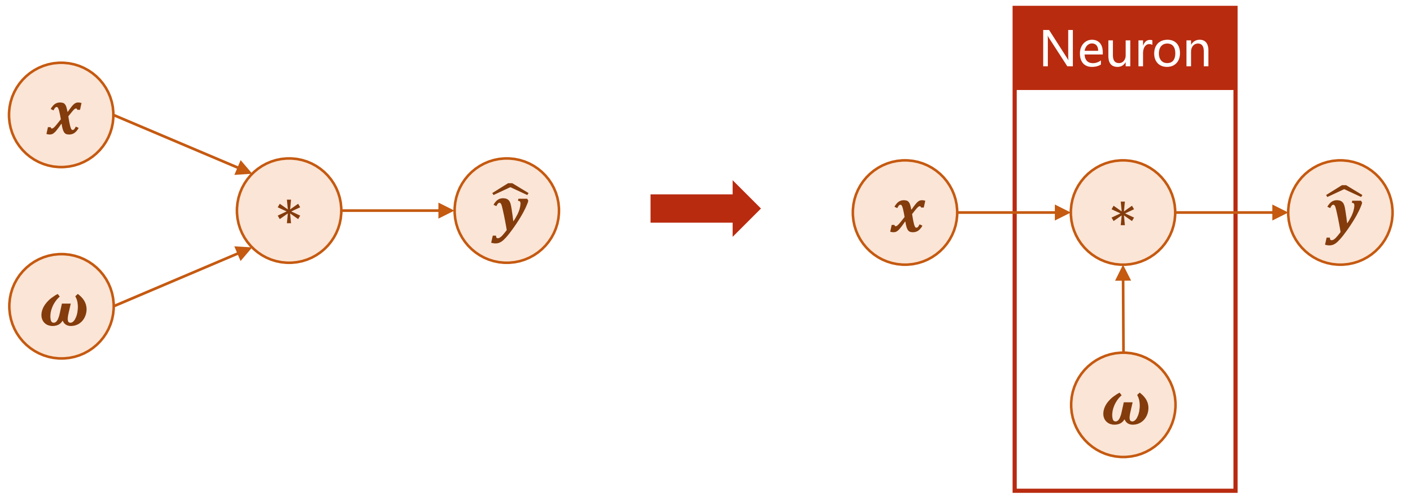
PyTorch Tutorial

04. Back Propagation

Compute gradient in simple network

Linear Model

$$\hat{y} = x * \omega$$



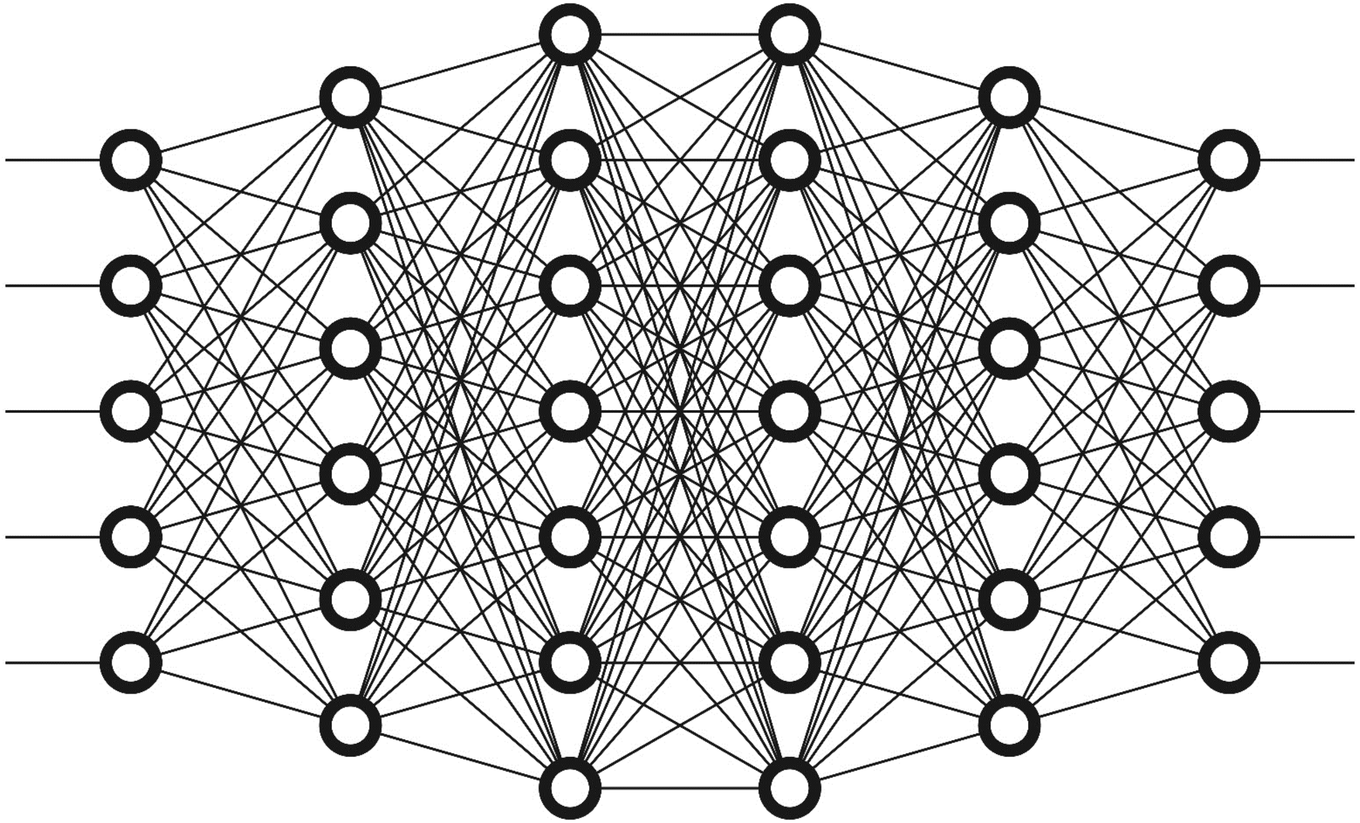
Stochastic Gradient Descent

$$\omega = \omega - \alpha \frac{\partial loss}{\partial \omega}$$

Derivative of Loss Function

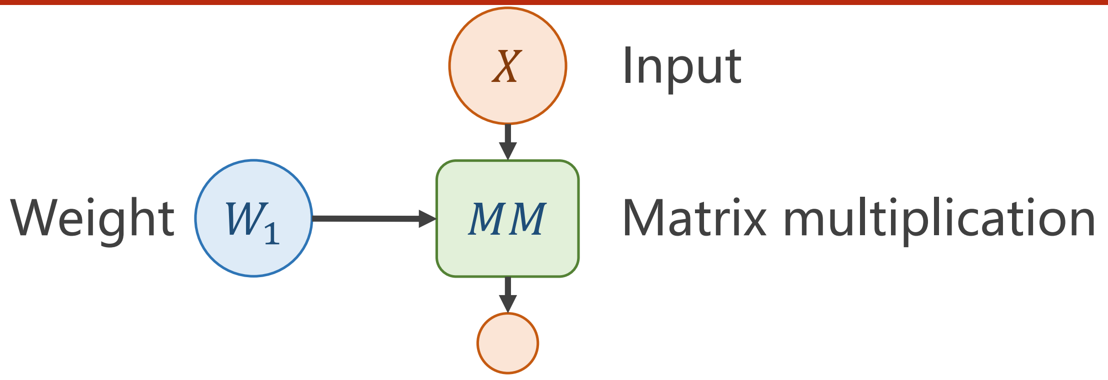
$$\frac{\partial loss_n}{\partial \omega} = 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

What about the complicated network?



Gradient

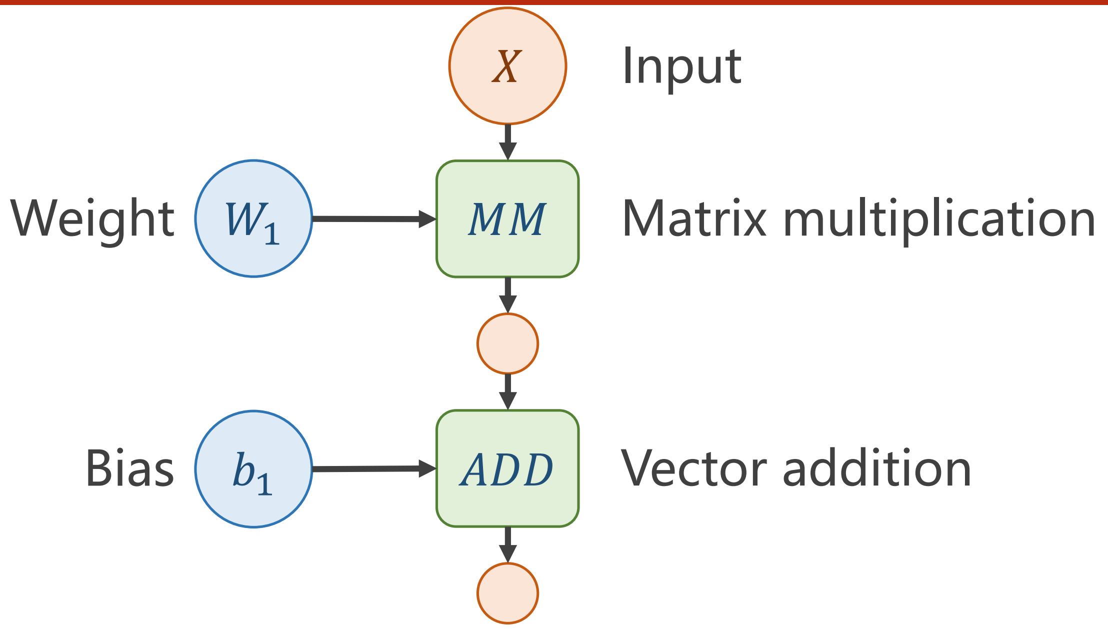
$$\frac{\partial loss}{\partial \omega} = ?$$



A two layer neural network

$$\hat{y} = W_2(\underline{W_1 \cdot X} + b_1) + b_2$$

Computational Graph



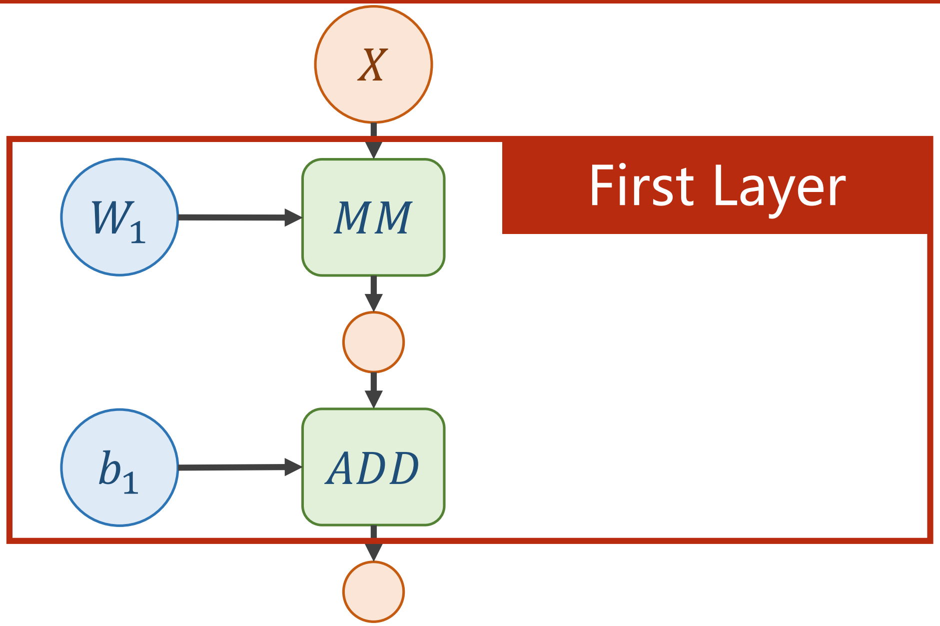
A two layer neural network

$$\hat{y} = W_2(\underline{W_1 \cdot X + b_1}) + b_2$$

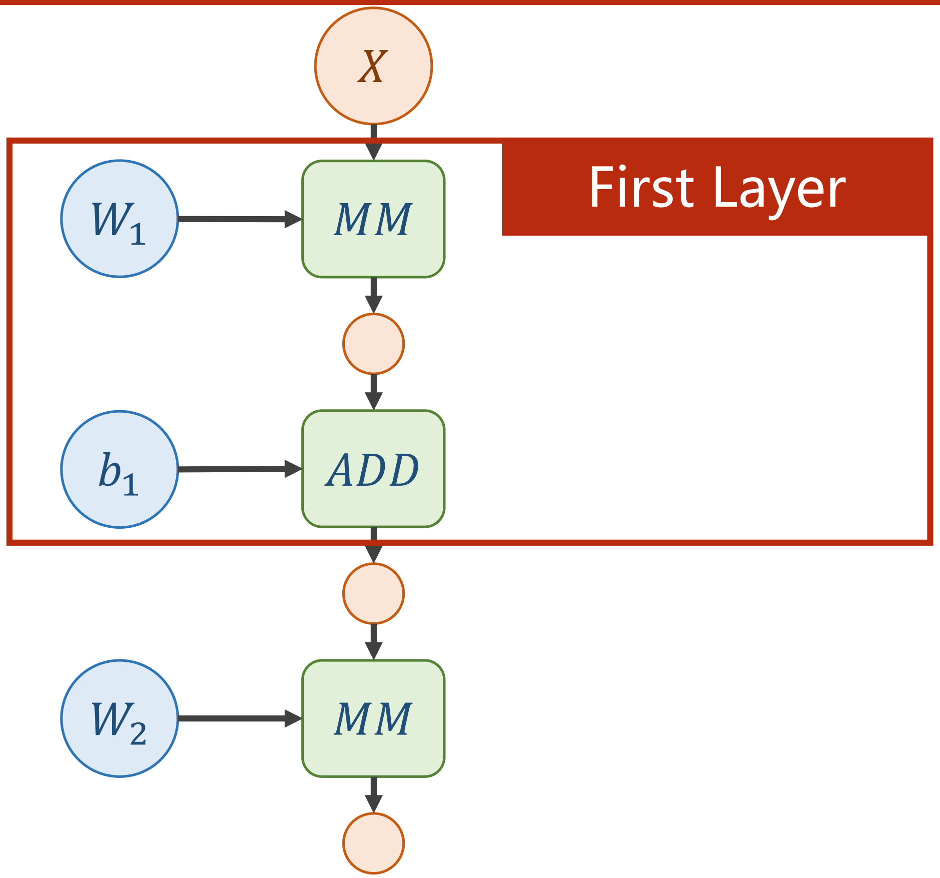
Computational Graph

A two layer neural network

$$\hat{y} = W_2(\underline{W_1 \cdot X + b_1}) + b_2$$



Computational Graph



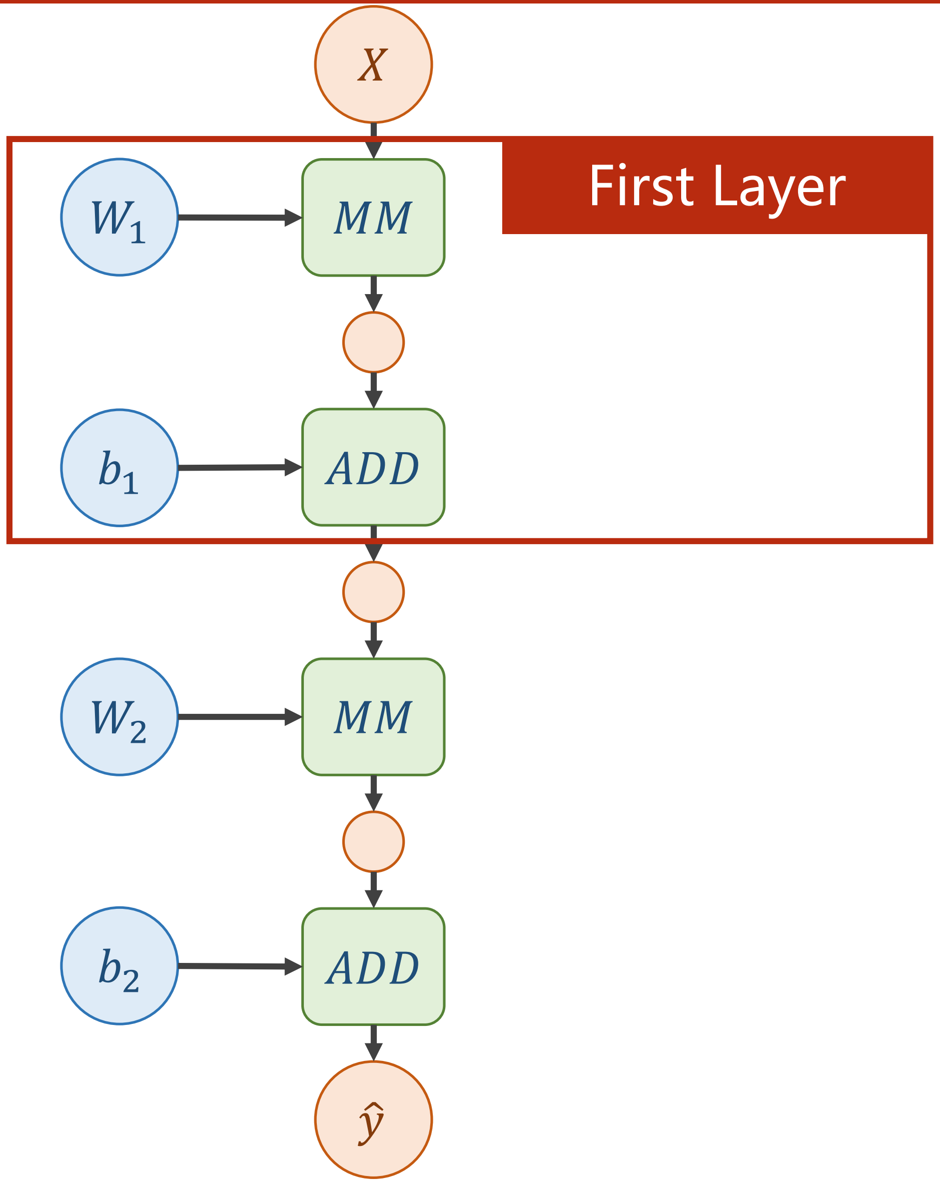
A two layer neural network

$$\hat{y} = \underline{W_2(W_1 \cdot X + b_1)} + b_2$$

Computational Graph

A two layer neural network

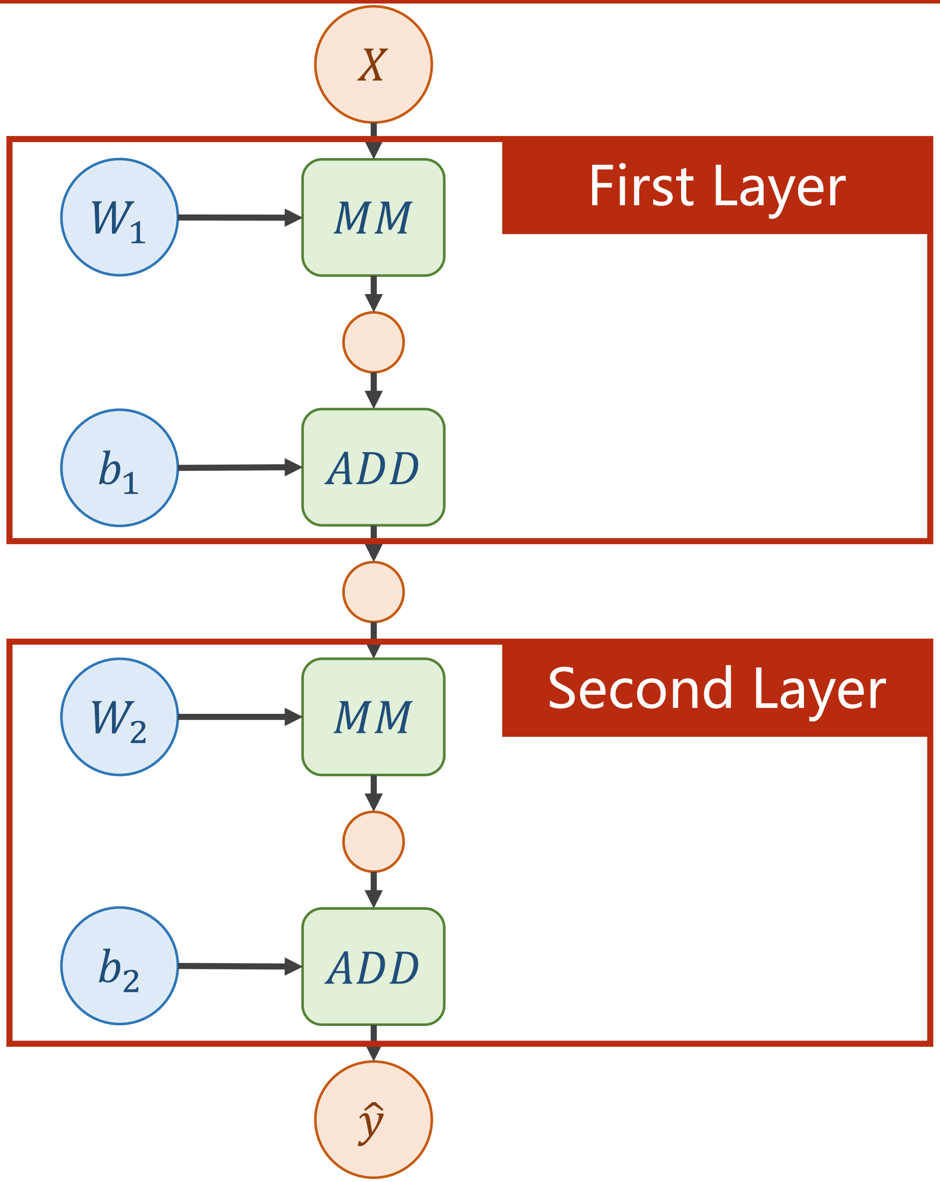
$$\hat{y} = \underline{W_2(W_1 \cdot X + b_1) + b_2}$$



Computational Graph

A two layer neural network

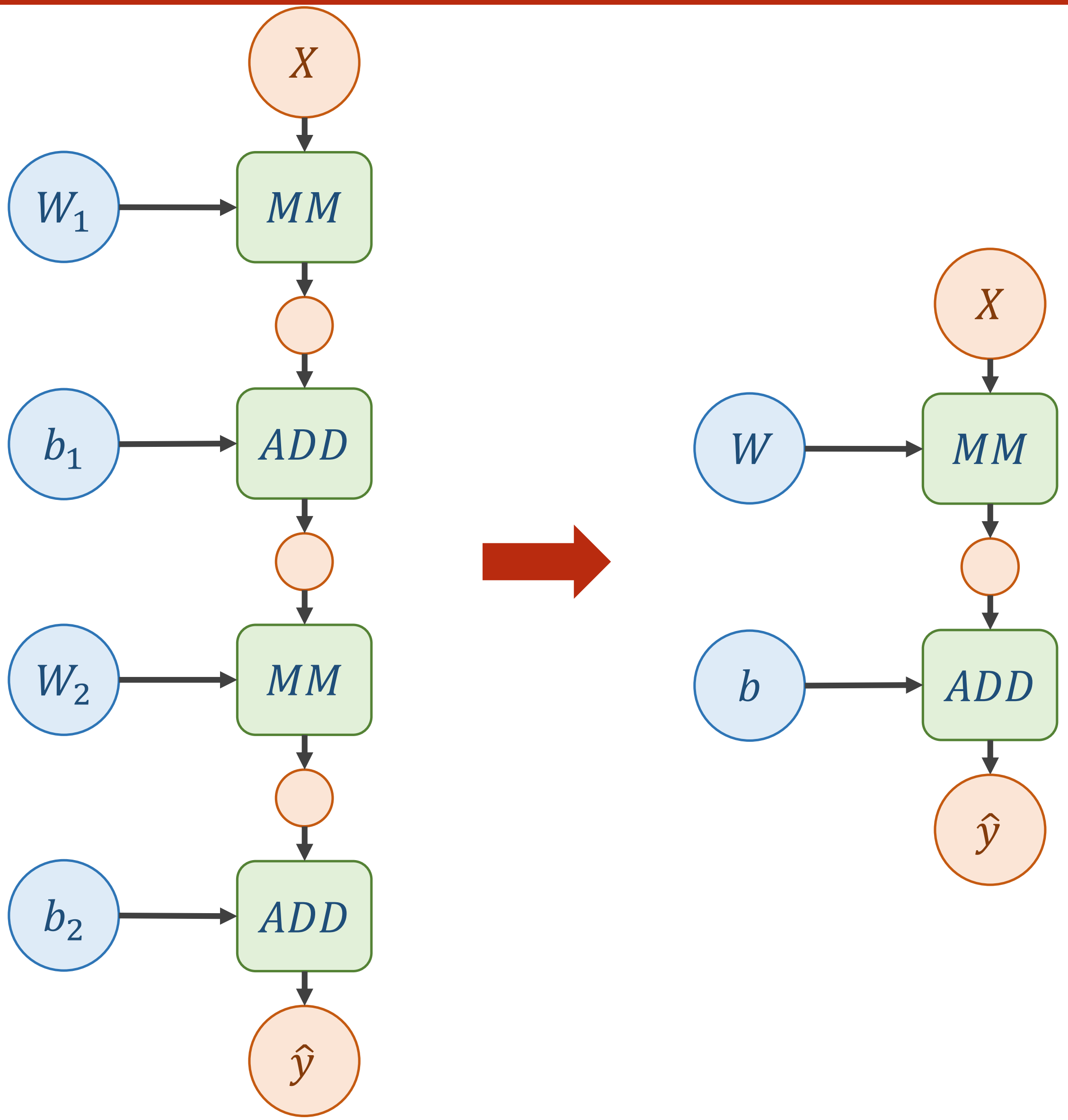
$$\hat{y} = \underline{W_2(W_1 \cdot X + b_1) + b_2}$$



What problem about this two layer neural network?

A two layer neural network

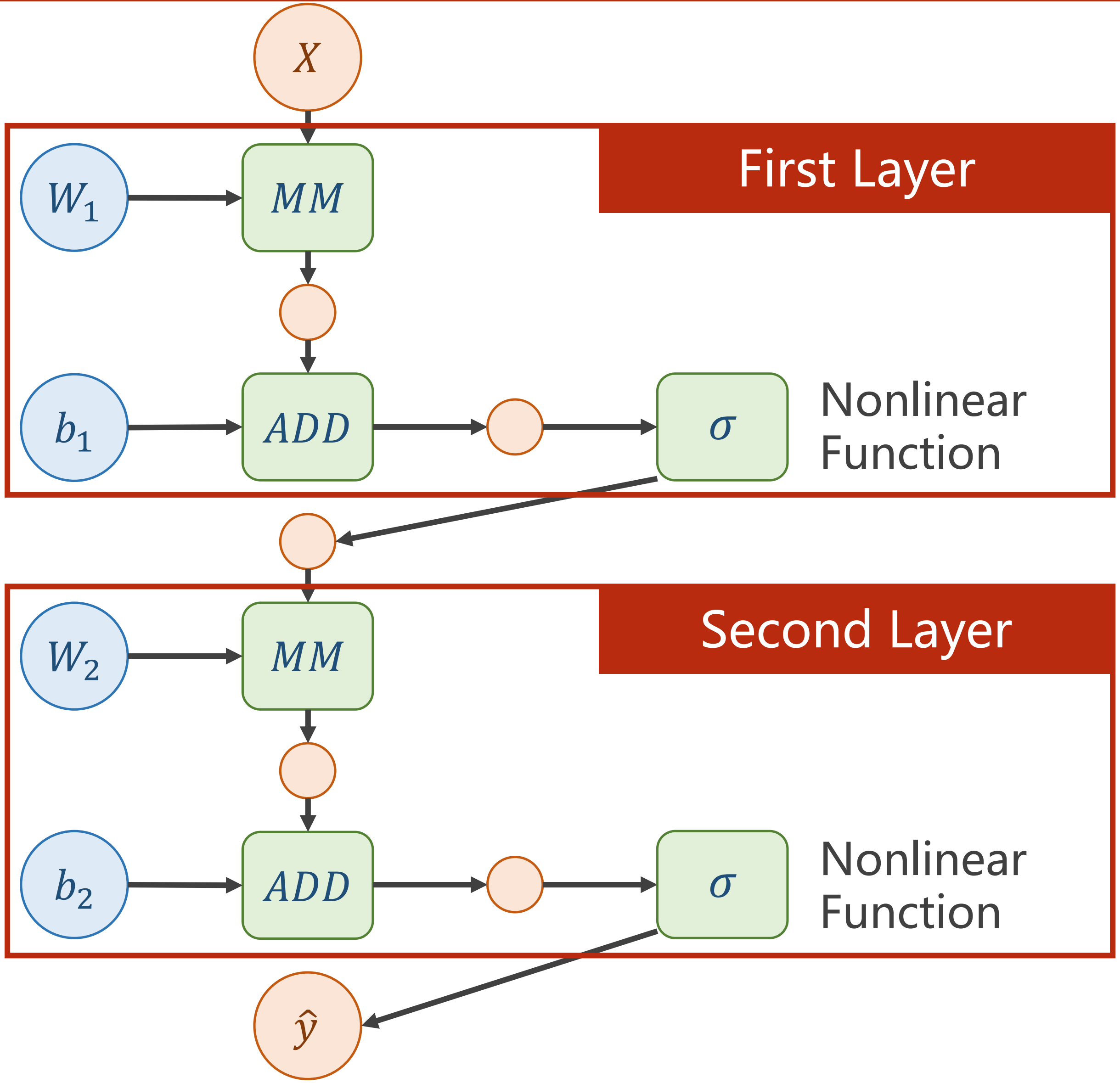
$$\hat{y} = W_2(W_1 \cdot X + b_1) + b_2$$
$$= W_2 \cdot W_1 \cdot X + (W_2 b_1 + b_2)$$
$$= W \cdot X + b$$



What problem about this two layer neural network?

A two layer neural network

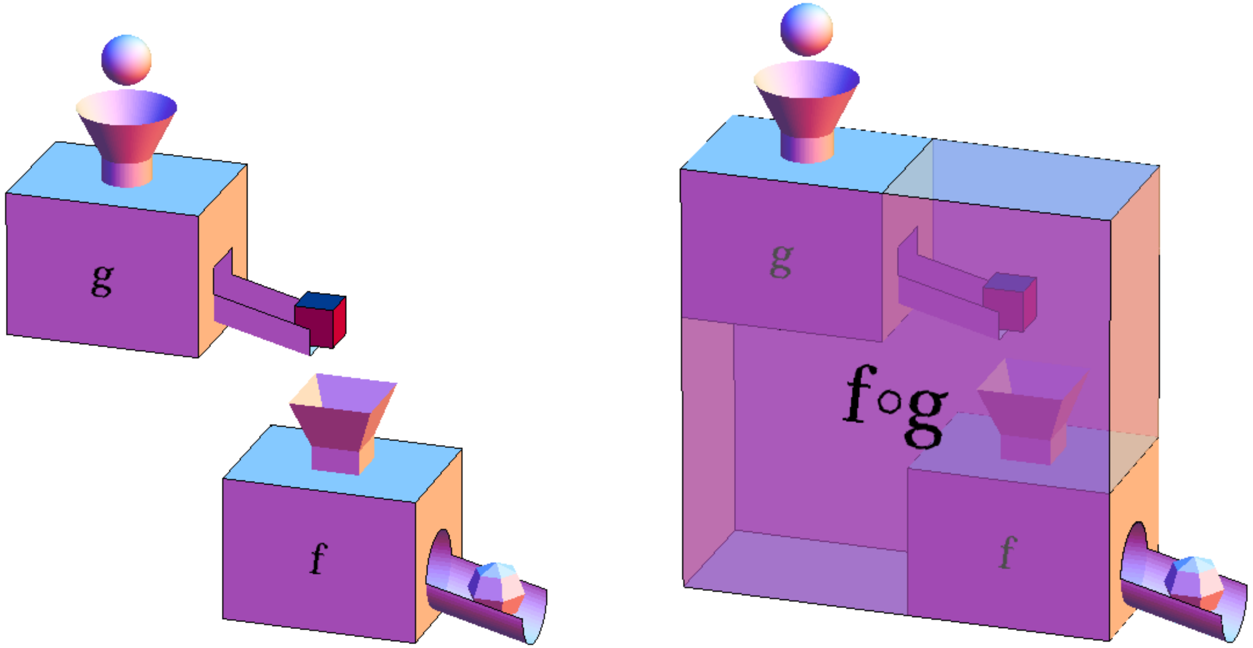
$$\hat{y} = W_2(W_1 \cdot X + b_1) + b_2$$
$$= W_2 \cdot W_1 \cdot X + (W_2 b_1 + b_2)$$
$$= W \cdot X + b$$

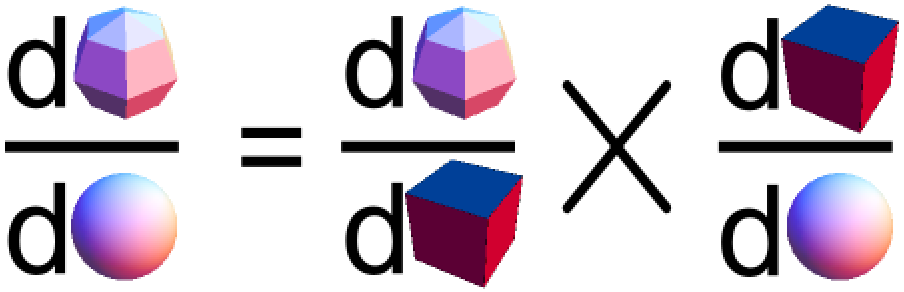


A nonlinear function is required by each layer.

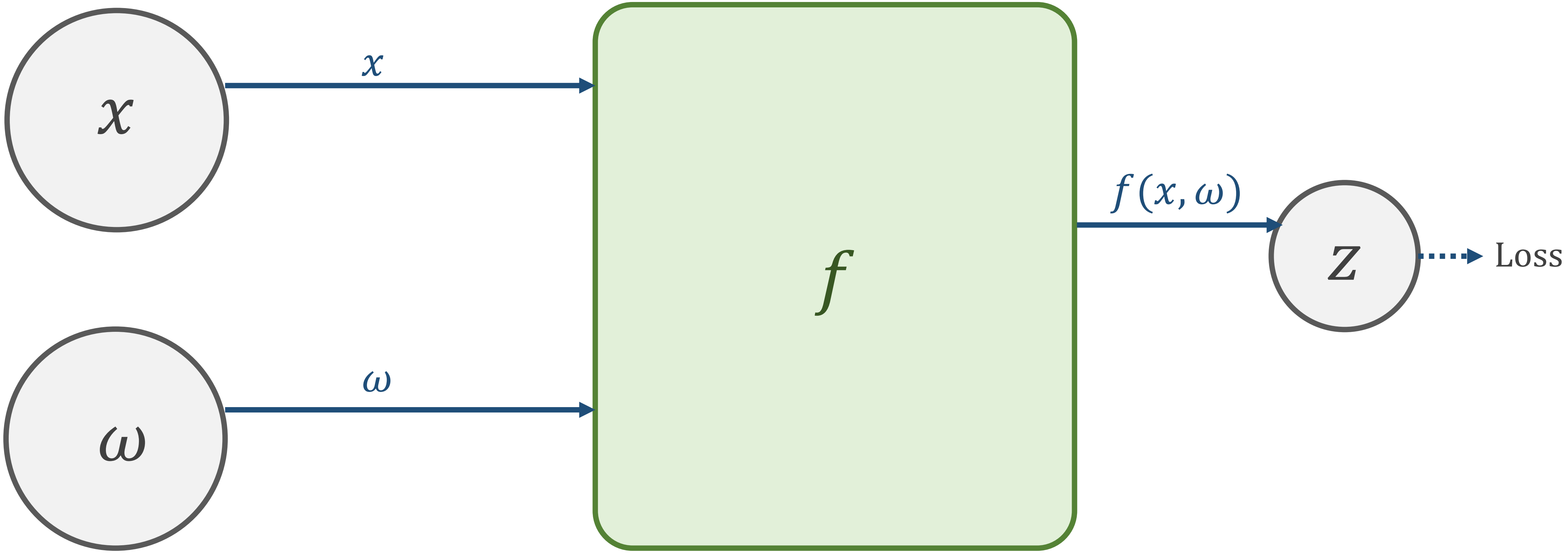
We shall talk about this later.

The composition of functions and Chain Rule

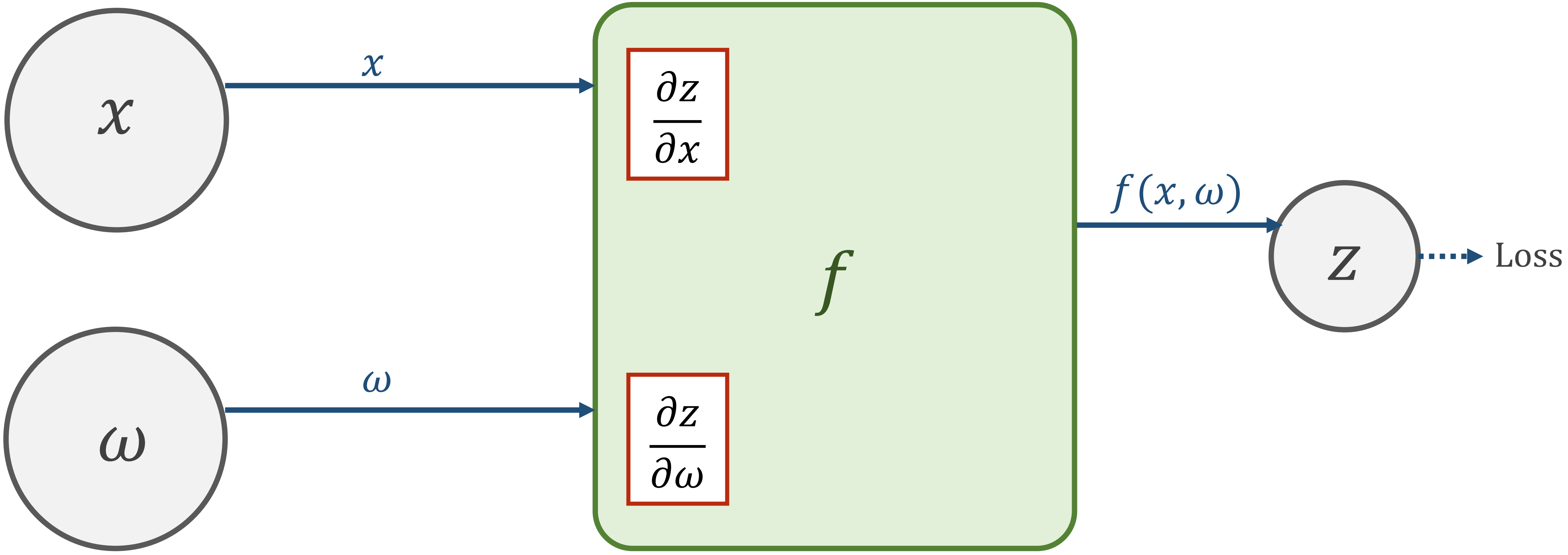




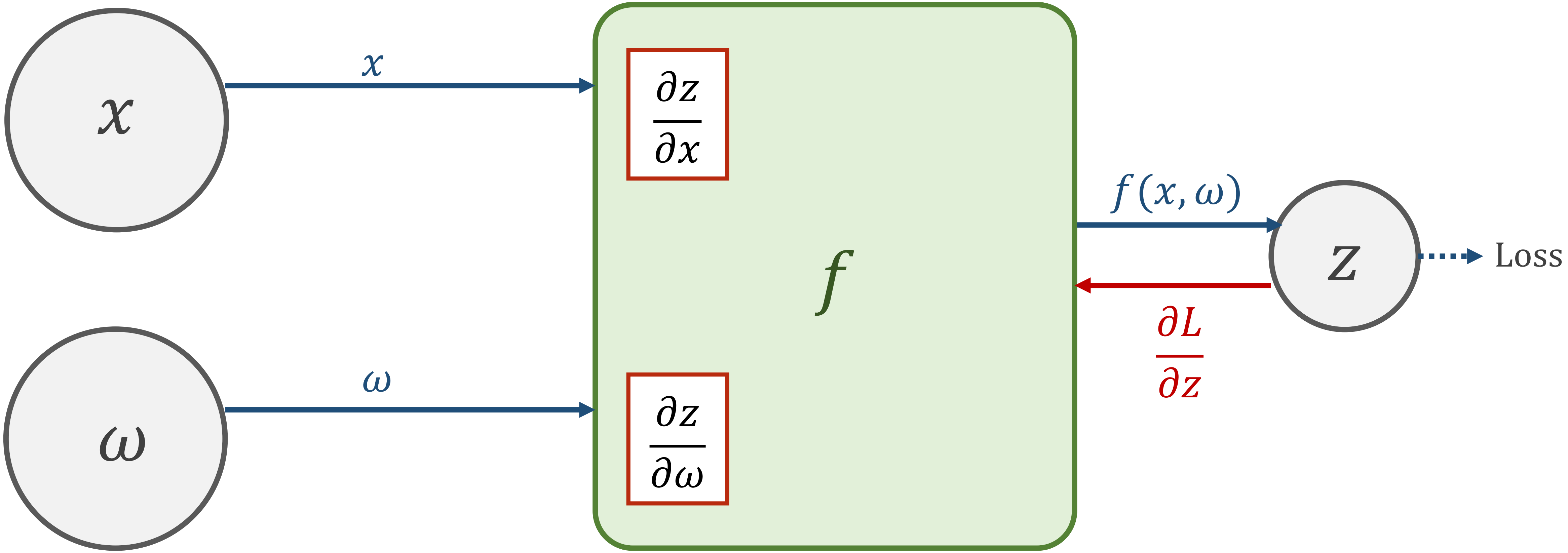
Chain Rule – 1. Create Computational Graph (Forward)



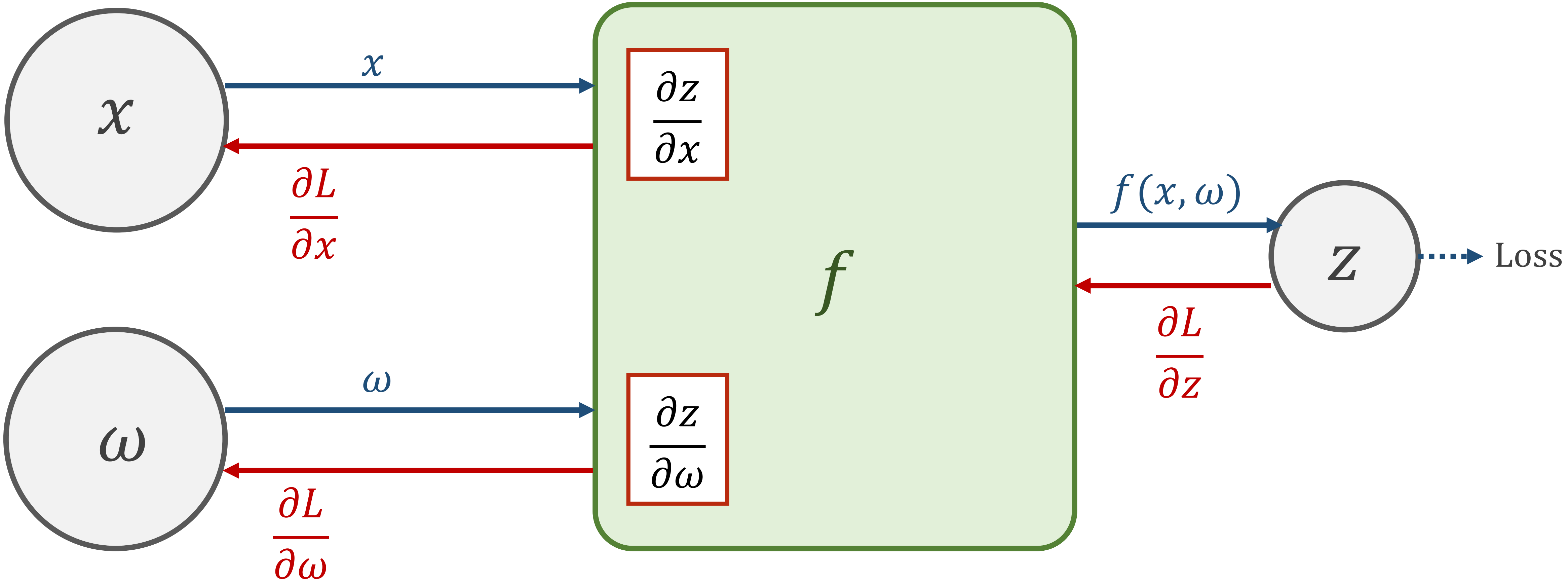
Chain Rule – 2. Local Gradient



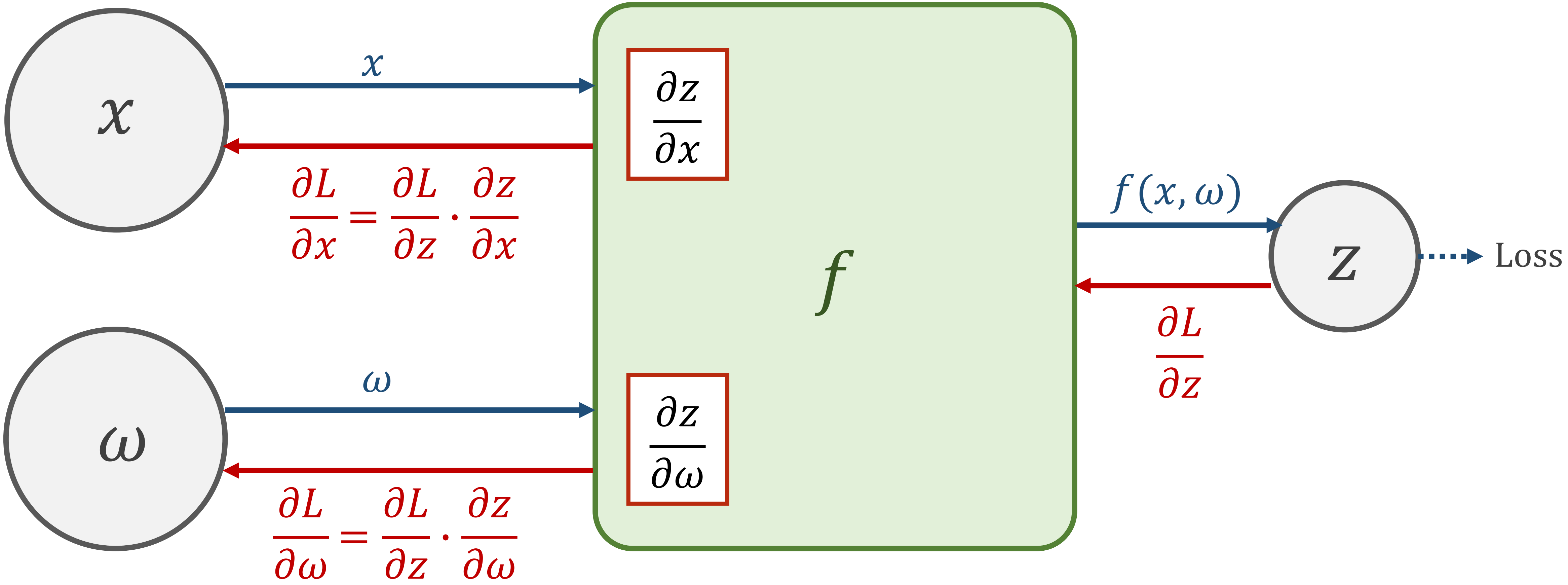
Chain Rule – 3. Given gradient from successive node



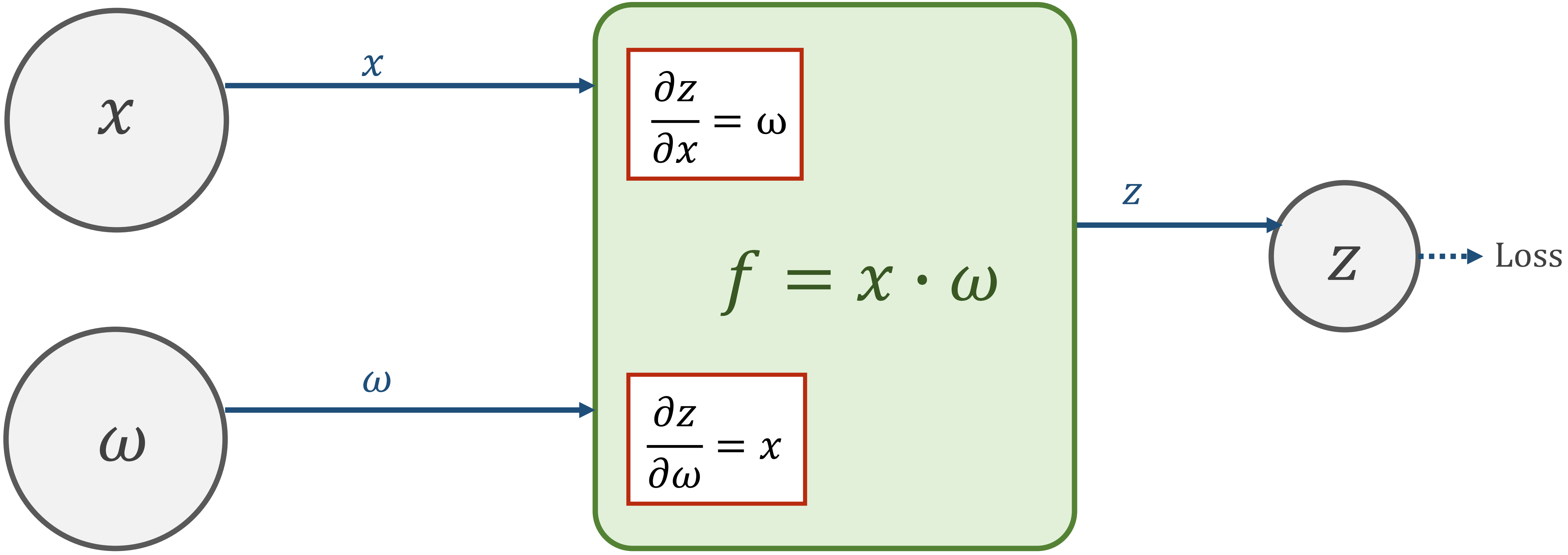
Chain Rule – 4. Use chain rule to compute the gradient (Backward)



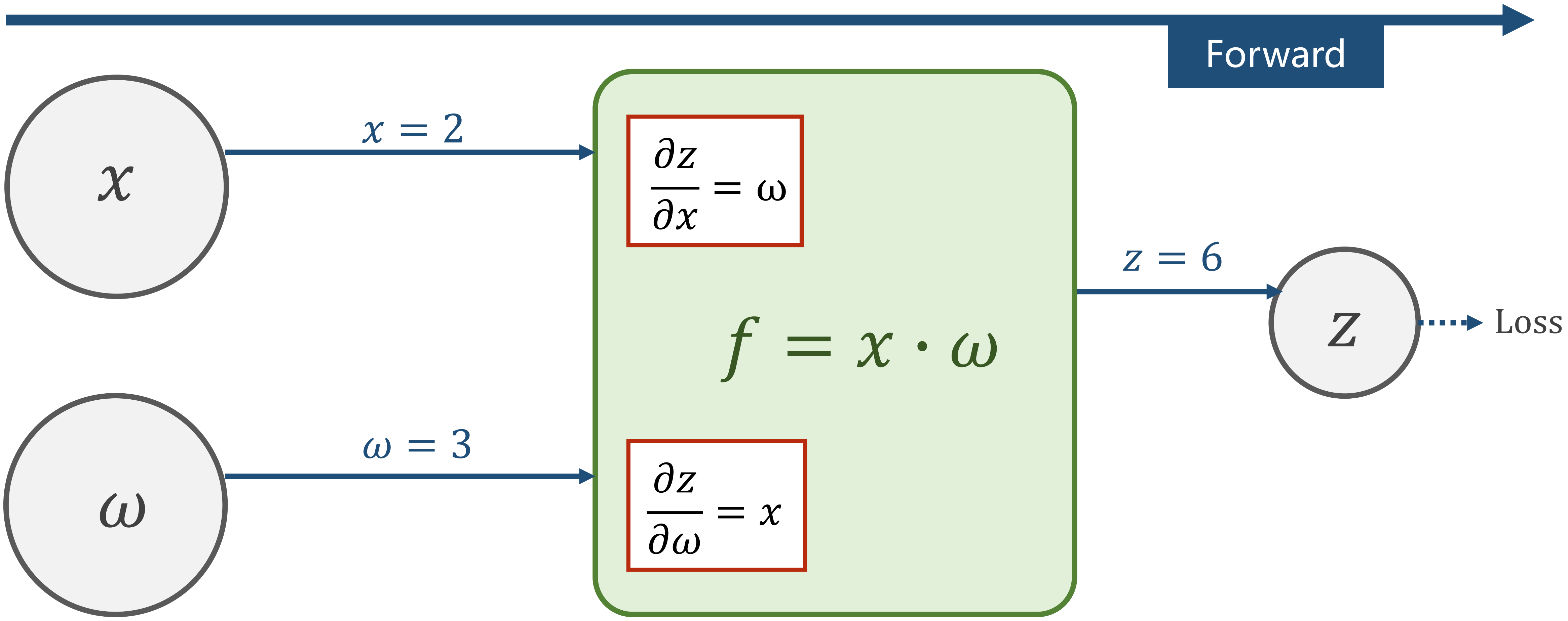
Chain Rule – 4. Use chain rule to compute the gradient (Backward)



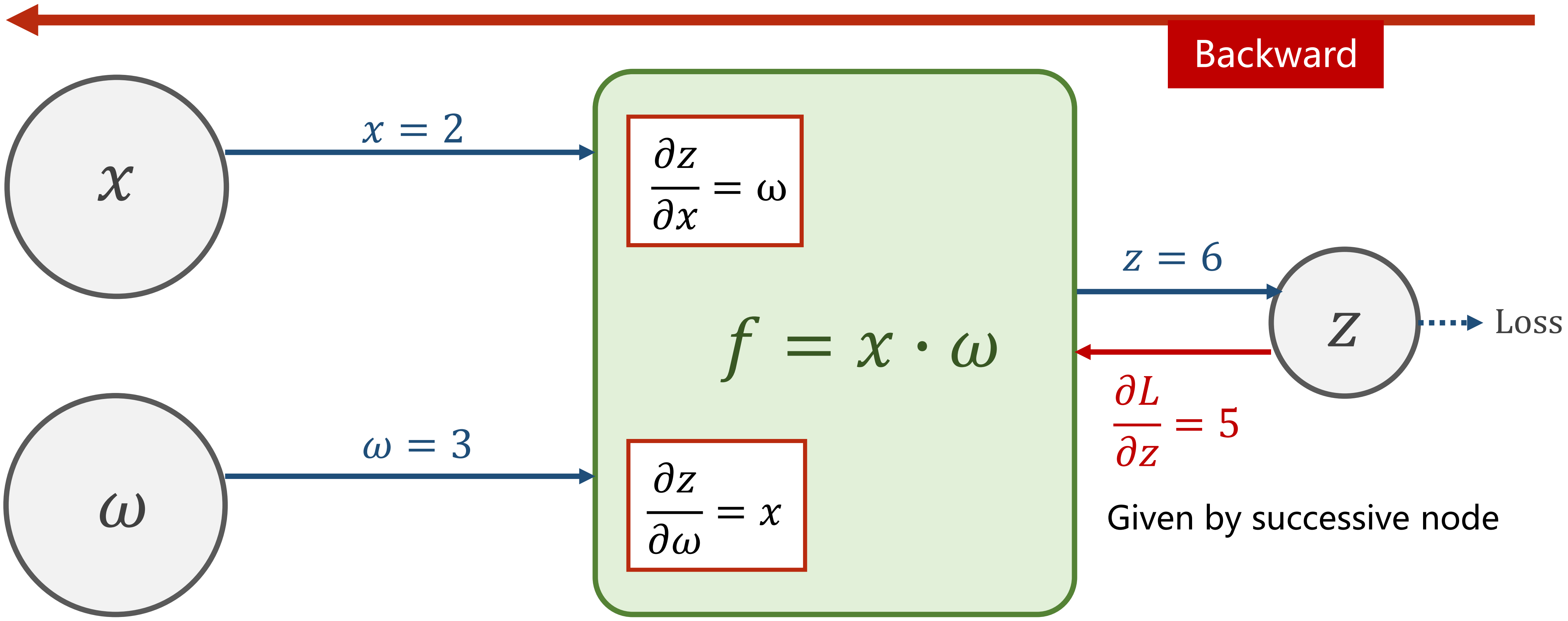
Example: $f = x \cdot \omega$



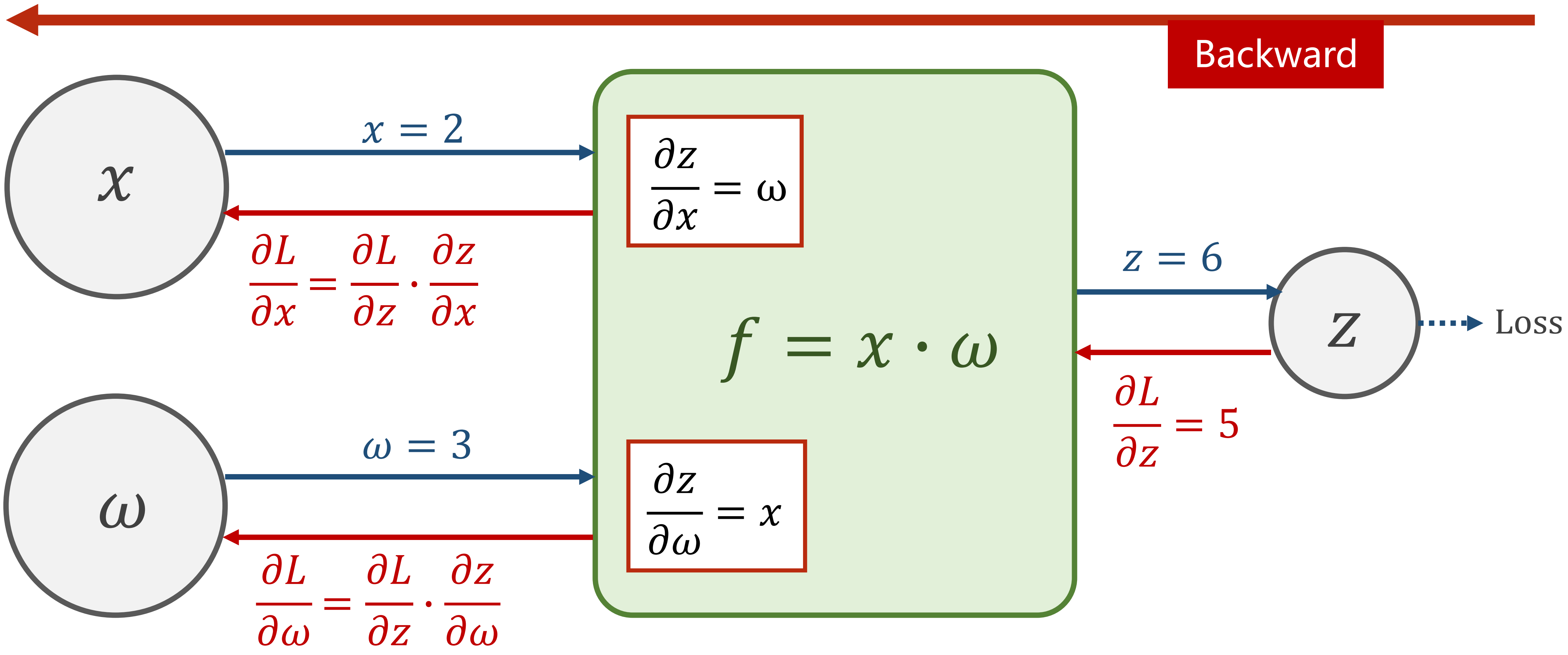
Example: $f = x \cdot \omega, x = 2, \omega = 3$



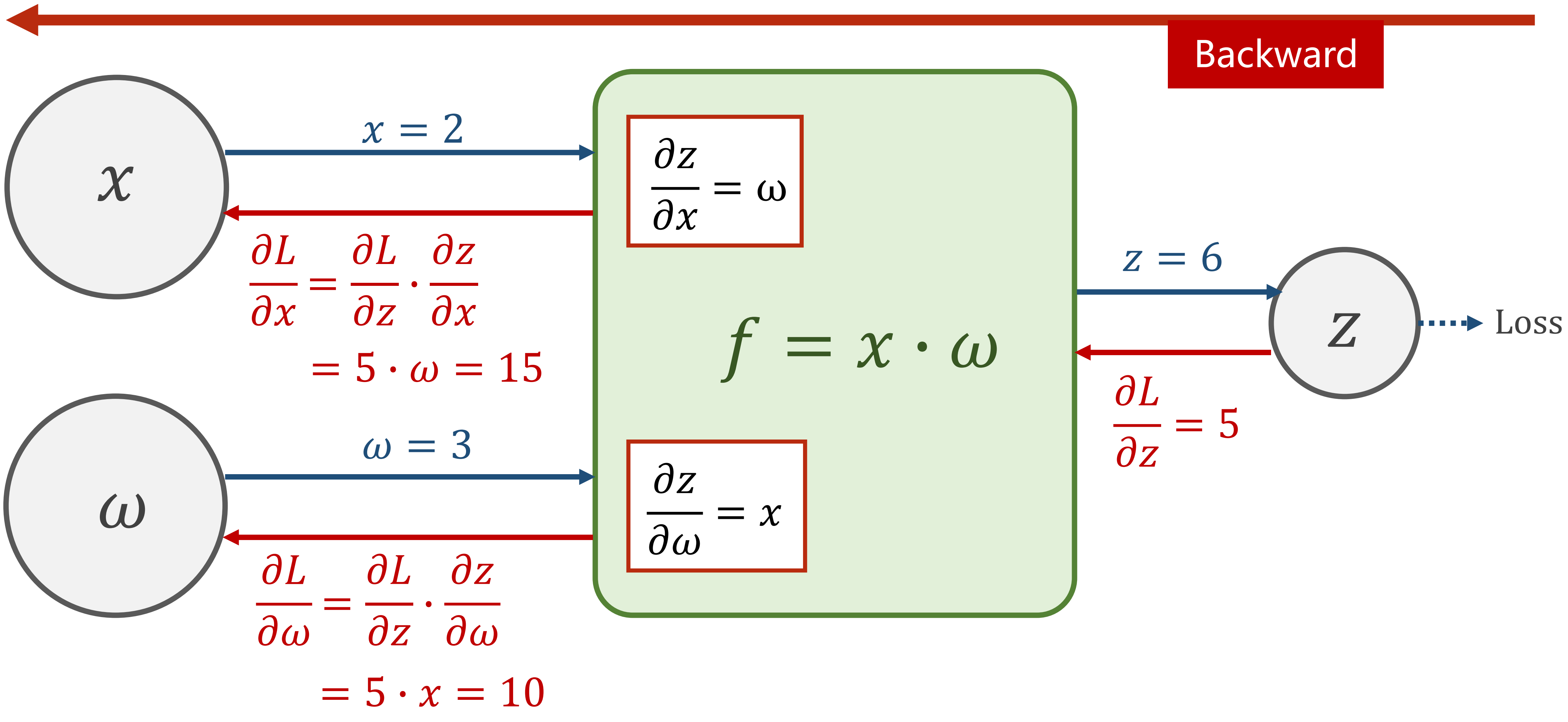
Example: Backward



Example: Backward



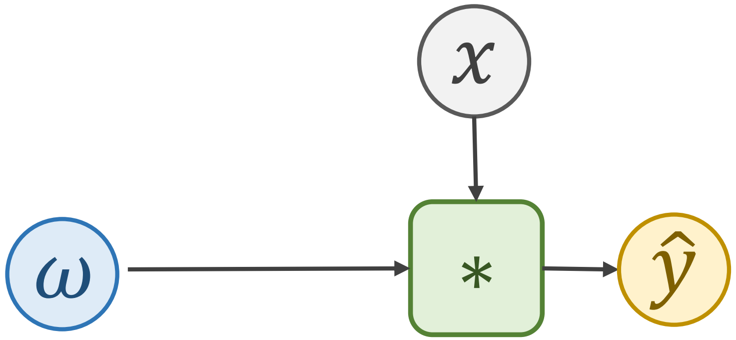
Example: Backward



Computational Graph of Linear Model

Linear Model

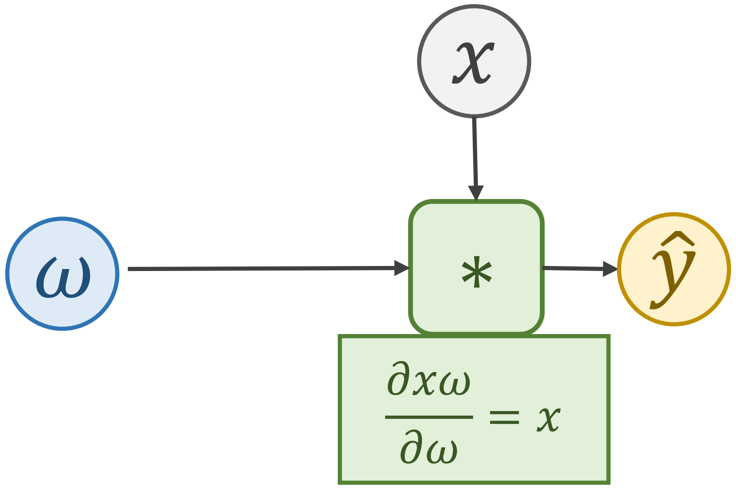
$$\hat{y} = x * \omega$$



Computational Graph of Linear Model

Linear Model

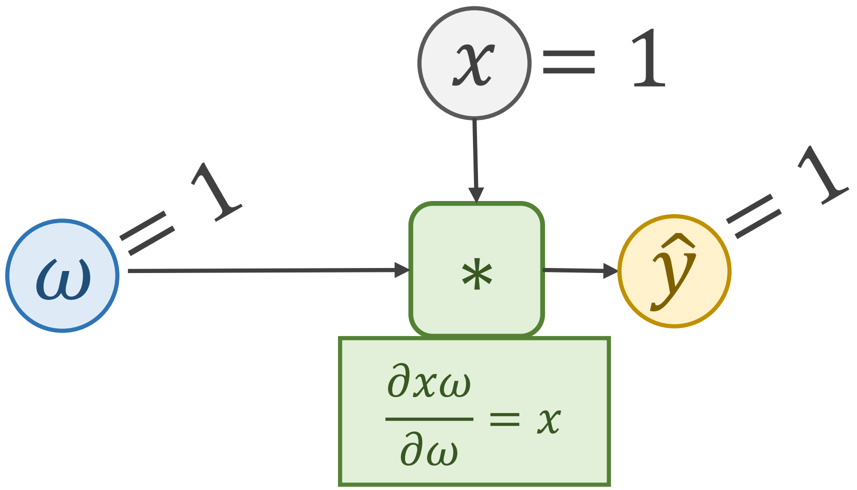
$$\hat{y} = x * \omega$$



Computational Graph of Linear Model

Linear Model

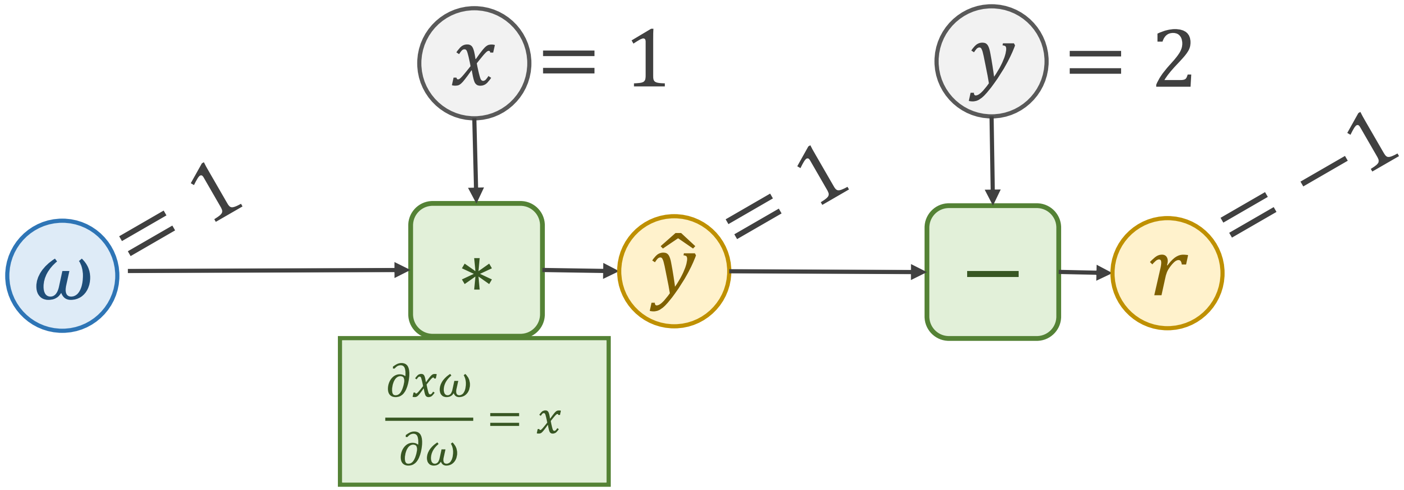
$$\hat{y} = x * \omega$$



Computational Graph of Linear Model

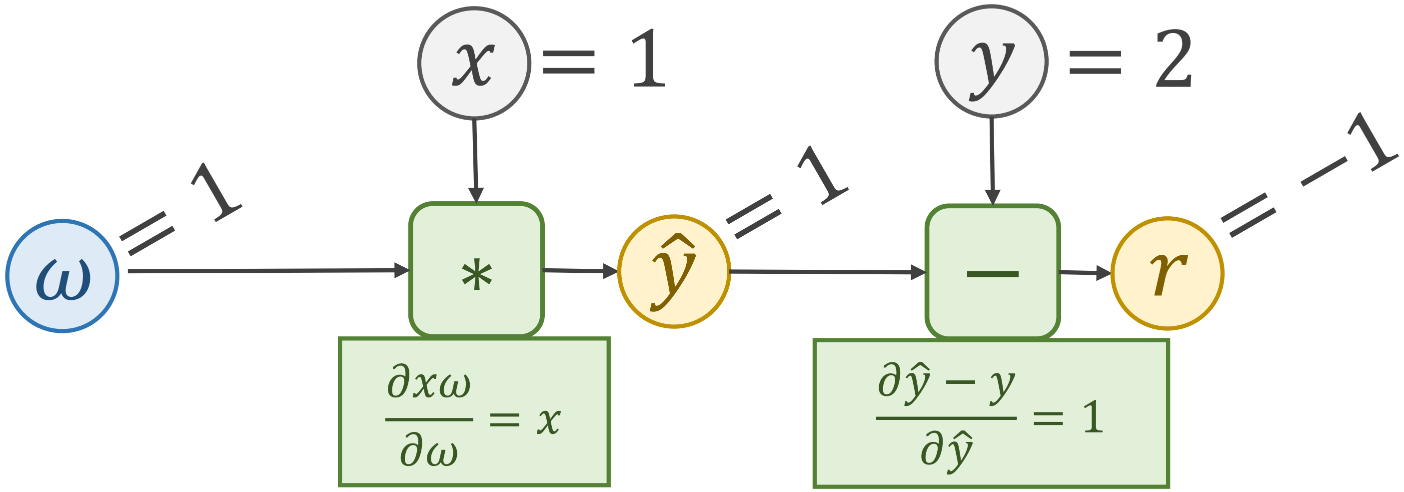
Linear Model
$\hat{y} = x * \omega$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Linear Model
$\hat{y} = x * \omega$

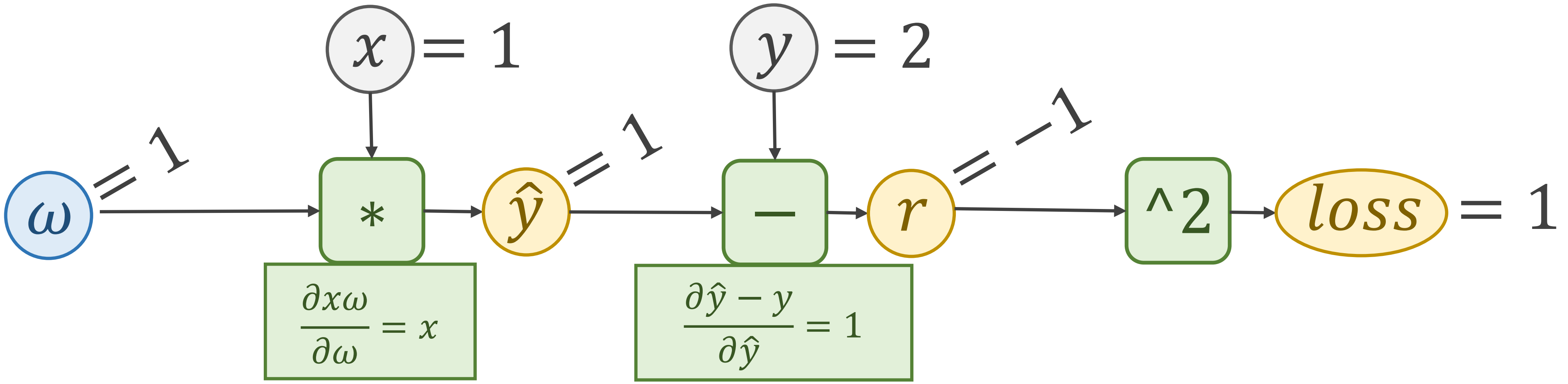
Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Computational Graph of Linear Model

Linear Model
$\hat{y} = x * \omega$

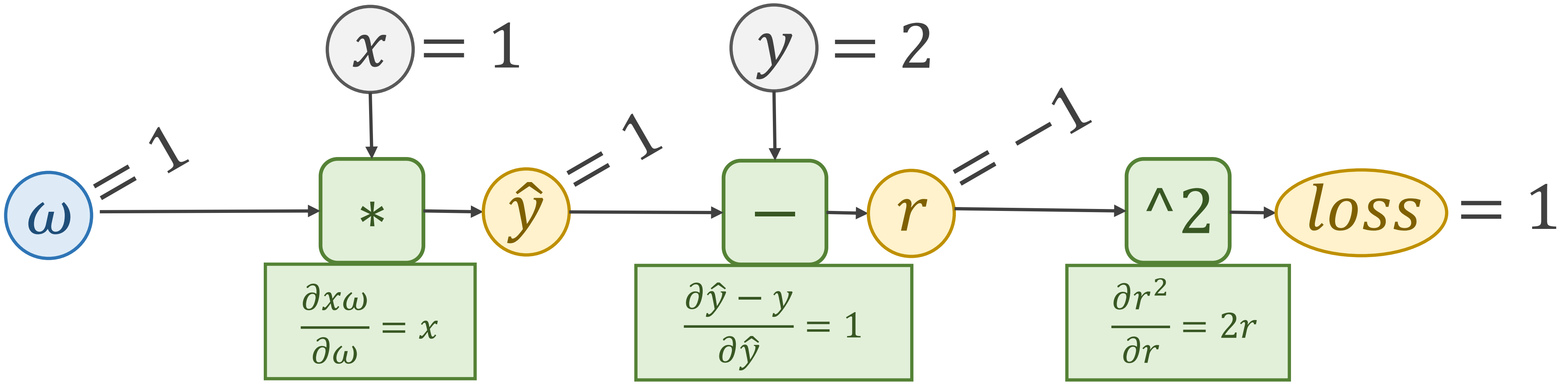
Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Computational Graph of Linear Model

Linear Model
$\hat{y} = x * \omega$

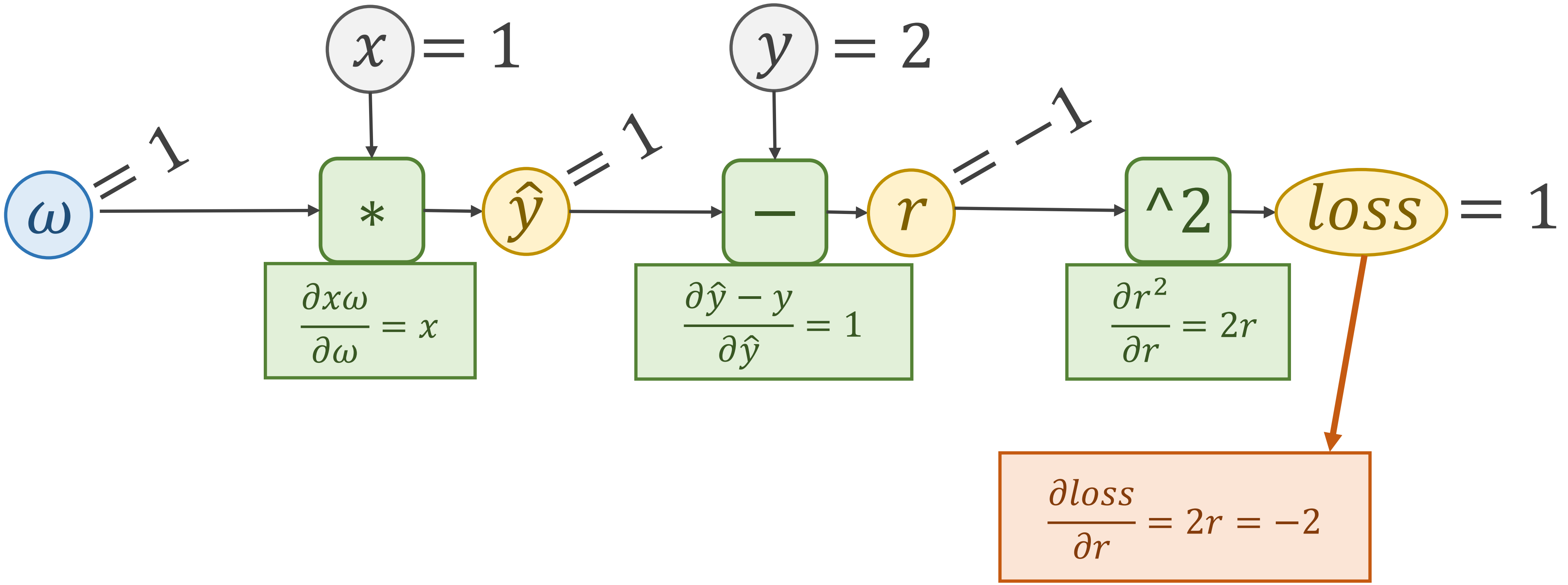
Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Computational Graph of Linear Model

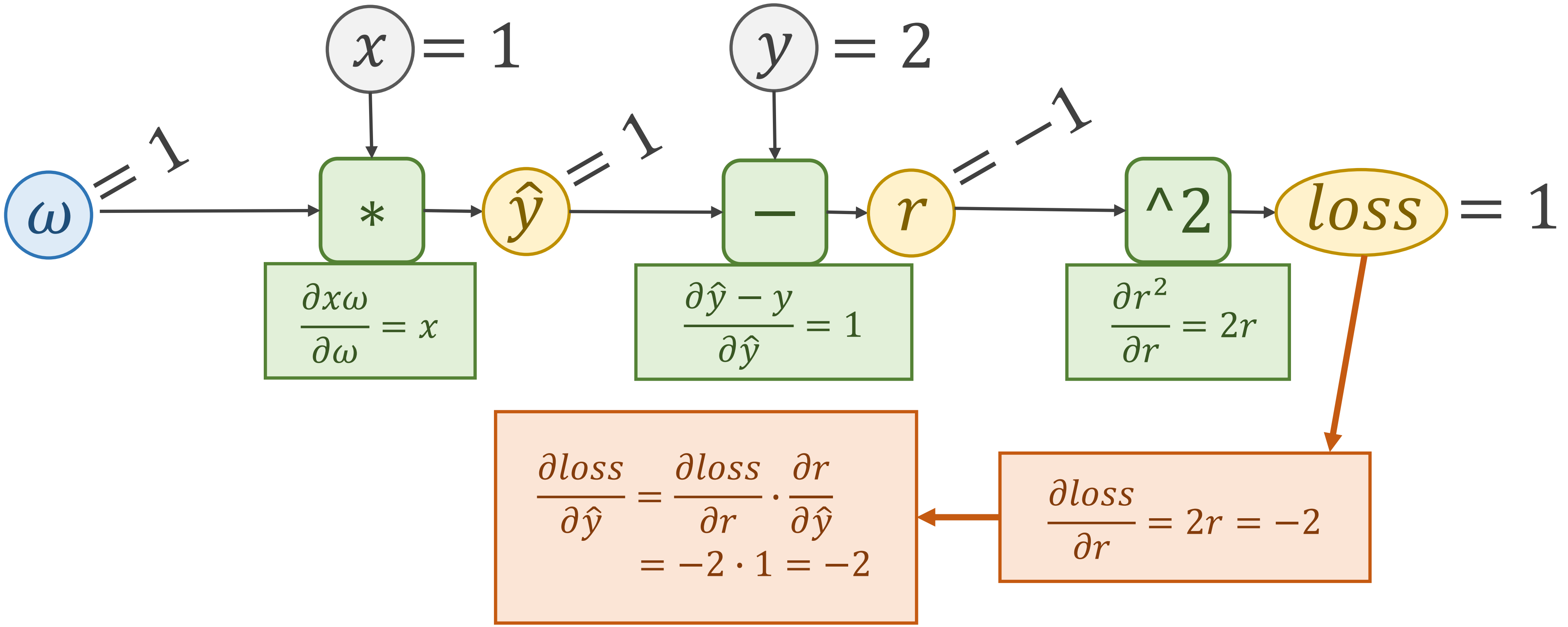
Linear Model
$\hat{y} = x * \omega$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Computational Graph of Linear Model

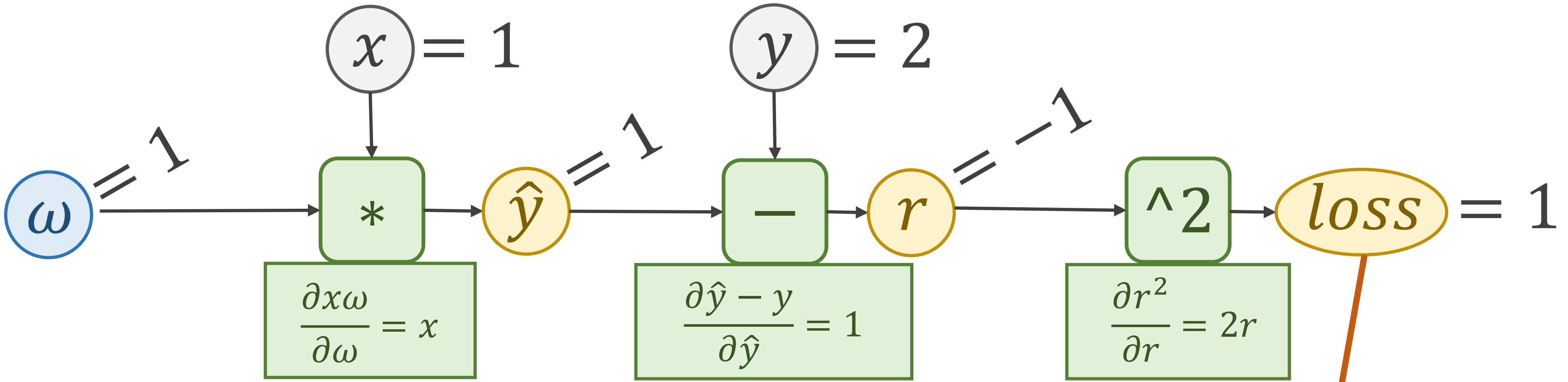
Linear Model	Loss Function
$\hat{y} = x * \omega$	$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



Computational Graph of Linear Model

Linear Model
$\hat{y} = x * \omega$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



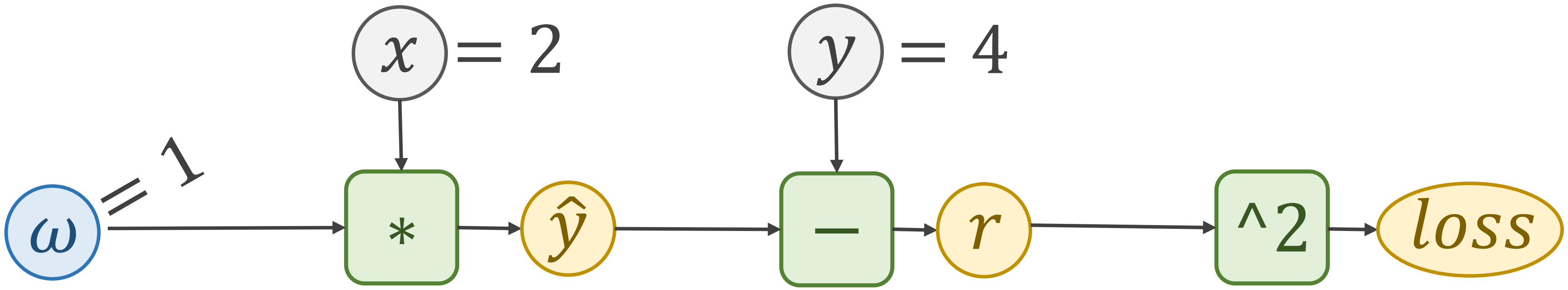
$$\frac{\partial loss}{\partial \omega} = \frac{\partial loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \omega} = -2 \cdot x = -2 \cdot 1 = -2$$

$$\frac{\partial loss}{\partial \hat{y}} = \frac{\partial loss}{\partial r} \cdot \frac{\partial r}{\partial \hat{y}} = -2 \cdot 1 = -2$$

$$\frac{\partial loss}{\partial r} = 2r = -2$$

Linear Model
$\hat{y} = x * \omega$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$

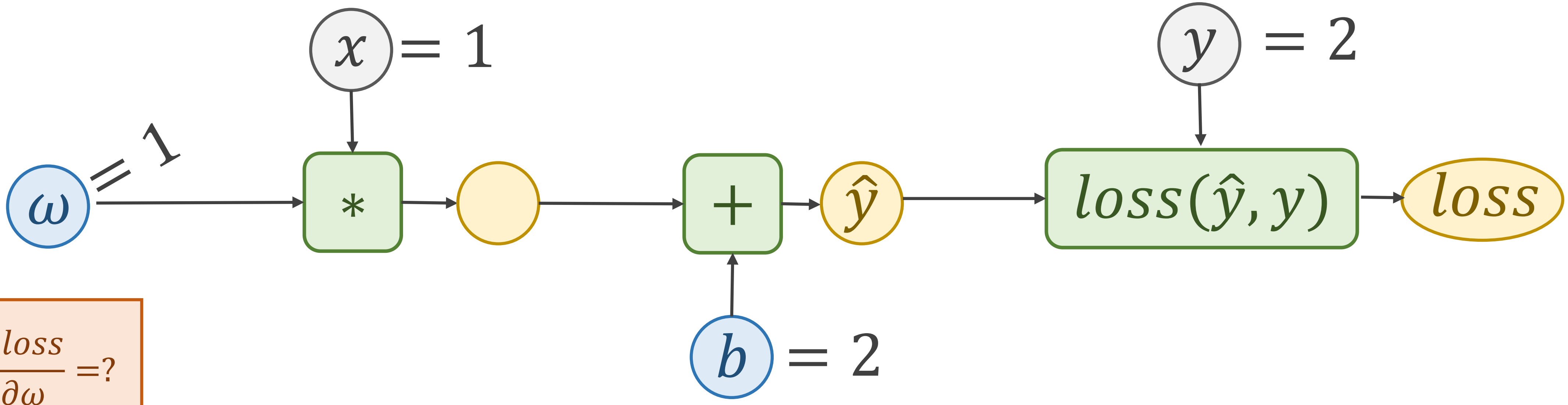


$$\frac{\partial loss}{\partial \omega} = ?$$

Exercise 4-2: Compute gradient of Affine model

Affine Model
$\hat{y} = x * \omega + b$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$



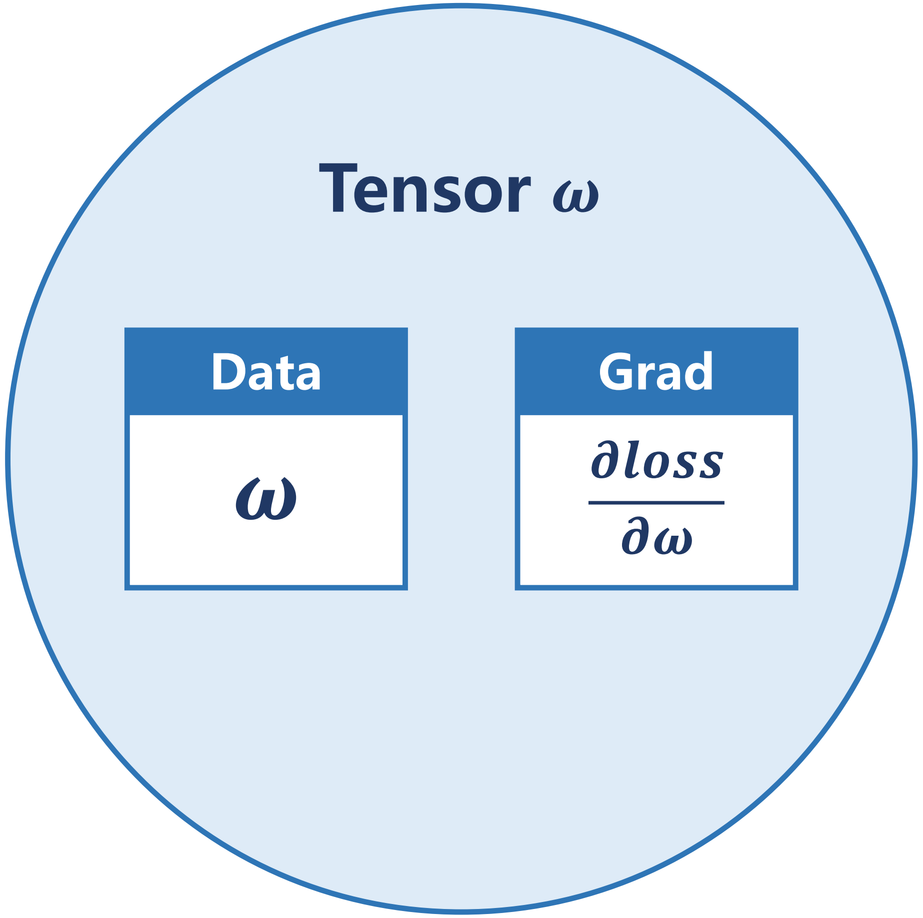
$$\frac{\partial loss}{\partial \omega} = ?$$

$$\frac{\partial loss}{\partial b} = ?$$

Tensor in PyTorch

In PyTorch, **Tensor** is the important component in constructing dynamic computational graph.

It contains **data** and **grad**, which storage the value of node and gradient w.r.t loss respectively.



```python
import torch

x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = torch.Tensor([1.0])
w.requires_grad = True
```

If **autograd mechanics** are required, the element variable **requires_grad** of **Tensor** has to be set to **True**.

Implementation of linear model with PyTorch

```python
def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2
```

Define the linear model:

Linear Model
$\hat{y} = x * \omega$

Implementation of linear model with PyTorch

```
def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2
```

Define the loss function:

Loss Function

$$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$$

Implementation of linear model with PyTorch

```
print("predict (before training)",  4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

Forward, compute the loss.

Implementation of linear model with PyTorch

```
print("predict (before training)",  4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

Backward, compute grad for Tensor whose **requires_grad** set to True

Implementation of linear model with PyTorch

```
print("predict (before training)",  4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

The **grad** is utilized to update weight.

Implementation of linear model with PyTorch

```
print("predict (before training)",  4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).i
```

NOTICE:

The grad computed by *.backward()* will be **accumulated**.

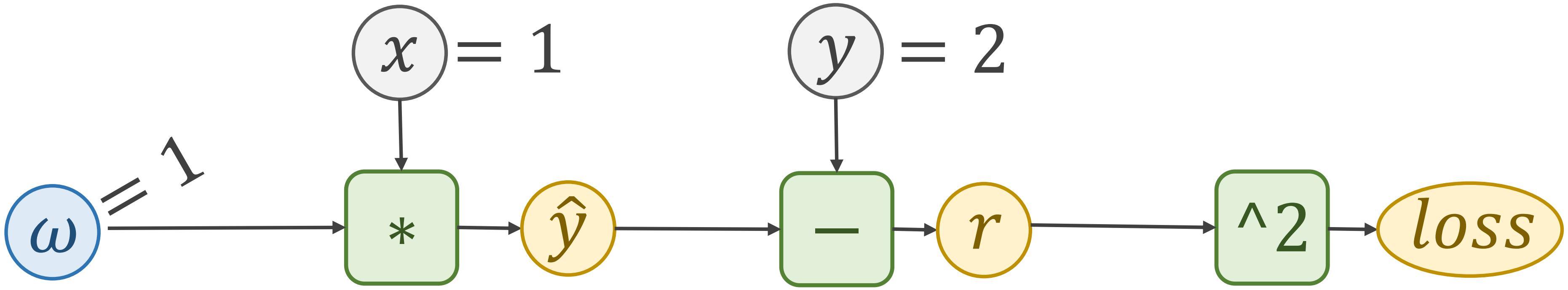
So after update, remember set the grad to **ZERO**!!!

Implementation of linear model with PyTorch

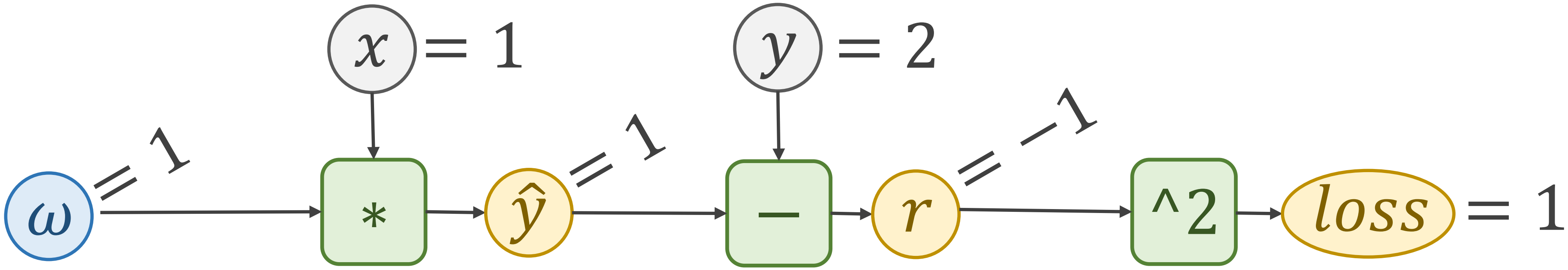
```
print("predict (before training)",  4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data

        w.grad.data.zero_()

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

```
predict (before training) 4 4.0
	grad: 1.0 2.0 -2.0
	grad: 2.0 4.0 -7.840000152587891
	grad: 3.0 6.0 -16.228801727294922
progress: 0 7.315943717956543
	grad: 1.0 2.0 -1.478623867034912
	grad: 2.0 4.0 -5.796205520629883
	grad: 3.0 6.0 -11.998146057128906
progress: 1 3.9987640380859375
	grad: 1.0 2.0 -1.0931644439697266
	grad: 2.0 4.0 -4.285204887390137
	grad: 3.0 6.0 -8.870372772216797
progress: 2 2.1856532096862793
	grad: 1.0 2.0 -0.8081896305084229
	grad: 2.0 4.0 -3.1681032180786133
	grad: 3.0 6.0 -6.557973861694336
progress: 3 1.1946394443511963
	grad: 1.0 2.0 -0.5975041389465332
	grad: 2.0 4.0 -2.3422164916992188
	grad: 3.0 6.0 -4.848389625549316
progress: 4 0.6529689431190491
	grad: 1.0 2.0 -0.4417421817779541
	grad: 2.0 4.0 -1.7316293716430664
	grad: 3.0 6.0 -3.58447265625
progress: 5 0.3569012284278869б
	grad: 1.0 2.0 -0.3265852928161621
	grad: 2.0 4.0 -1.2802143096923828
	grad: 3.0 6.0 -2.650045394897461
```

Forward/Backward in PyTorch

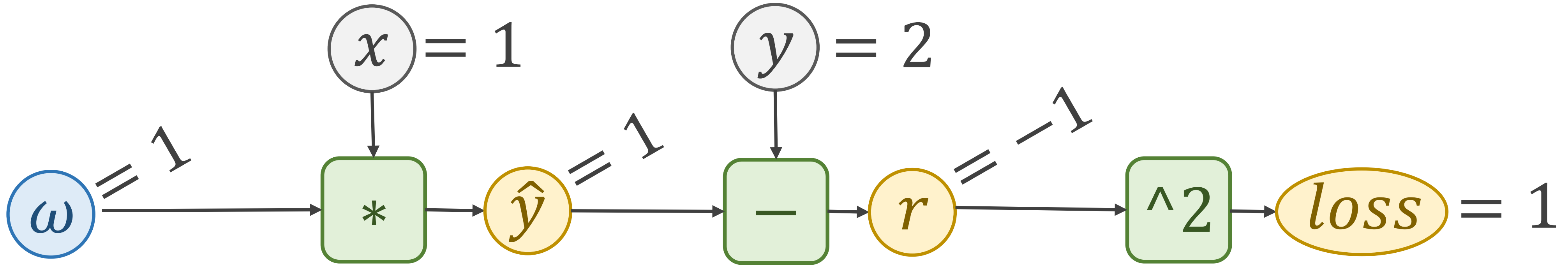


Forward in PyTorch




```
w = torch.Tensor([1.0])
w.requires_grad = True

l = loss(x, y)
```

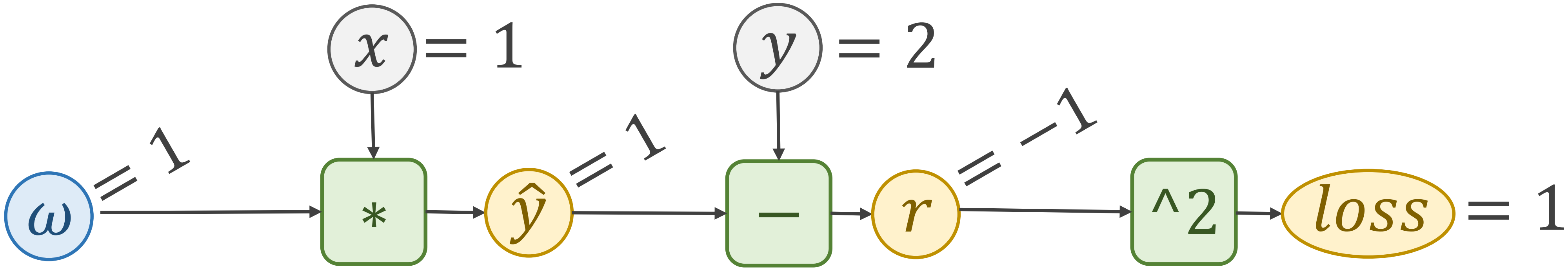
Backward in PyTorch



$l.\mathrm{backward}()$

$$\frac{\partial loss}{\partial \omega} = w.grad$$

Update weight in PyTorch



1. backward()

$$\frac{\partial loss}{\partial \omega} = w.grad$$

```
w.data = w.data - 0.01 * w.grad.data
```

Exercise 4-3: Compute gradients using computational graph

Quadratic Model
$\hat{y} = \omega_1 x^2 + \omega_2 x + b$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$

$$\frac{\partial loss}{\partial \omega_1} = ?$$

$$\frac{\partial loss}{\partial \omega_2} = ?$$

$$\frac{\partial loss}{\partial b} = ?$$

Exercise 4-4: Compute gradients using PyTorch

Quadratic Model
$\hat{y} = \omega_1 x^2 + \omega_2 x + b$

Loss Function
$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$

$$\frac{\partial loss}{\partial \omega_1} = ?$$

$$\frac{\partial loss}{\partial \omega_2} = ?$$

$$\frac{\partial loss}{\partial b} = ?$$

PyTorch Tutorial

04. Back Propagation