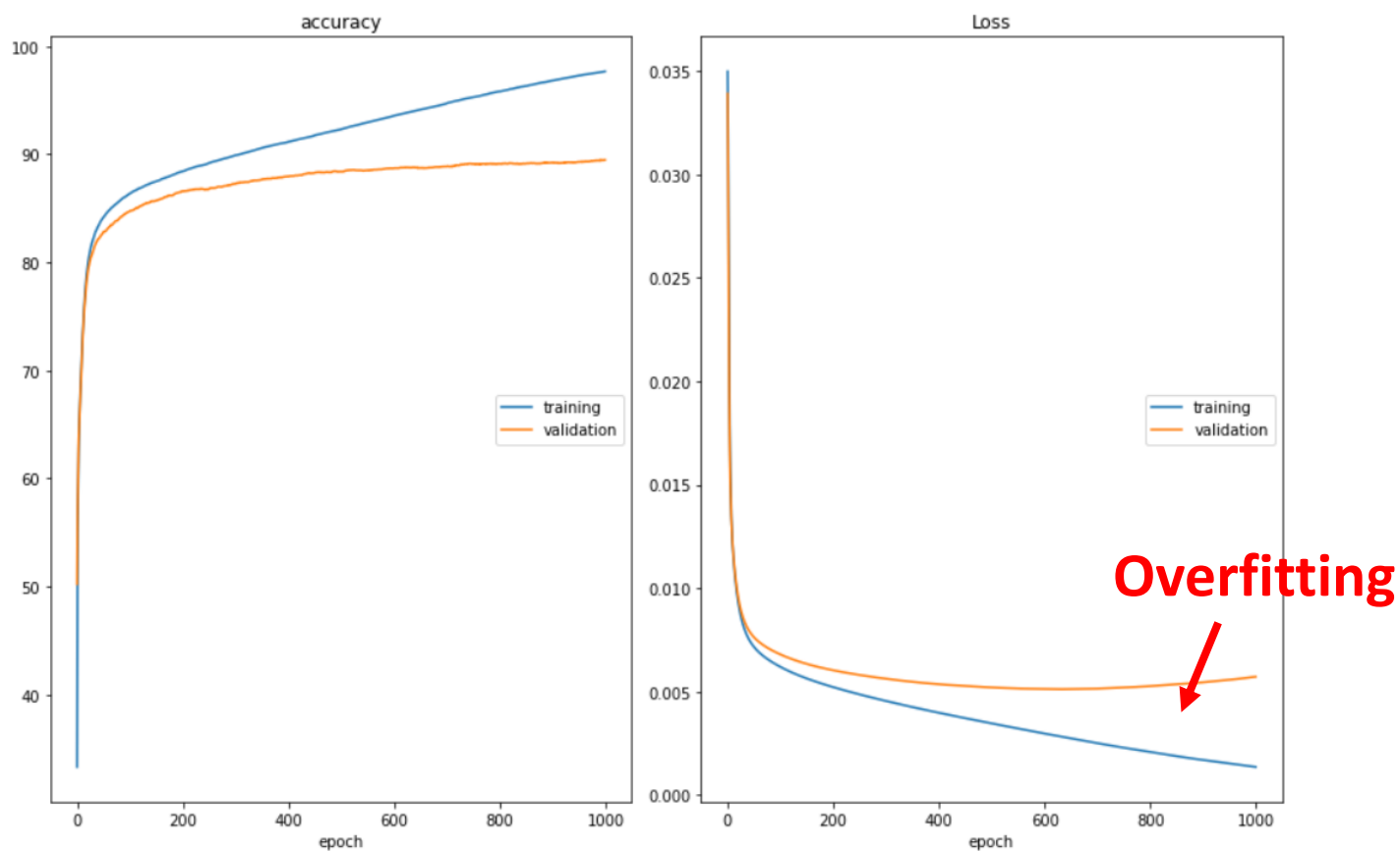# 4. Training and Inference

Training Loop

```python
# iterate through all batches
for batch, (X, y) in enumerate(dataloader):
    # move data to device
    X, y = X.to(device), y.to(device)
    # forward pass
    pred = model(X)
    loss = criterion(pred, y)
    # new gradients per batch
    optimizer.zero_grad()
    # backward pass
    loss.backward()
    # update
    optimizer.step()
```

Inference Loop

```python
# disregard gradients when not training
with torch.no_grad():
    # iterate through all batches
    for X, y in dataloader:
        # move data to device
        X, y = X.to(device), y.to(device)
        # forward pass
        pred = model(X)
```

Based on Pytorch tutorials

5. Overall Training Process

```
for t in range(EPOCHS):
    train_res = train_loop(train_dataloader, model, criterion, optimizer)
    val_res = inference_loop(val_dataloader, model, criterion)
```


Based on Pytorch tutorials

Weights & Biases

Visualizations

Monitor experiments

jupyter

TensorBoard

Tensor

nn.functional

hooks

Parameters search

PyTorch

AutoGrad

nn.Module

torchvision

. . .

einops

Transforms

Albumentations

kornia

DL4CV Weizmann

Next Monday:
No class (Christmas)

Next Wednesday:
CNNs