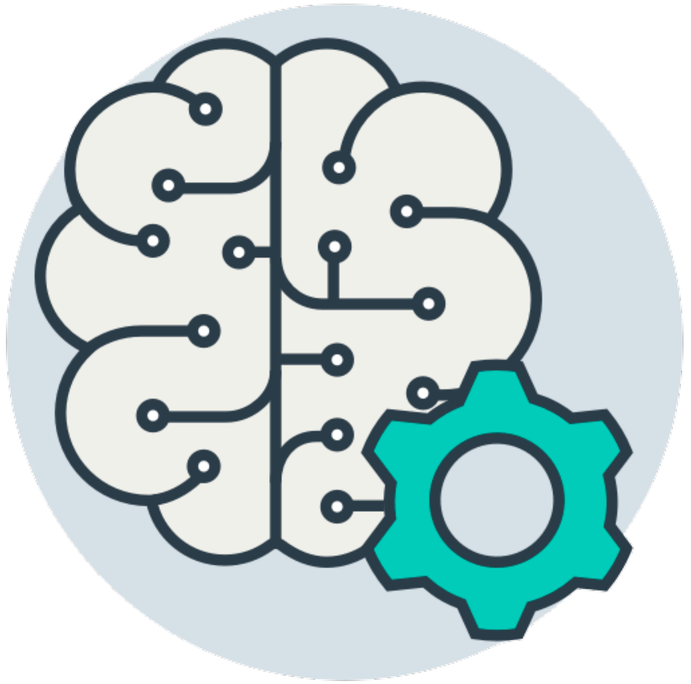
ML/DL for Everyone with PYTORCH

Lecture 1: Introduction

Sung Kim <hunkim+ml@gmail.com> HKUST
Code: https://github.com/hunkim/PyTorchZeroToAll

What is ML?

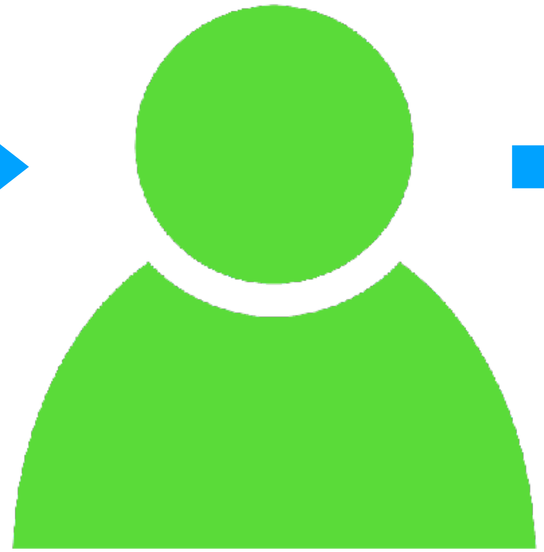


What is Human Intelligence?

What is Human Intelligence?

What to eat for lunch?

Information

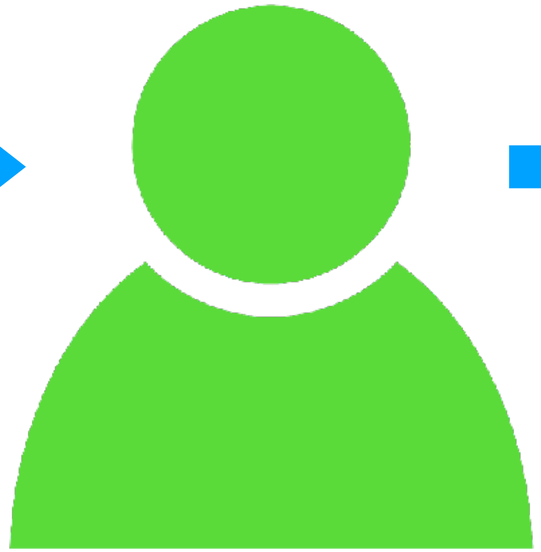


Infer

What is Human Intelligence?

What to dress?

Information

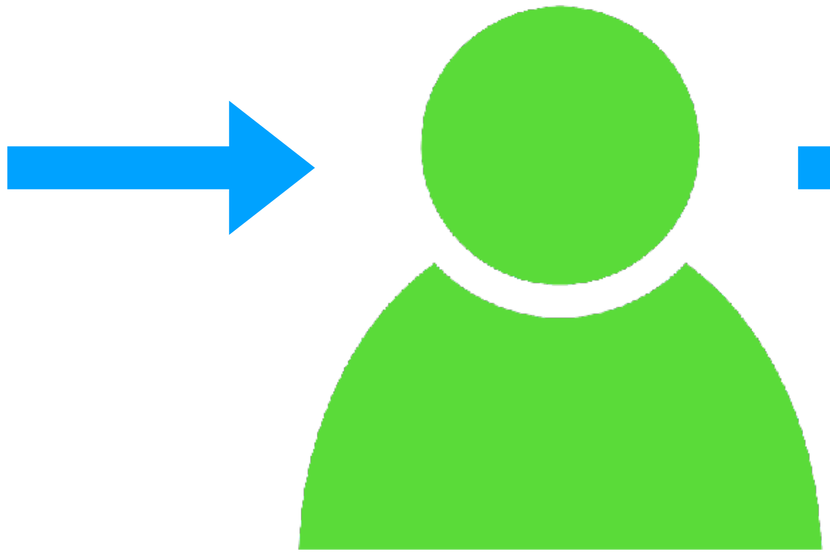


Infer

What is Human Intelligence?

What is this picture?

Image information

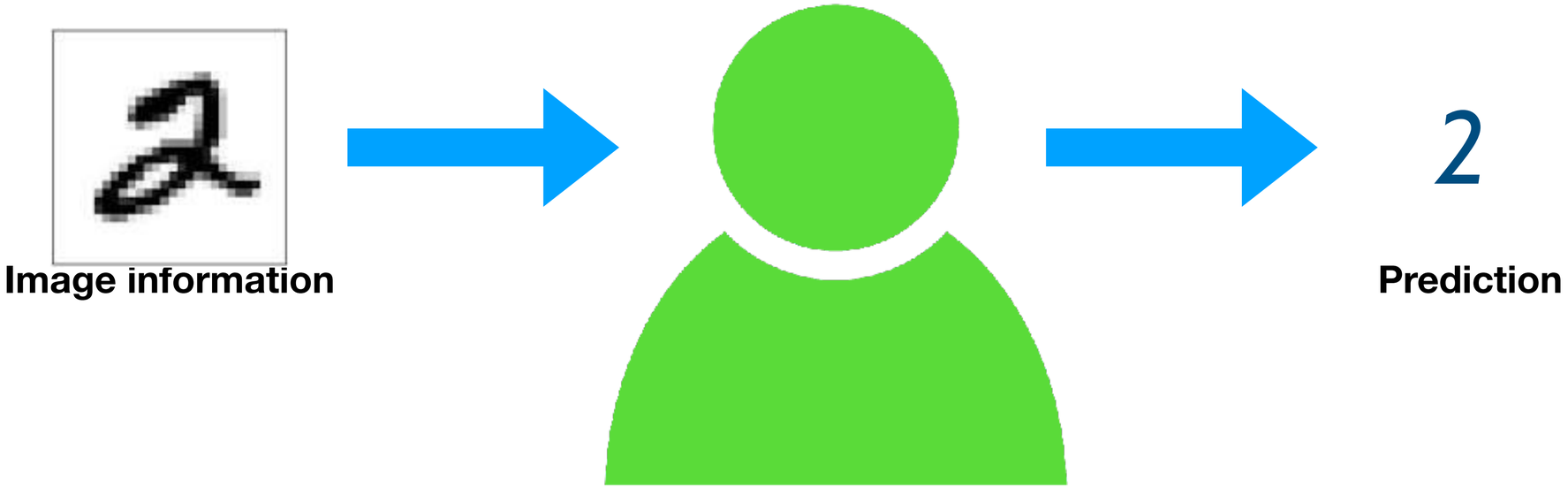


CAT

Prediction

What is Human Intelligence?

What is this number?

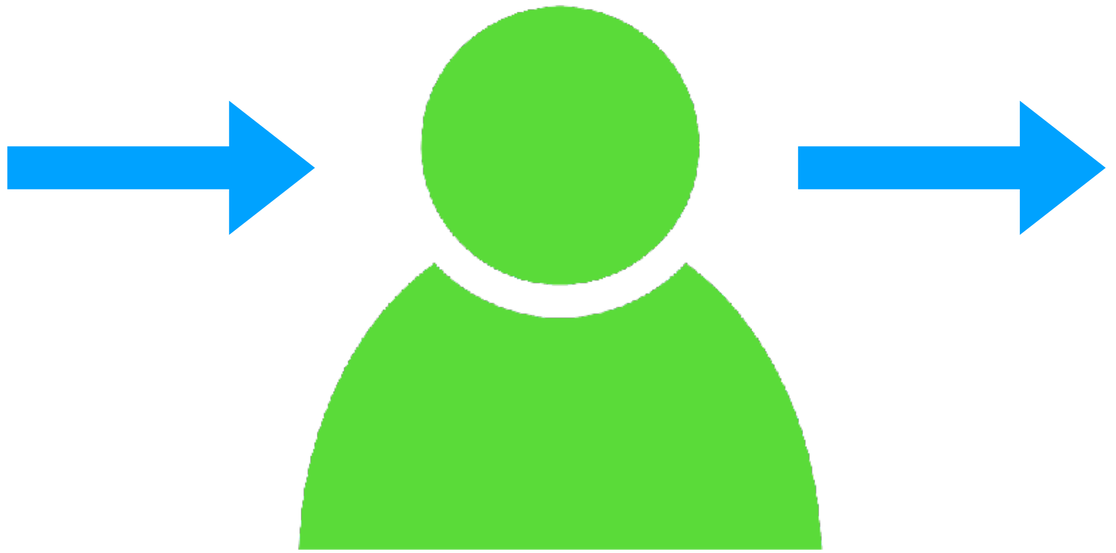


What is Human Intelligence?

What would be the grade if I study only 3 hours?

4 hours

information



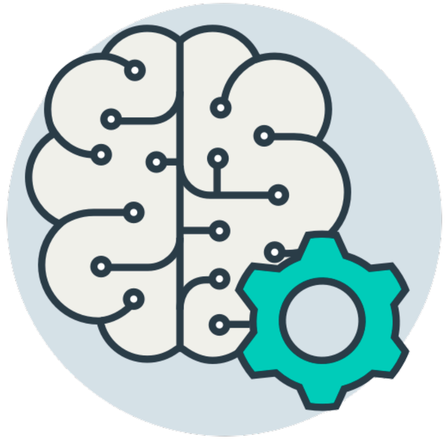
? points

Prediction

Machine Learning

What to dress?

Information

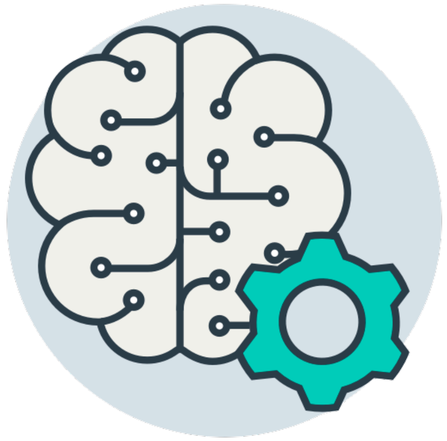


Infer

Machine Learning

What is this picture?

Image information

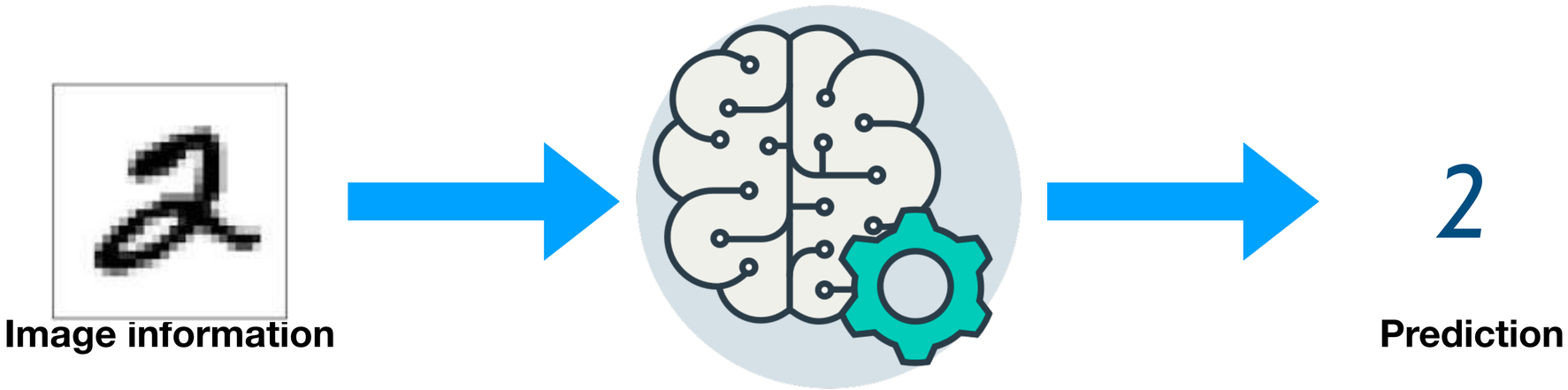


CAT

Prediction

Machine Learning

What is this number?

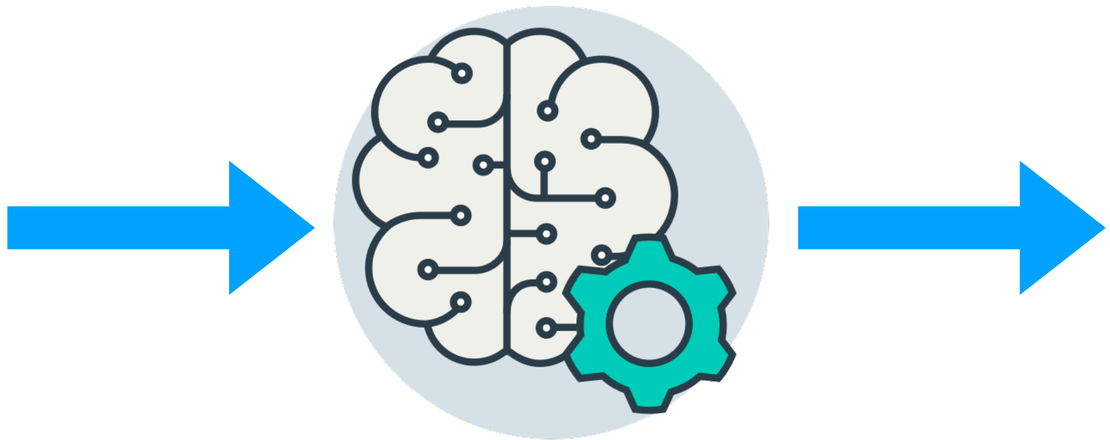


Machine Learning

What would be the grade if I study only 3 hours?

4 hours

information

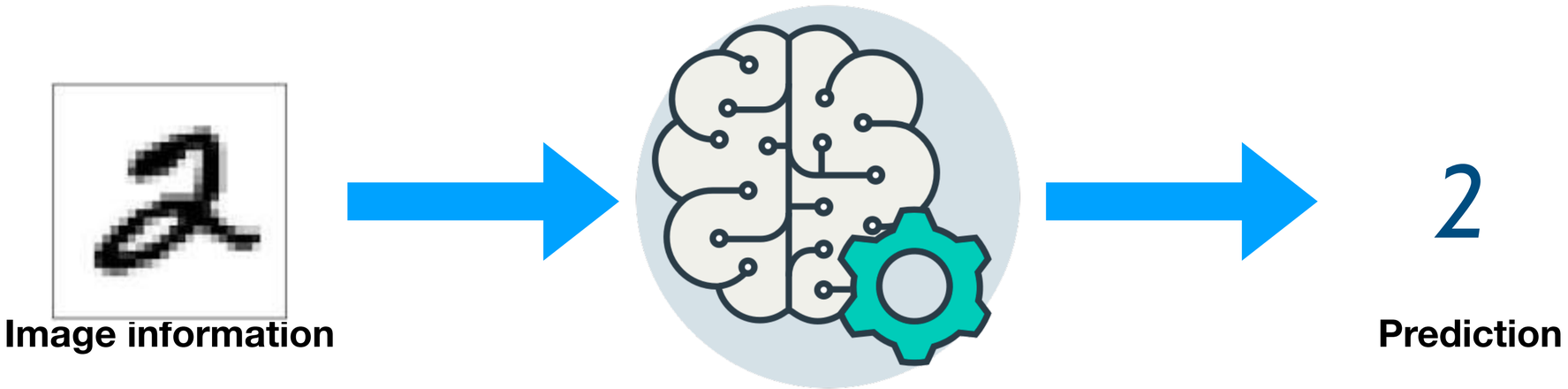


? points

Prediction

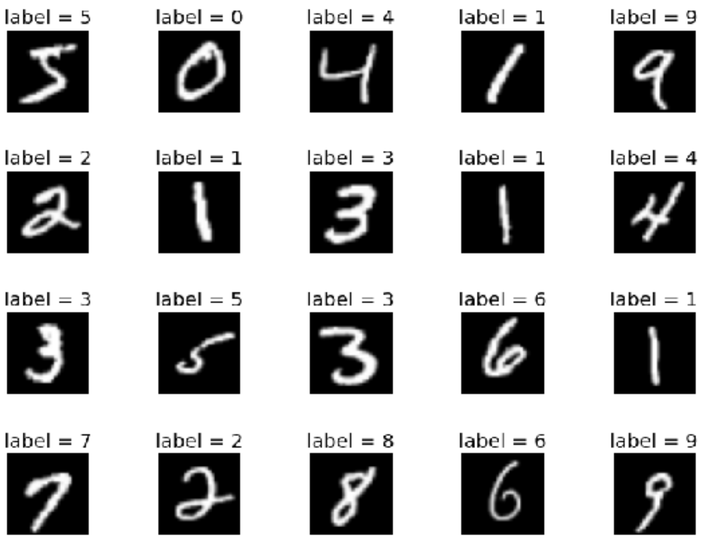
Machine Learning

Machine need lots of training



Machine Learning

Machine need lots of training



Labeled dataset

training

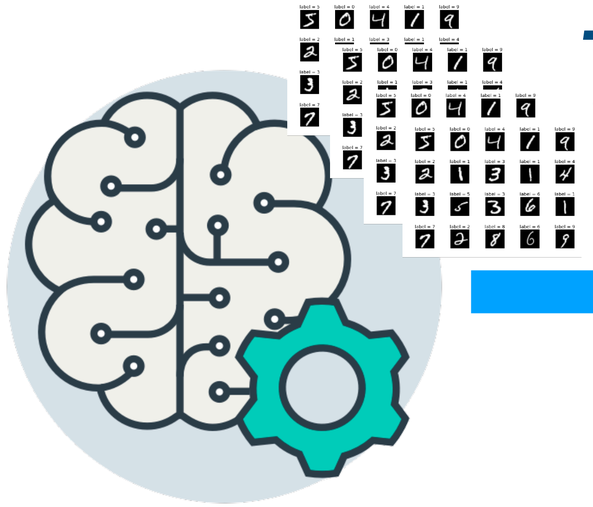
Model

Machine Learning

Predict (test) with trained model

Image information

Test dataset



Trained model

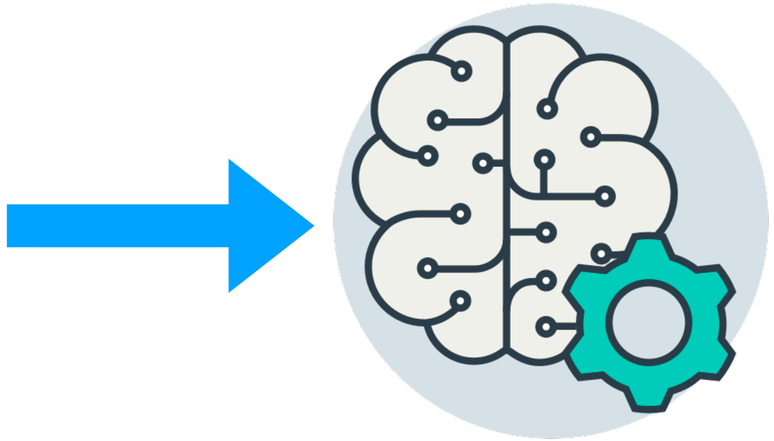
2

Prediction

Machine Learning

What would be the grade if I study only 3 hours?

4 hours



? points

Prediction

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

Training dataset

Test dataset

Lecture 2: Linear Model

ML/DL for Everyone with PYTORCH

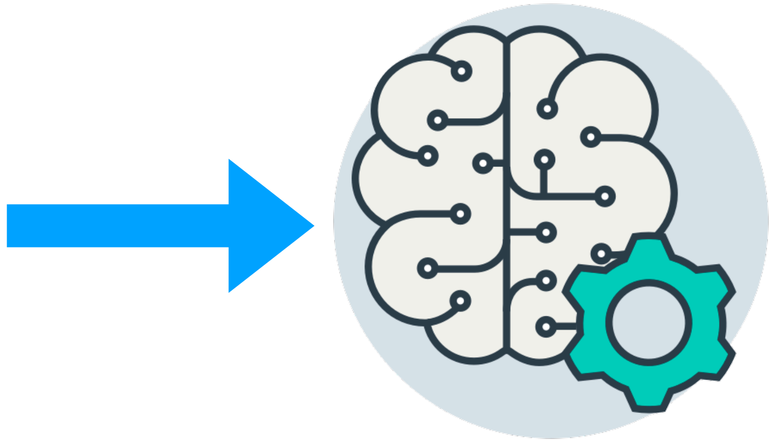
Lecture 2: Linear Model

Sung Kim <hunkim+ml@gmail.com> HKUST
Code: https://github.com/hunkim/PyTorchZeroToAll

Machine Learning

What would be the grade if I study only 3 hours?

4 hours



? points

Prediction

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

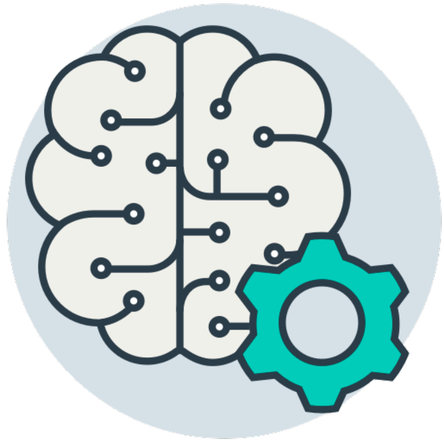
Training dataset

Test dataset

Model design

What would be the best model for the data? Linear?

Hours (x)	Points (y)
1	2
2	4
3	6
4	?

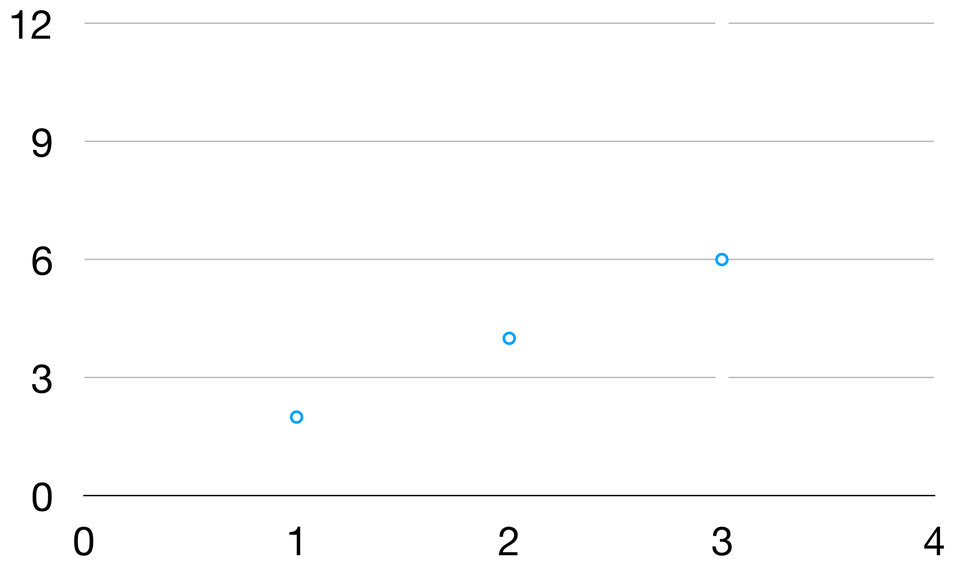


$$\hat{y} = x * w + b$$

Linear Regression

$$\hat{y} = x * w + b \qquad \hat{y} = x * w$$

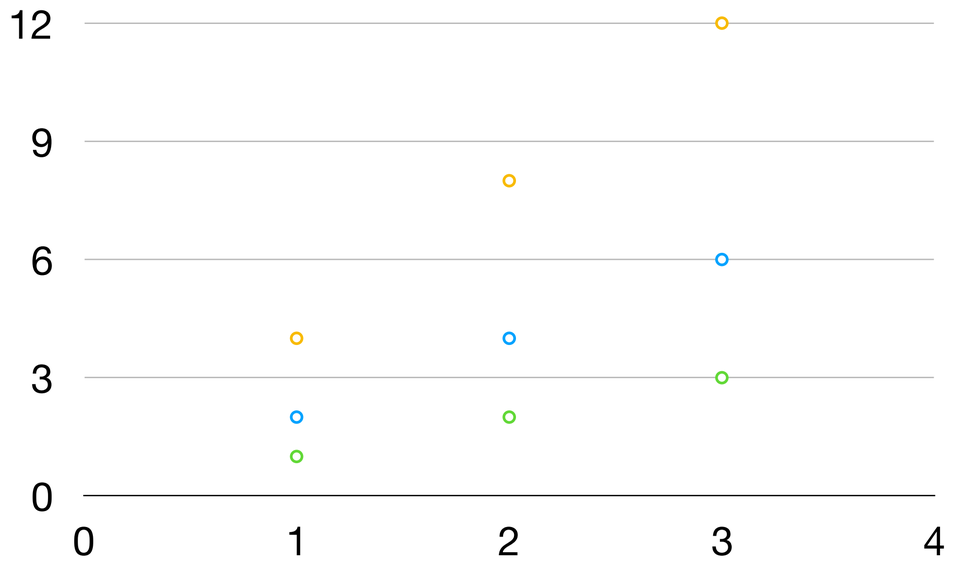
Hours (x)	Points (y)
1	2
2	4
3	6



Linear Regression error?

$$\hat{y} = x * w + b$$

Hours (x)	Points (y)
1	2
2	4
3	6



Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=3)	Loss (w=3)
1	2	3	1
2	4	6	4
3	6	9	9
			mean=14/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=4)	Loss (w=4)
1	2	4	4
2	4	8	16
3	6	12	36
			mean=56/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=0)	Loss (w=0)
1	2	0	4
2	4	0	16
3	6	0	36
			mean=56/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=1)	Loss (w=1)
1	2	1	1
2	4	2	4
3	6	3	9
			mean=14/3

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Points, y	Prediction, y^(w=2)	Loss (w=2)
1	2	0	0
2	4	0	0
3	6	0	0
			mean=0

Training Loss (error)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Hours, x	Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
1	4	1	0	1	4
2	16	4	0	4	16
3	36	9	0	9	36
	mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7

Loss graph

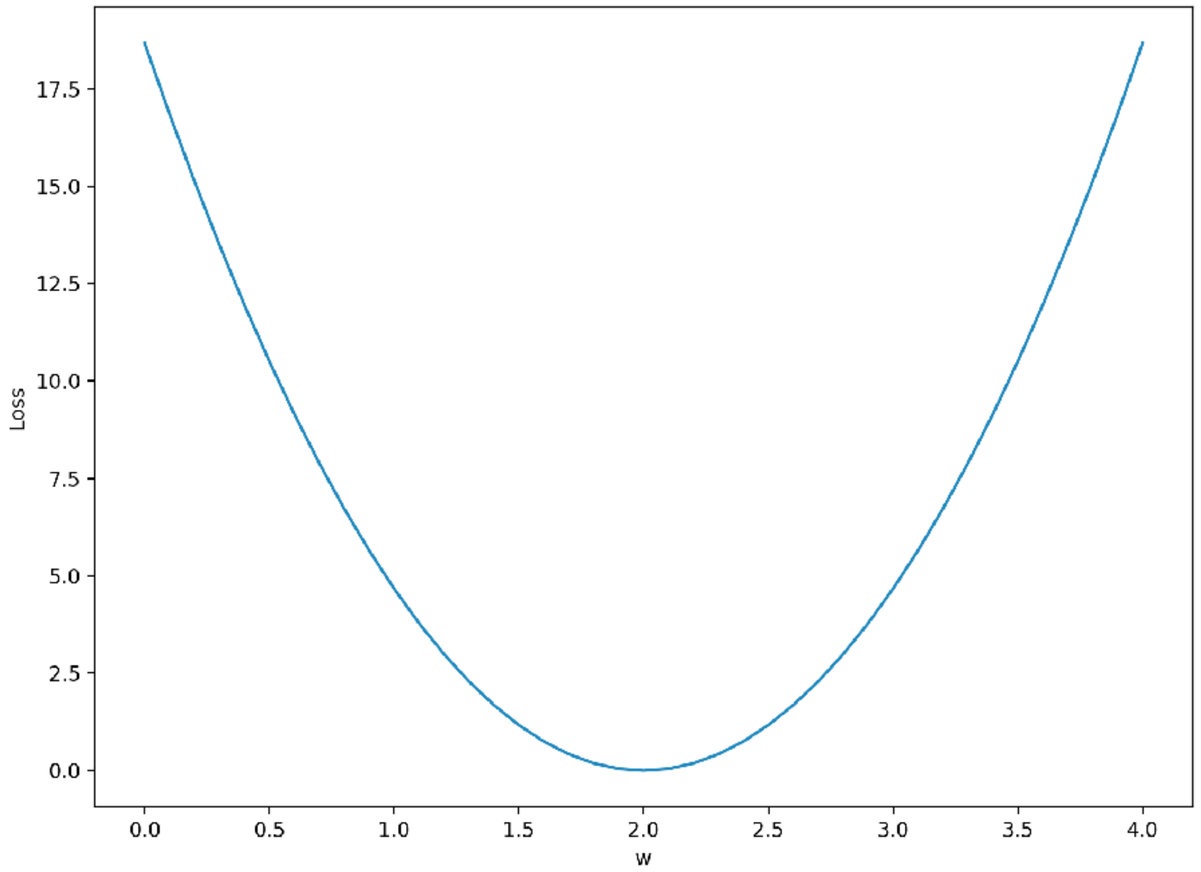
$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7

Loss graph

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7



Model & Loss

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

```
# our model forward pass
def forward(x):
    return x*w
```

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

```
# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred-y)*(y_pred-y)
```

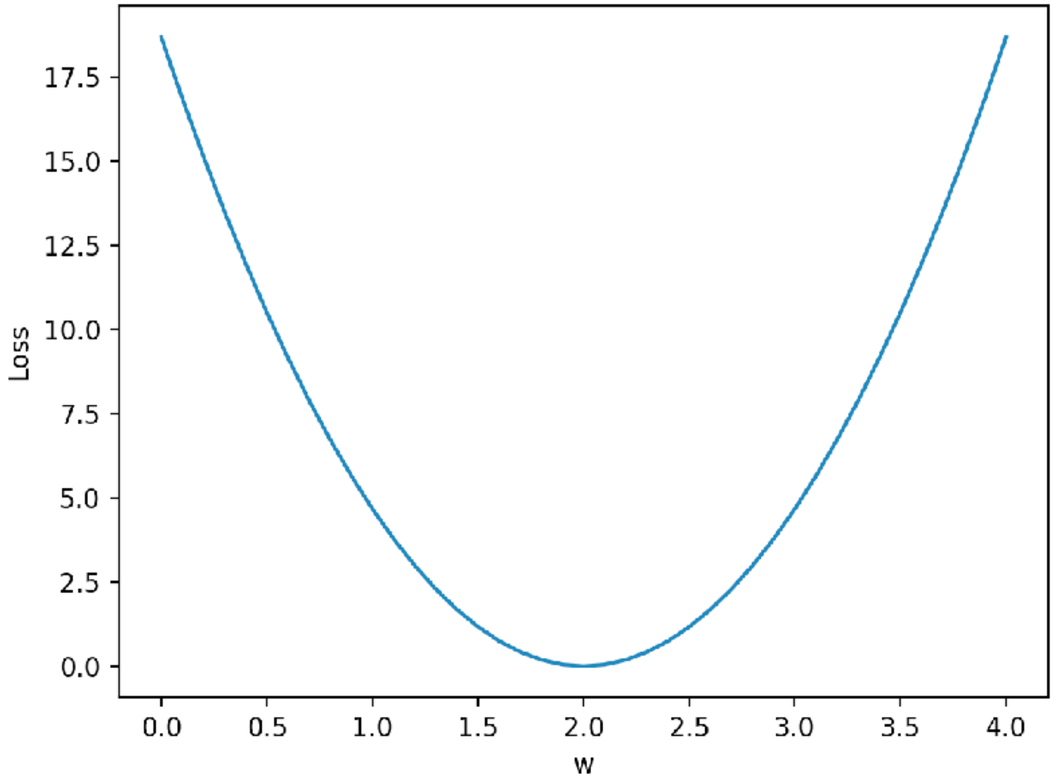
Compute loss for w

```
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred, l)
    print("NSE=", l_sum/3)
```

```
w= 0.0
	 1.0 2.0 0.0 4.0
	 2.0 4.0 0.0 16.0
	 3.0 6.0 0.0 36.0
NSE= 18.6666666667
w= 0.1
	 1.0 2.0 0.1 3.61
	 2.0 4.0 0.2 14.44
	 3.0 6.0 0.3 32.49
NSE= 16.8466666667
w= 0.2
	 1.0 2.0 0.2 3.24
	 2.0 4.0 0.4 12.96
	 3.0 6.0 0.6 29.16
NSE= 15.12
w= 0.3
	 1.0 2.0 0.3 2.89
	 2.0 4.0 0.6 11.56
	 3.0 6.0 0.9 26.01
NSE= 13.4866666667
w= 0.4
	 1.0 2.0 0.4 2.56
	 2.0 4.0 0.8 10.24
	 3.0 6.0 1.2 23.04
NSE= 11.9466666667
w= 0.5
	 1.0 2.0 0.5 2.25
	 2.0 4.0 1.0 9.0
	 3.0 6.0 1.5 20.25
NSE= 10.5
```

Plot graph

```
w_list = []
mse_list = []

for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred, l)
    print("NSE=", l_sum/3)
    w_list.append(w)
    mse_list.append(l_sum/3)

plt.plot(w_list, mse_list)
plt.ylabel('Loss')
plt.xlabel('w')
plt.show()
```



```python
import numpy as np
import matplotlib.pyplot as plt

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


# our model forward pass
def forward(x):
    return x*w


# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred-y)*(y_pred-y)


w_list = []
mse_list = []

for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred, l)
    print("NSE=", l_sum/3)
    w_list.append(w)
    mse_list.append(l_sum/3)

plt.plot(w_list, mse_list)
plt.ylabel('Loss')
plt.xlabel('w')
plt.show()
```

Lecture 3:
Gradient Decent

ML/DL for Everyone with PYTORCH

Lecture 3: Gradient Decent

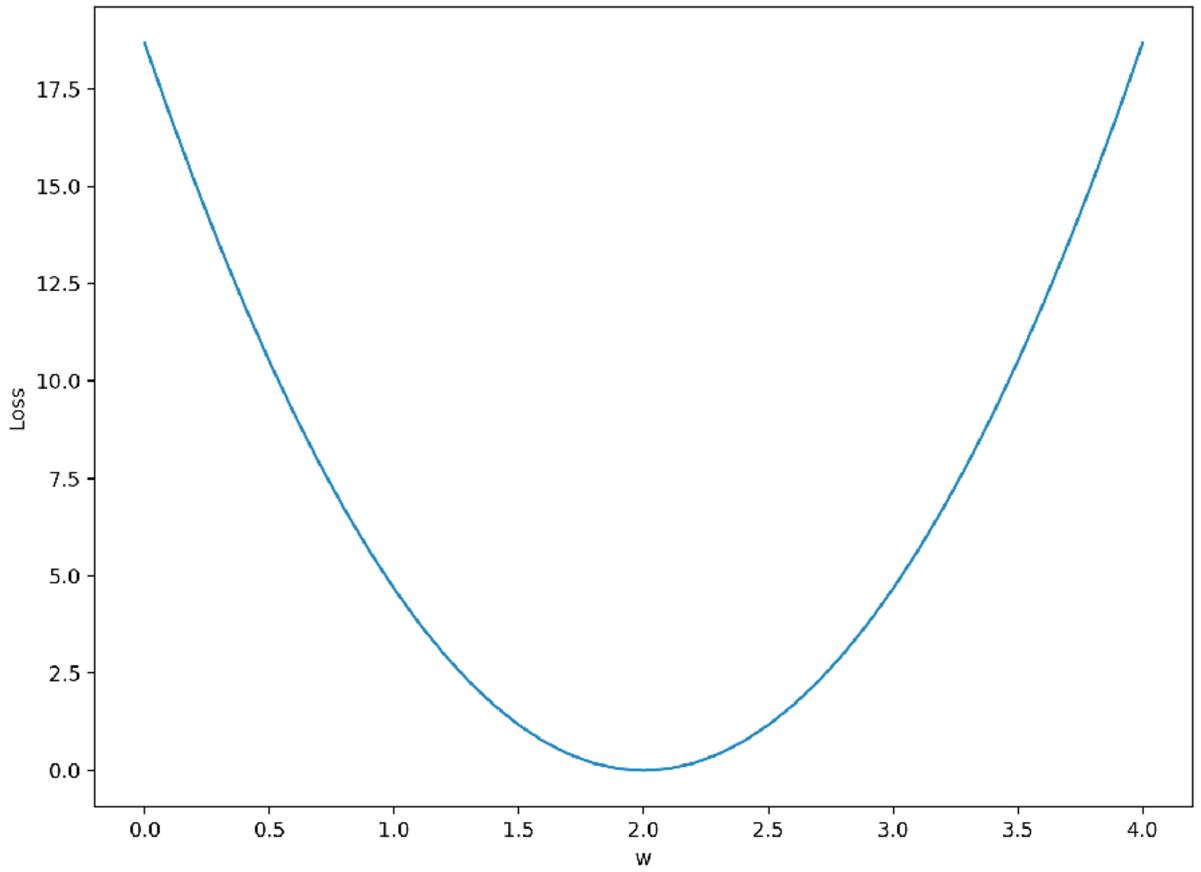
Sung Kim <hunkim+ml@gmail.com> HKUST

Code: https://github.com/hunkim/PyTorchZeroToAll

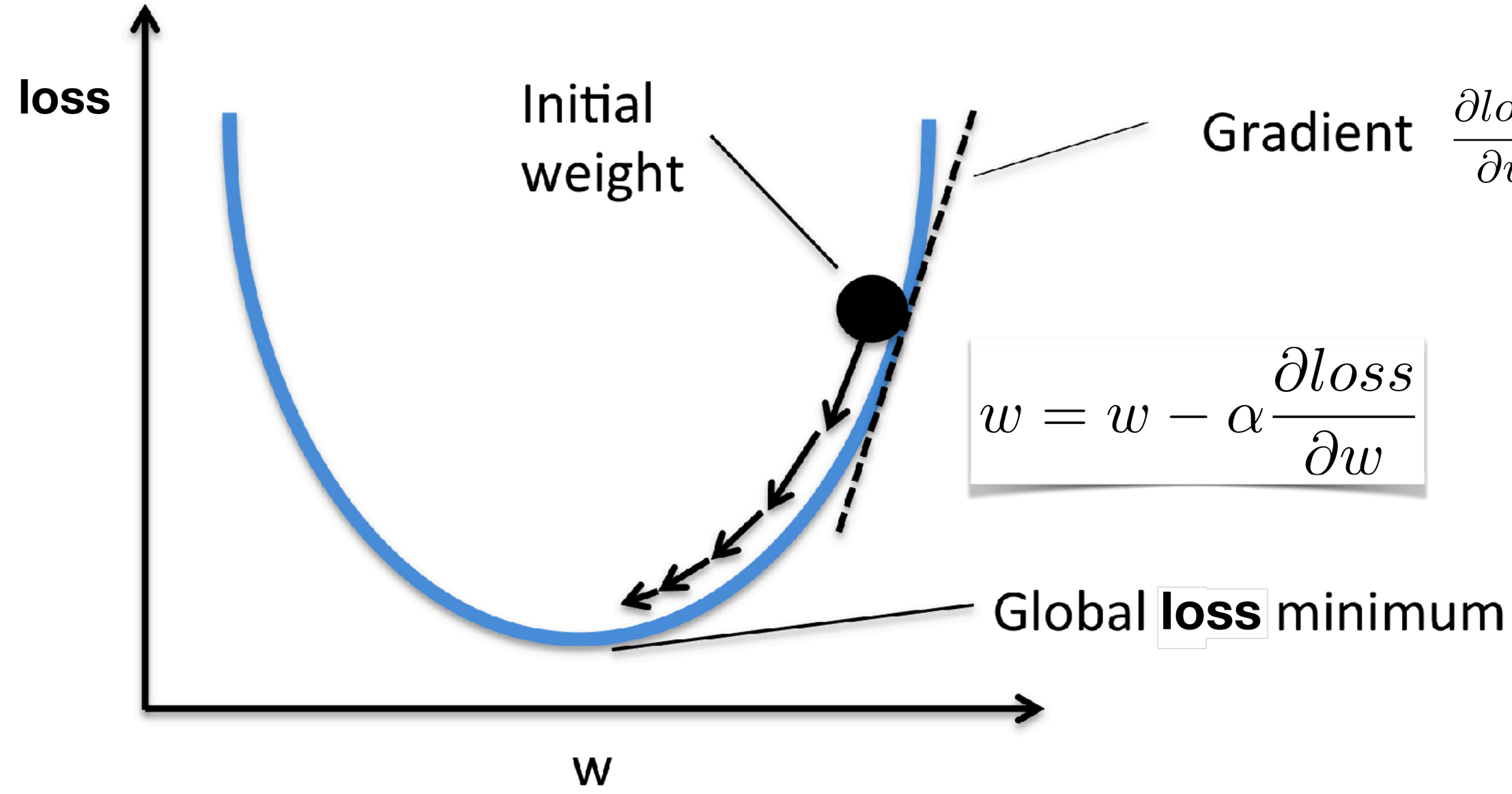
Loss graph

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7



Gradient decent algorithm



Derivative

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

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

Derivative

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

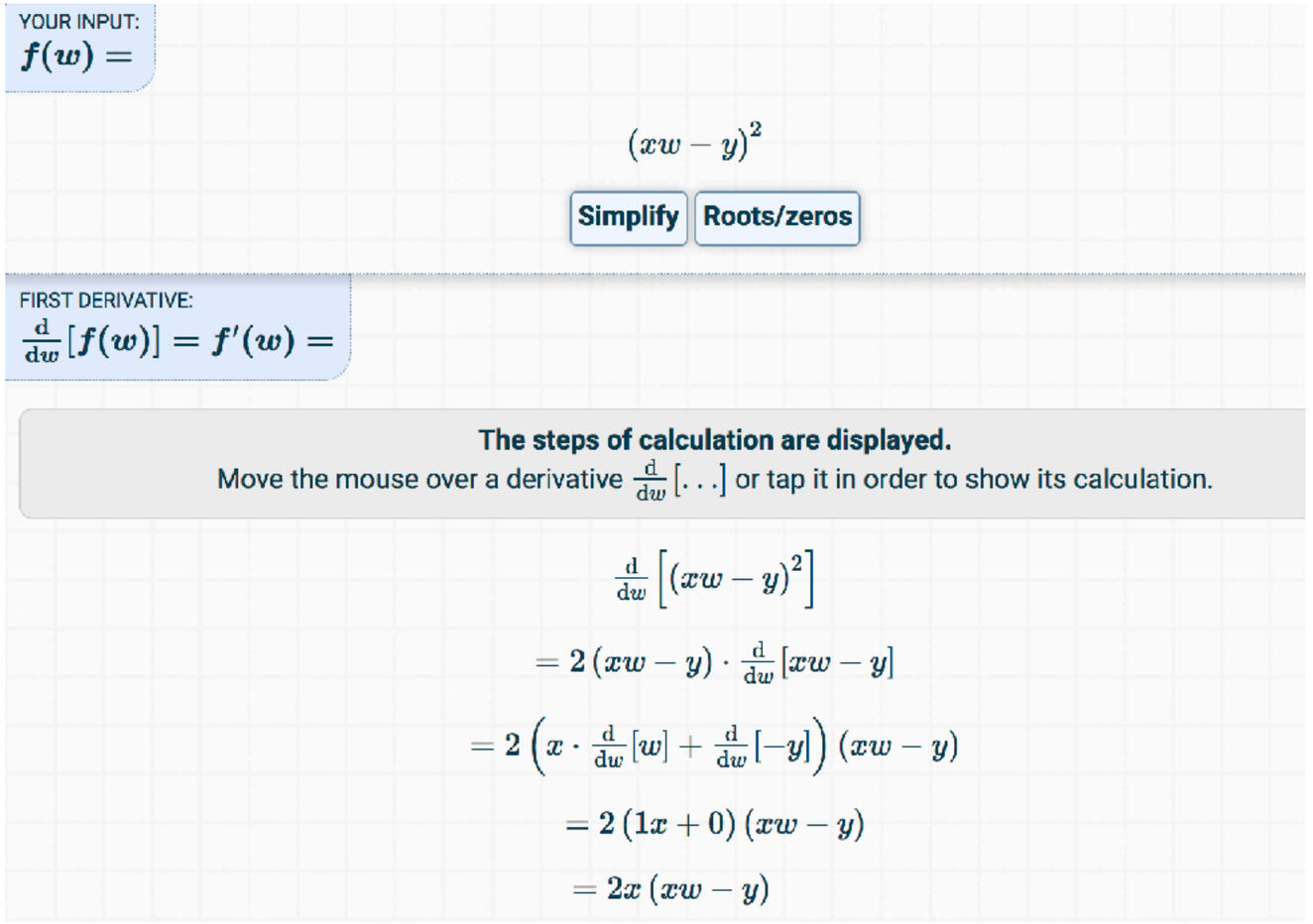
$$w = w - \alpha \frac{\partial loss}{\partial w}$$

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

Derivative

$$loss = (\hat{y} - y)^2 = (x * w - y)^2$$

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



https://www.derivative-calculator.net/

Data, Model, Loss, and Gradient

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

w = 1.0 # any random value


# our model forward pass
def forward(x):
    return x*w


# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred-y)*(y_pred-y)


# compute gradient
def gradient(x, y): # d_loss/d_w
    return 2*x*(x*w-y)
```

Training: updating weight

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

w = 1.0 # any random value


# our model forward pass
def forward(x):
    return x*w


# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred-y)*(y_pred-y)


# compute gradient
def gradient(x, y): # d_loss/d_w
    return 2*x*(x*w-y)
```

```
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print ("progress:", epoch, l)

# After training
print("predict (after training)",  4, forward(4))
```

```
predict (before training) 4 4.0
     grad:  1.0 2.0 -2.0
     grad:  2.0 4.0 -7.84
     grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
     grad:  1.0 2.0 -1.478624
     grad:  2.0 4.0 -5.796206079999999
     grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
     grad:  1.0 2.0 -1.093164466688
     grad:  2.0 4.0 -4.285204709416961
     grad:  3.0 6.0 -8.87037374849311
progress: 2 1.4696334962911515
     grad:  1.0 2.0 -0.8081896081960389
     grad:  2.0 4.0 -3.1681032641284723
     grad:  3.0 6.0 -6.557973756745939
progress: 3 0.8032755585999681
     grad:  1.0 2.0 -0.59750427561463
     grad:  2.0 4.0 -2.3422167604093502
     grad:  3.0 6.0 -4.848388694047353
progress: 4 0.43905614881022015
     grad:  1.0 2.0 -0.44174208101320334
     grad:  2.0 4.0 -1.7316289575717576
     grad:  3.0 6.0 -3.584471942173538
progress: 5 0.2399802903801062
     grad:  1.0 2.0 -0.3265852213980338
     grad:  2.0 4.0 -1.2802140678802925
     grad:  3.0 6.0 -2.650043120512205
progress: 6 0.1311689630744999
     grad:  1.0 2.0 -0.241448373202223
     grad:  2.0 4.0 -0.946477622952715
     grad:  3.0 6.0 -1.9592086795121197
progress: 7 0.07169462478267678
     grad:  1.0 2.0 -0.17850567968888198
     grad:  2.0 4.0 -0.6997422643804168
     grad:  3.0 6.0 -1.4484664872674653
progress: 8 0.03918700813247573
     grad:  1.0 2.0 -0.13197139106214673
     grad:  2.0 4.0 -0.5173278529636143
     grad:  3.0 6.0 -1.0708686556346834
progress: 9 0.021418922423117836
predict (after training) 4 7.804863933862125
```

Output

(from gradient numeric computation)

```python
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print ("progress:", epoch, l)

# After training
print("predict (after training)",  4, forward(4))
```

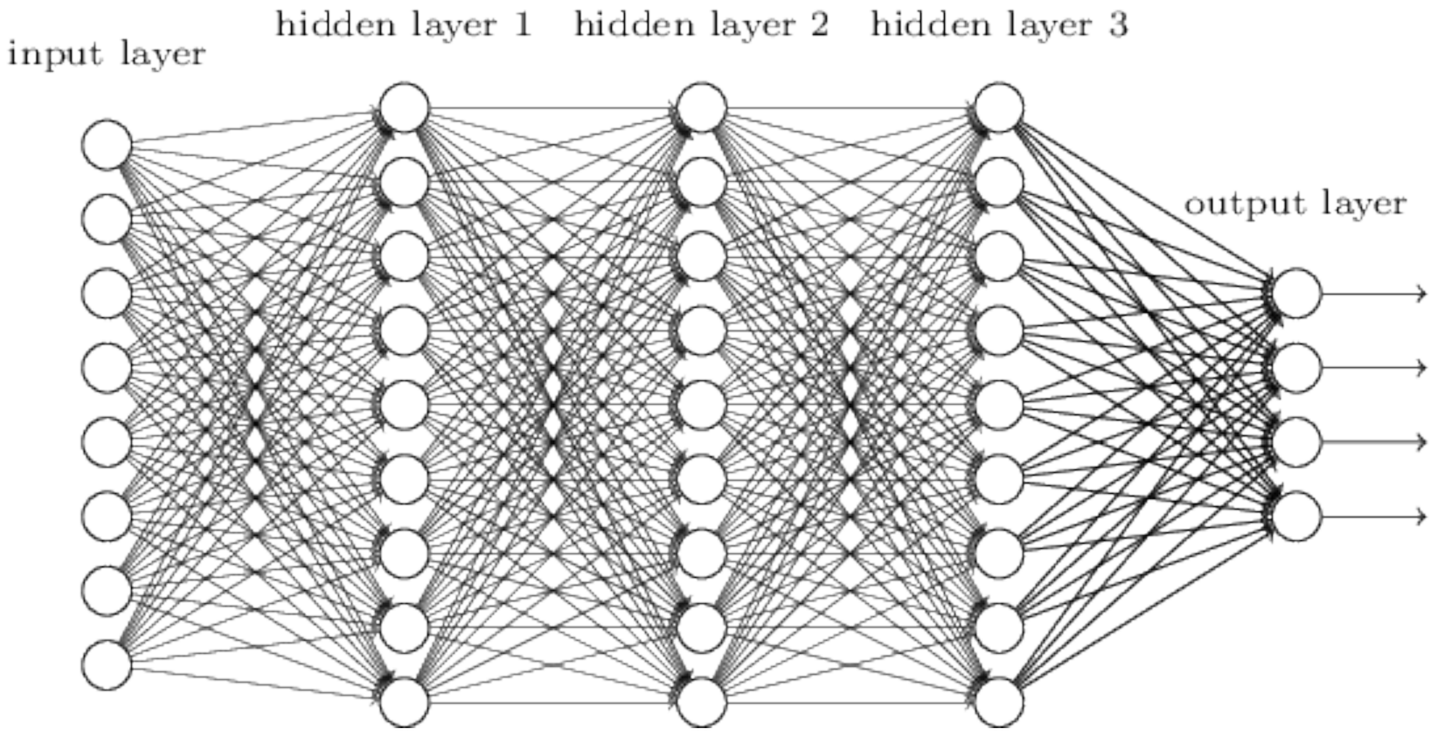
Lecture 4: Back-propagation

ML/DL for Everyone with PYTORCH

Lecture 4: Back-propagation

Sung Kim <hunkim+ml@gmail.com> HKUST
Code: https://github.com/hunkim/PyTorchZeroToAll

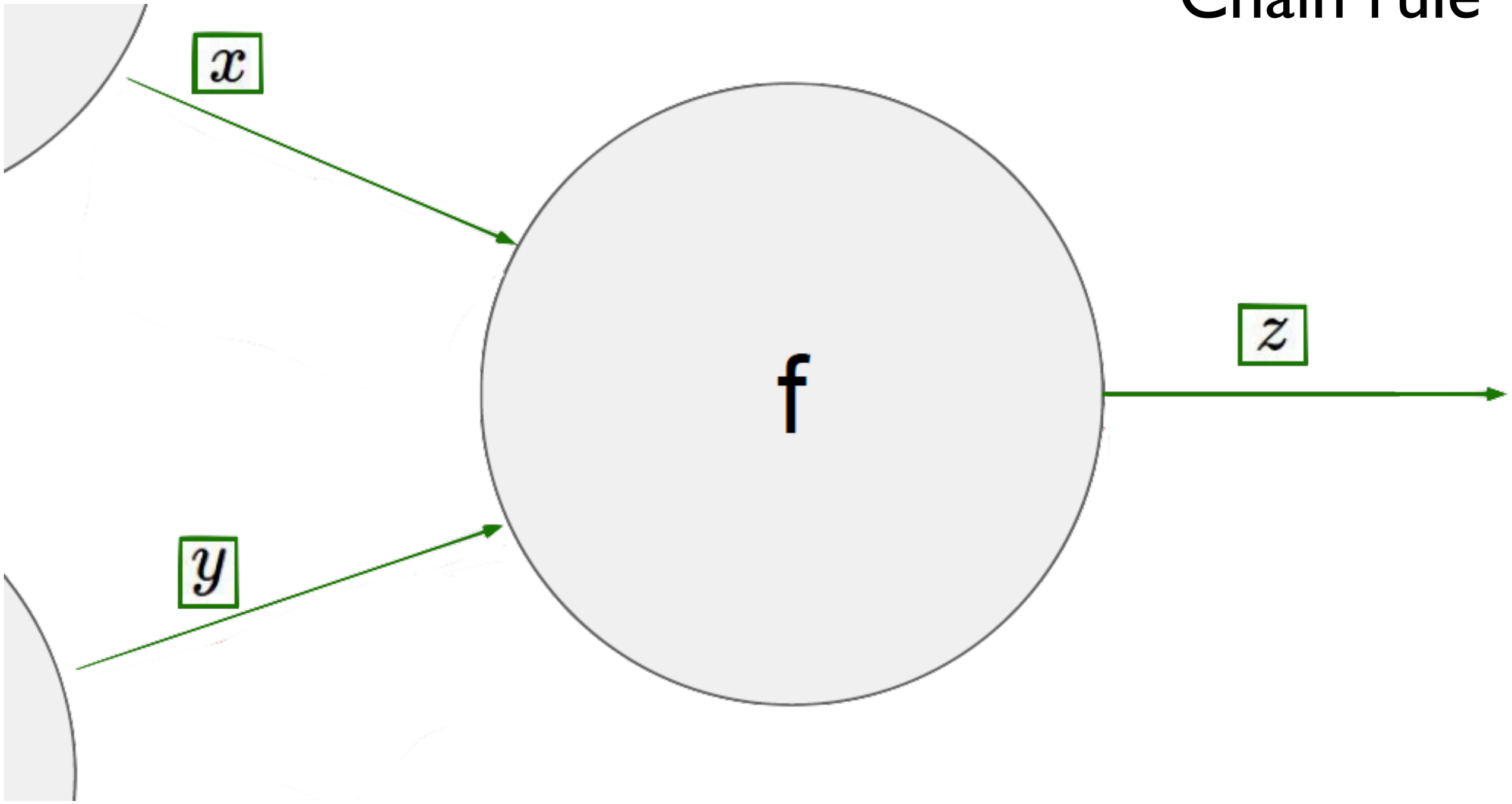
Complicated network?



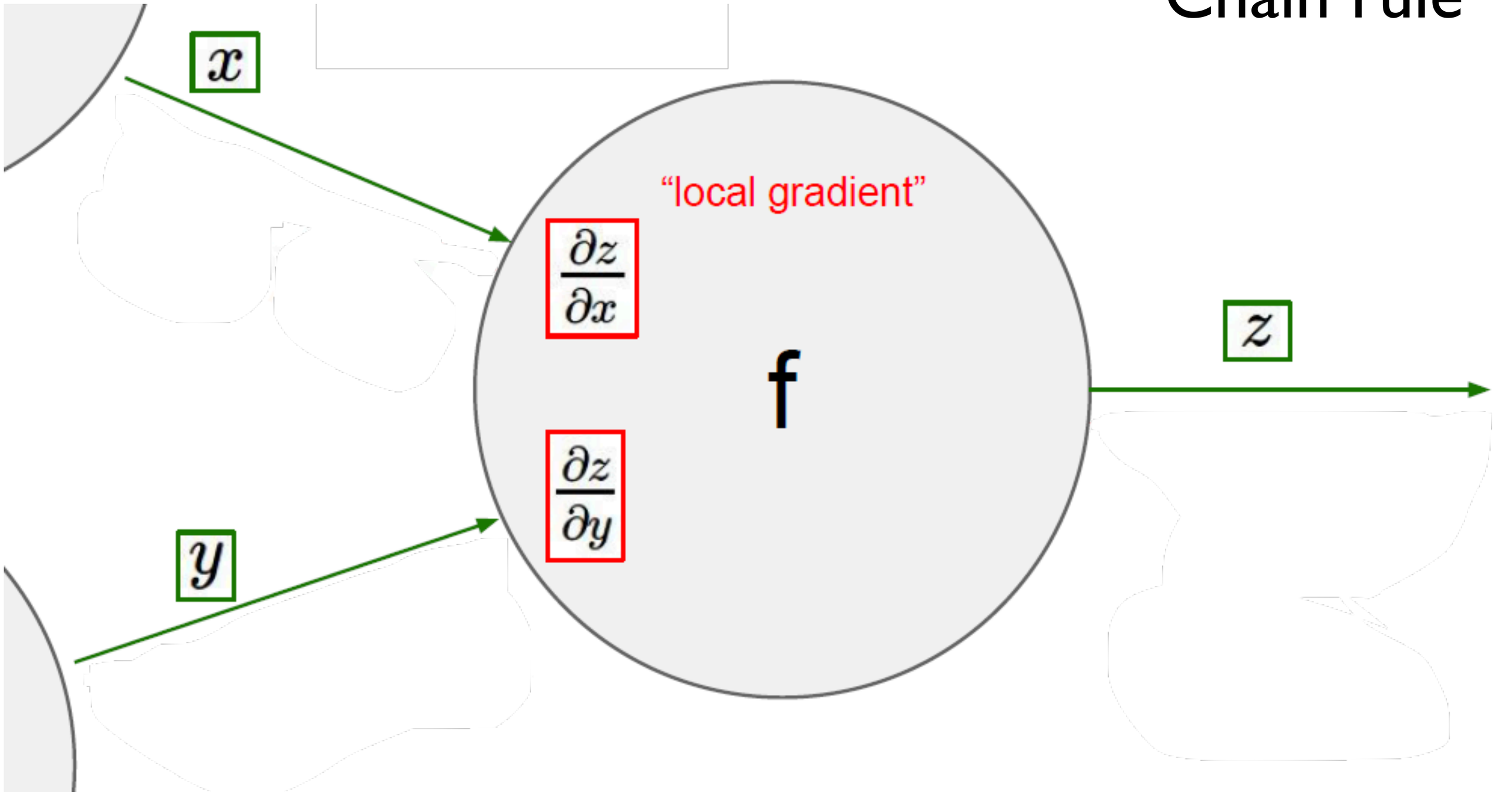
$$\frac{\partial loss}{\partial w} = ?$$

Better way? Computational graph + chain rule

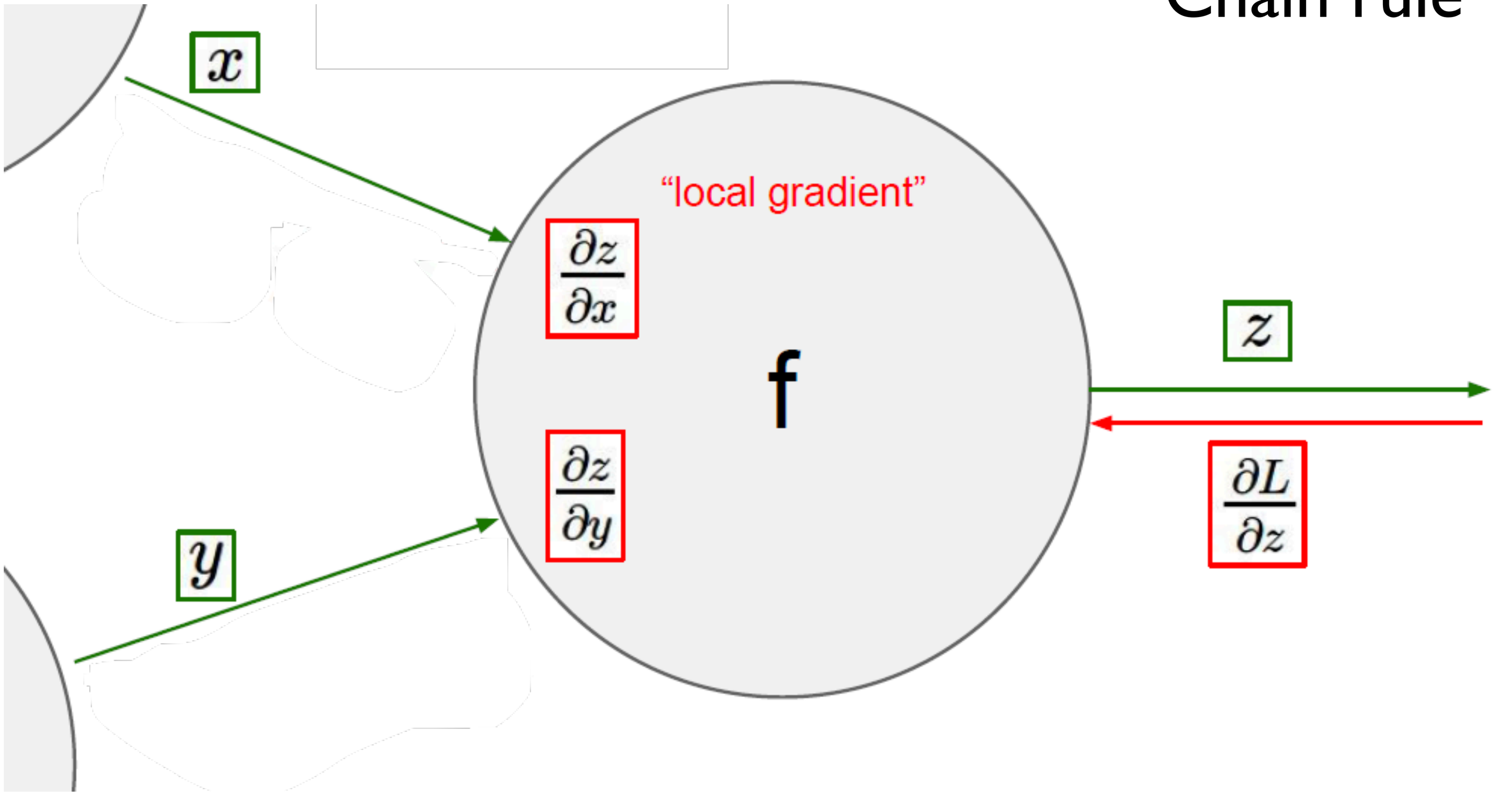
Chain rule



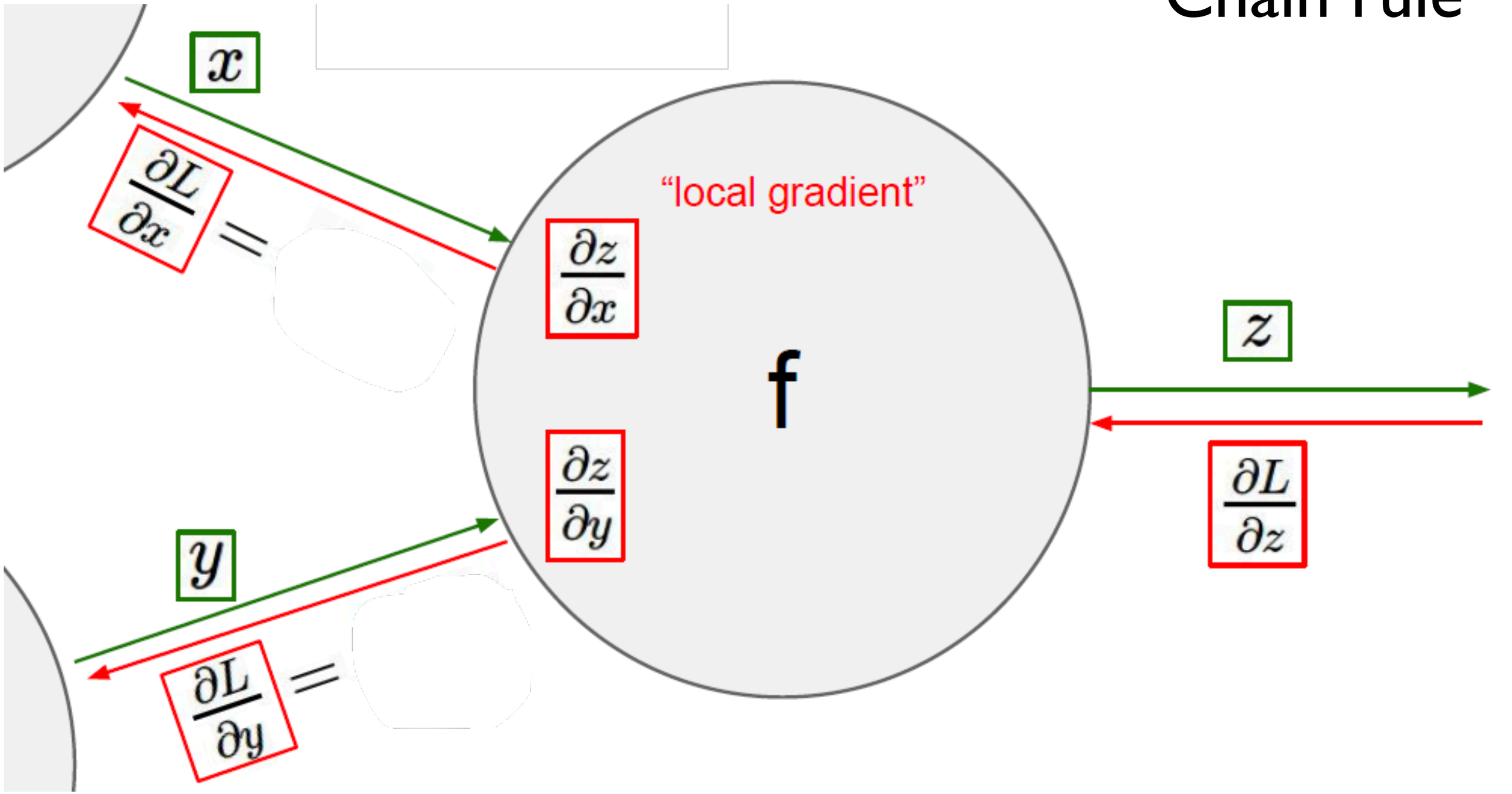
Chain rule


x
y
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z

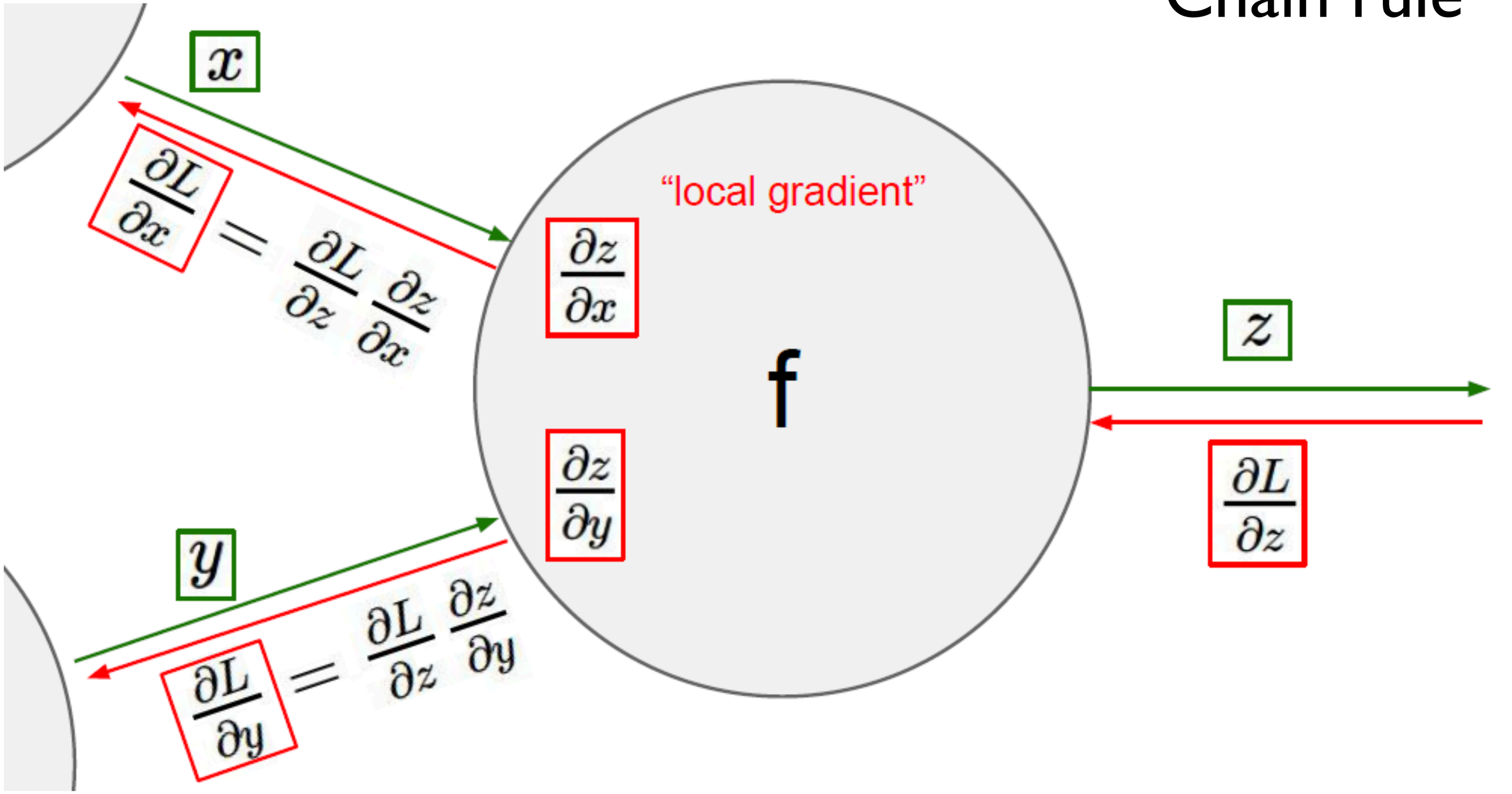
Chain rule


x
y
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z
$\frac{\partial L}{\partial z}$

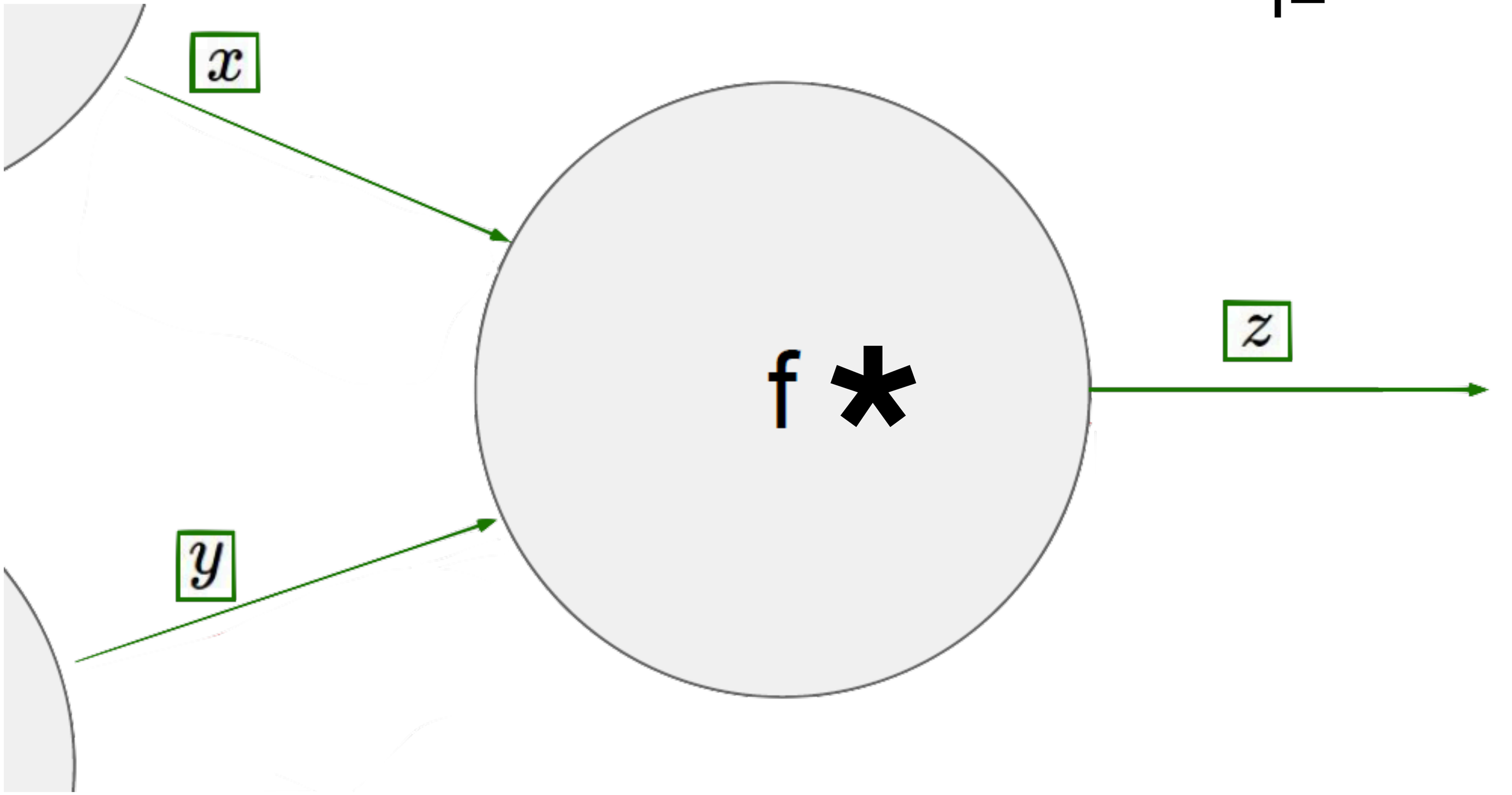
Chain rule

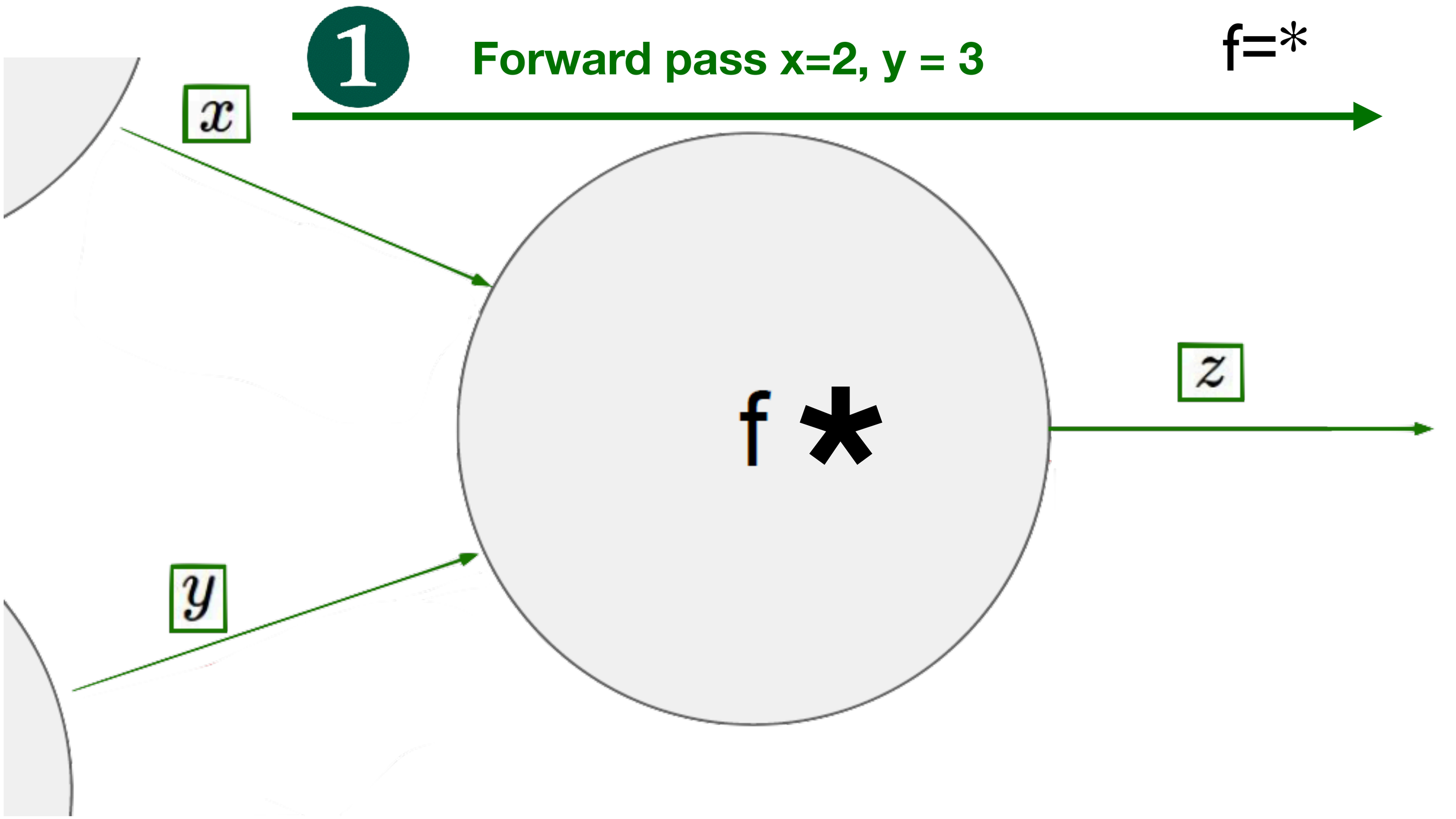

x
$\frac{\partial L}{\partial x}$ =
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y}$ =

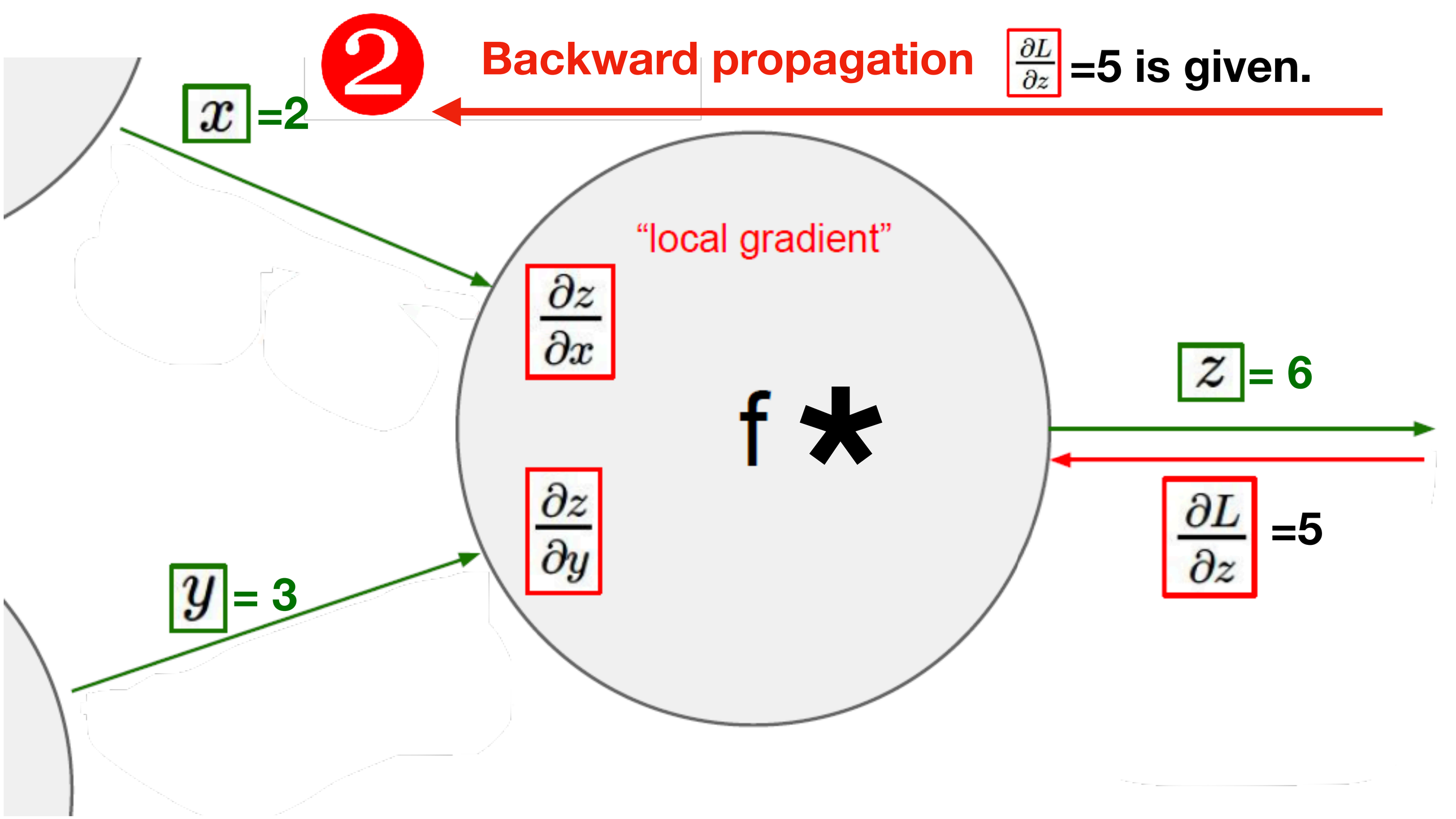
Chain rule

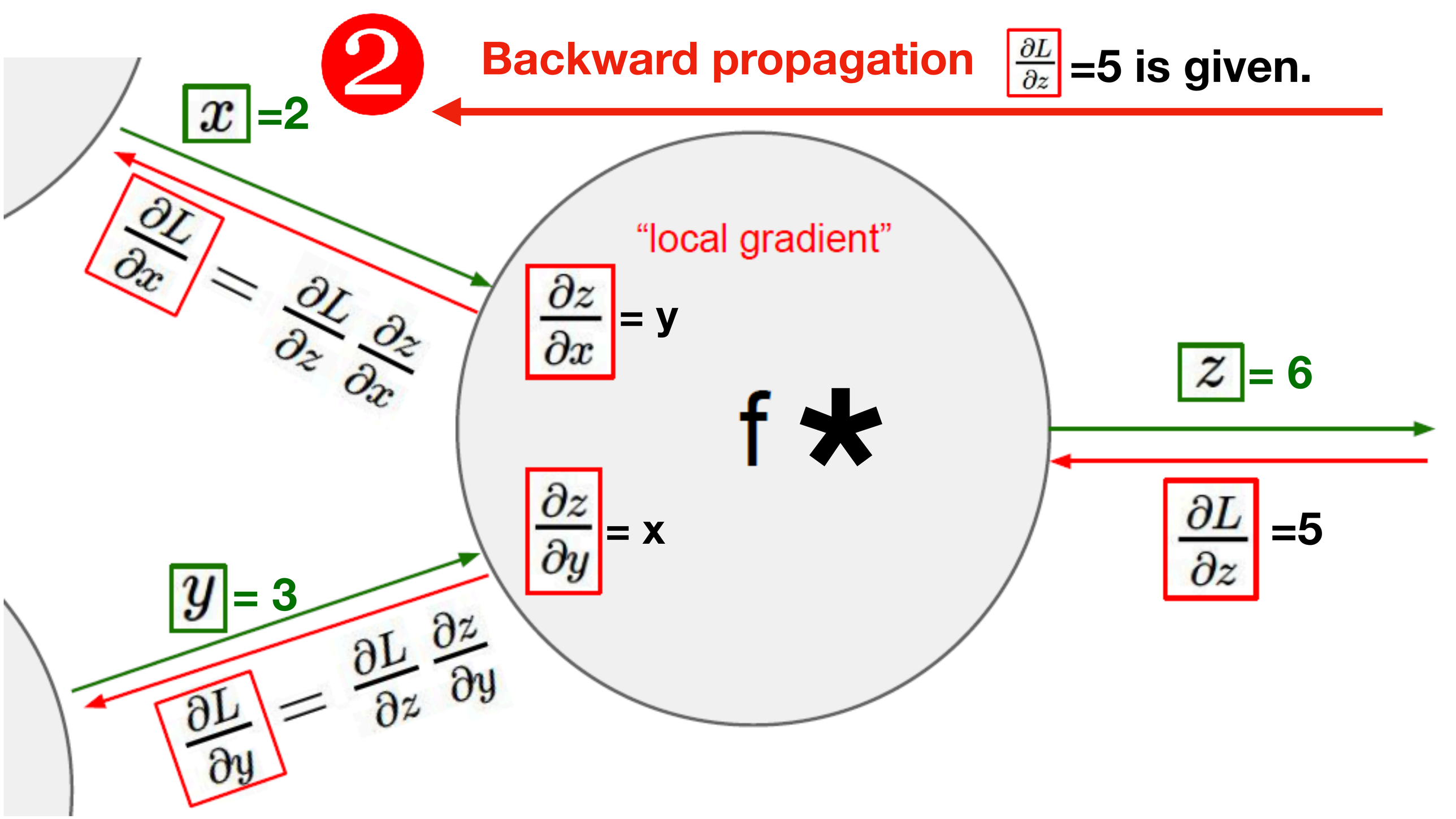

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

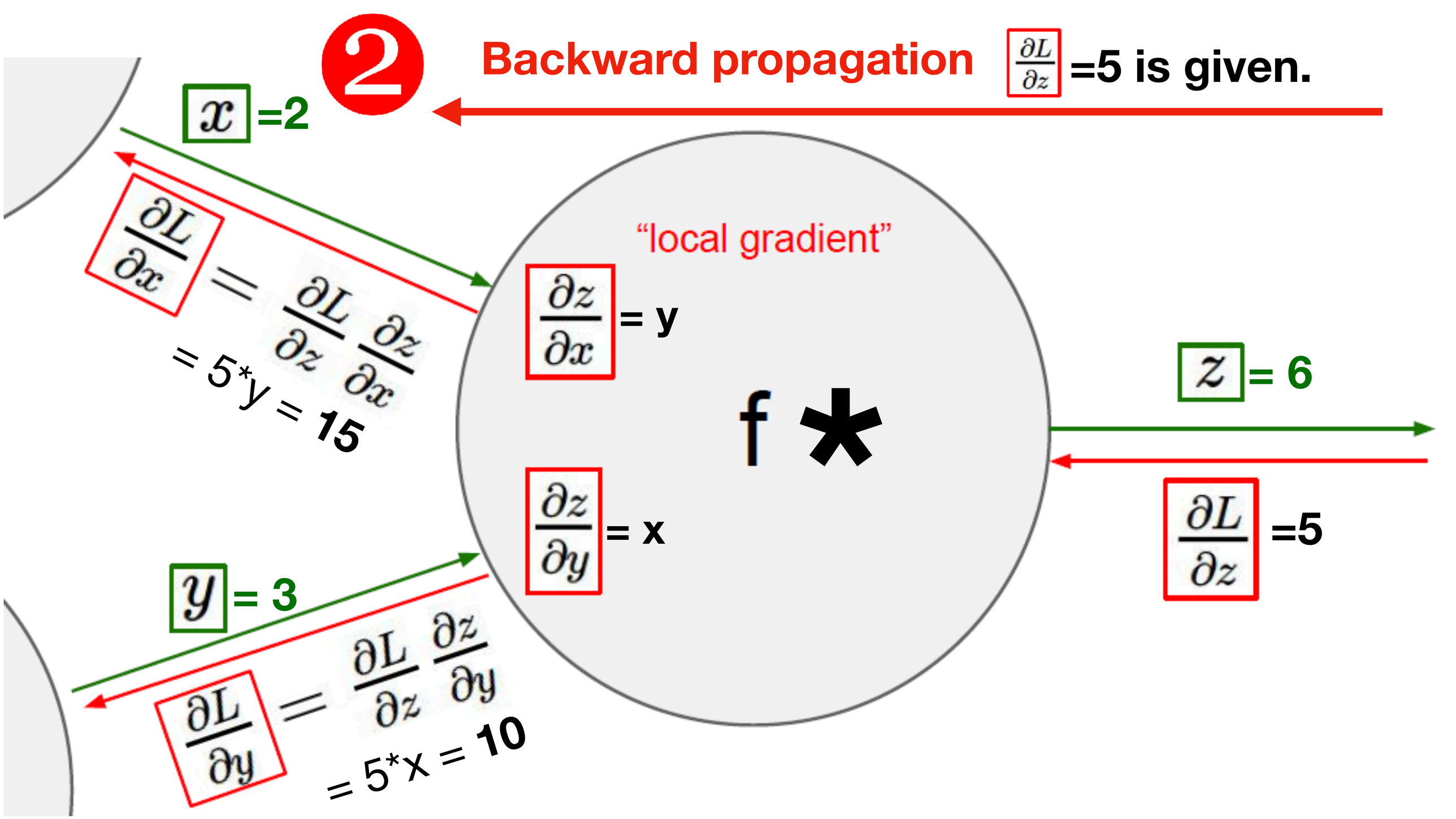
f=*




1
Forward pass x=2, y = 3
f=*
x
y
f *
z


2
Backward propagation
$\frac{\partial L}{\partial z}$ =5 is given.
x =2
"local gradient"
$\frac{\partial z}{\partial x}$
f *
$\frac{\partial z}{\partial y}$
y = 3
z = 6
$\frac{\partial L}{\partial z}$ =5


2
Backward propagation
$\frac{\partial L}{\partial z}$ =5 is given.
x =2
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$ = y
f *
$\frac{\partial z}{\partial y}$ = x
z = 6
$\frac{\partial L}{\partial z}$ =5
y = 3
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

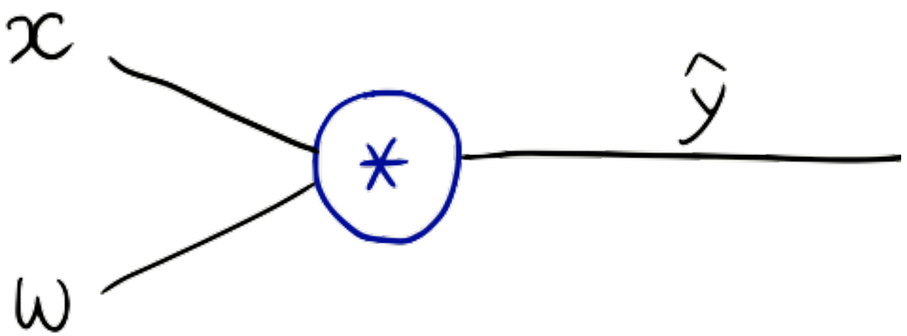

2
Backward propagation
$\frac{\partial L}{\partial z}$ =5 is given.
x =2
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
= 5*y = 15
"local gradient"
$\frac{\partial z}{\partial x}$ = y
f *
$\frac{\partial z}{\partial y}$ = x
z = 6
$\frac{\partial L}{\partial z}$ =5
y = 3
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
= 5*x = 10

Computational graph + chain rule

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

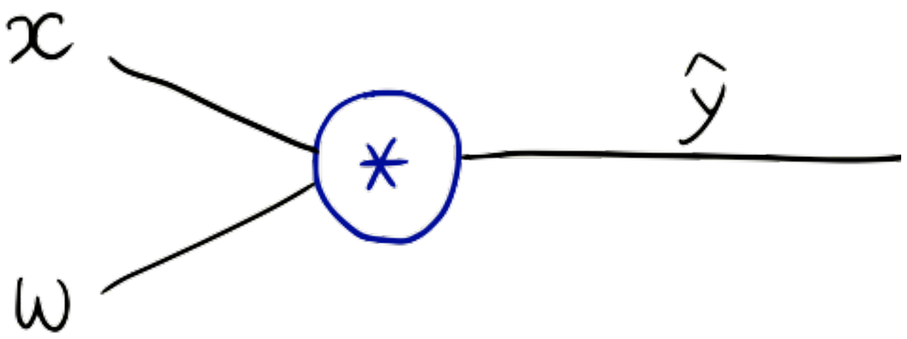
Computational graph

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



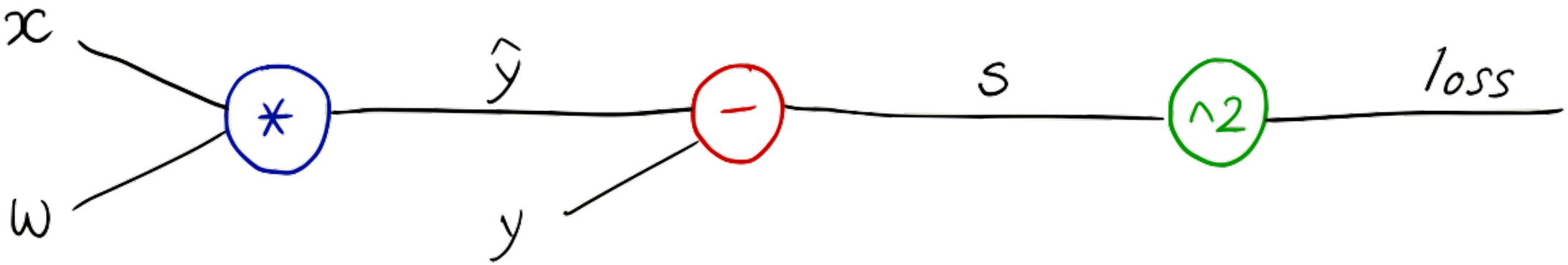
Computational graph

$$\hat{y} = x * w \qquad loss = (\hat{y} - y)^2 = (x * w - y)^2$$

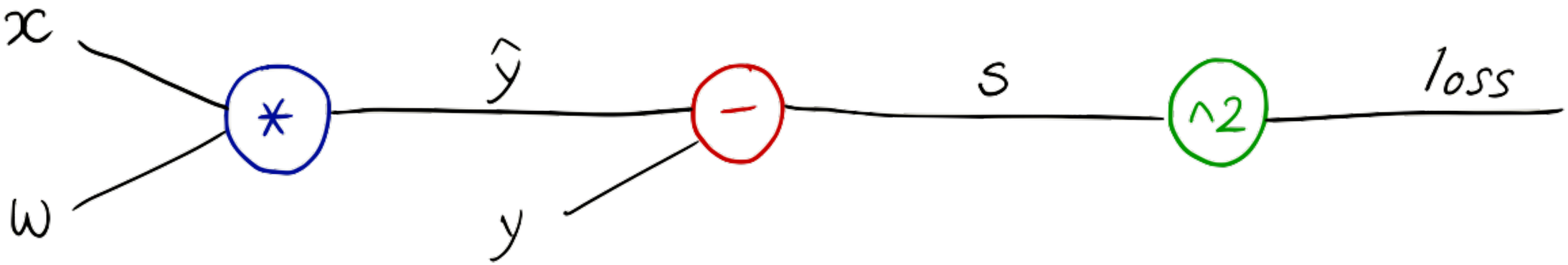


Computational graph

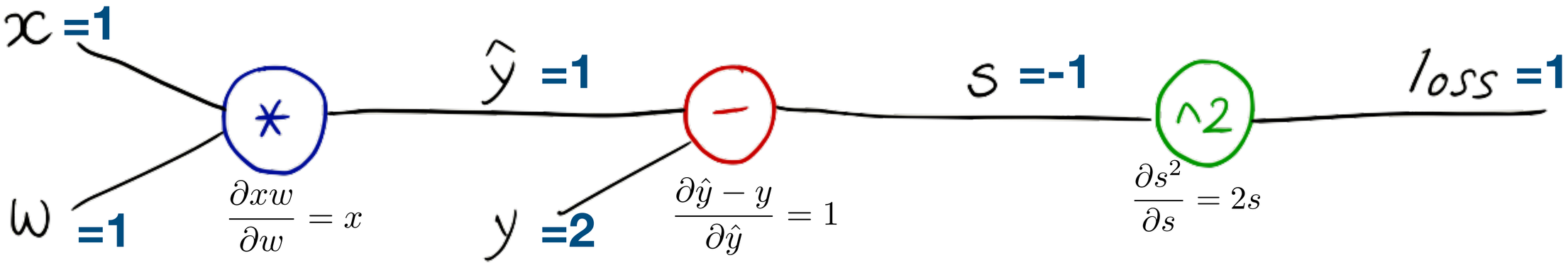
$$\hat{y} = x * w \qquad loss = (\hat{y} - y)^2 = (x * w - y)^2$$



1 Forward pass x=1, y = 2 where w=1

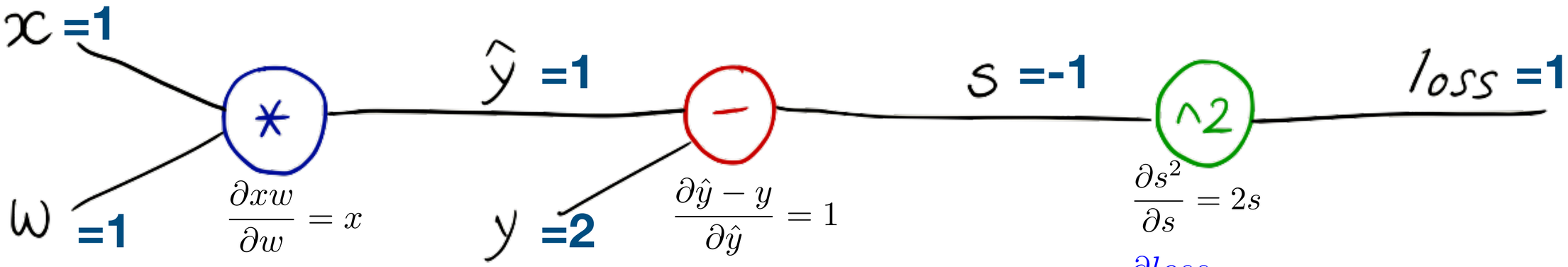


2 Backward propagation



$$\frac{\partial loss}{\partial w} =$$

2 Backward propagation



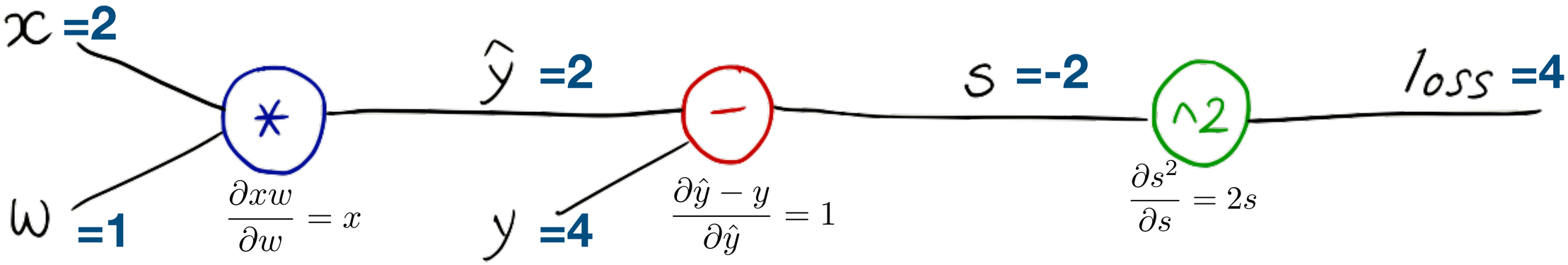
$$\frac{\partial loss}{\partial s} = 2s = -2$$

$$\frac{\partial loss}{\partial \hat{y}} = \frac{\partial loss}{\partial s}\frac{\partial s}{\partial \hat{y}} = -2 * 1 = -2$$

$$\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} = -2 * x = -2 * 1 = -2$$

```
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
```

Exercise: x = 2, y=4, w=1



$$\frac{\partial loss}{\partial w} =$$

Data and **Variable**

```python
import torch
from torch import nn
from torch.autograd import Variable


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

w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value
```

http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

Data and **Variable**

A graph is created on the fly


```
from torch.autograd import Variable

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

http://pytorch.org/docs/master/notes/autograd.html?highlight=variable

Model and Loss

```
import torch
from torch import nn
from torch.autograd import Variable


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

w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value

# our model forward pass
def forward(x):
    return x*w

# Loss function
def loss(x, y):
    y_pred = forward(x)
    return (y_pred-y)*(y_pred-y)
```

Training: forward, backward, and update weight

```
# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print("\tgrad: ", x, y, w.grad.data[0])
        w.data = w.data - 0.01 * w.grad.data

        # Manually zero the gradients after running the backward pass and update w
        w.grad.data.zero_()

    print("progress:", epoch, l.data[0])
```

Output

```
# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)
        l.backward()
        print("\tgrad: ", x, y, w.grad.data[0])
        w.data = w.data - 0.01 * w.grad.data

        # Manually zero the gradients after running the backward pass and update w
        w.grad.data.zero_()

    print("progress:", epoch, l.data[0])
```

```
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 0.35690122842788696
    grad:  1.0 2.0 -0.3265852928161621
    grad:  2.0 4.0 -1.2802143096923828
    grad:  3.0 6.0 -2.650045394897461
progress: 6 0.195076122879982
    grad:  1.0 2.0 -0.24144840240478516
    grad:  2.0 4.0 -0.9464778900146484
    grad:  3.0 6.0 -1.9592113494873047
progress: 7 0.10662525147199631
    grad:  1.0 2.0 -0.17850565910339355
    grad:  2.0 4.0 -0.699742317199707
    grad:  3.0 6.0 -1.4484672546386719
```

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
    grad:  1.0 2.0 -1.478624
    grad:  2.0 4.0 -5.796206079999999
    grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
    grad:  1.0 2.0 -1.093164466688
    grad:  2.0 4.0 -4.285204709416961
    grad:  3.0 6.0 -8.87037374849311
progress: 2 1.4696334962911515
    grad:  1.0 2.0 -0.8081896081960389
    grad:  2.0 4.0 -3.1681032641284723
    grad:  3.0 6.0 -6.557973756745939
progress: 3 0.8032755585999681
    grad:  1.0 2.0 -0.59750427561463
    grad:  2.0 4.0 -2.3422167604093502
    grad:  3.0 6.0 -4.848388694047353
progress: 4 0.43905614881022015
    grad:  1.0 2.0 -0.44174208101320334
    grad:  2.0 4.0 -1.7316289575717576
    grad:  3.0 6.0 -3.584471942173538
progress: 5 0.2399802903801062
    grad:  1.0 2.0 -0.3265852213980338
    grad:  2.0 4.0 -1.2802140678802925
    grad:  3.0 6.0 -2.650043120512205
progress: 6 0.1311689630744999
    grad:  1.0 2.0 -0.241448373202223
    grad:  2.0 4.0 -0.946477622952715
    grad:  3.0 6.0 -1.9592086795121197
progress: 7 0.07169462478267678
    grad:  1.0 2.0 -0.17850567968888198
    grad:  2.0 4.0 -0.6997422643804168
    grad:  3.0 6.0 -1.4484664872674653
progress: 8 0.03918700813247573
    grad:  1.0 2.0 -0.13197139106214673
    grad:  2.0 4.0 -0.5173278529636143
    grad:  3.0 6.0 -1.0708686556346834
progress: 9 0.021418922423117836
predict (after training) 4 7.804863933862125
```

Output

(from numeric gradient computation)

```python
# Before training
print("predict (before training)",  4, forward(4))

# Training loop
for epoch in range(10):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print ("progress:", epoch, l)

# After training
print("predict (after training)",  4, forward(4))
```

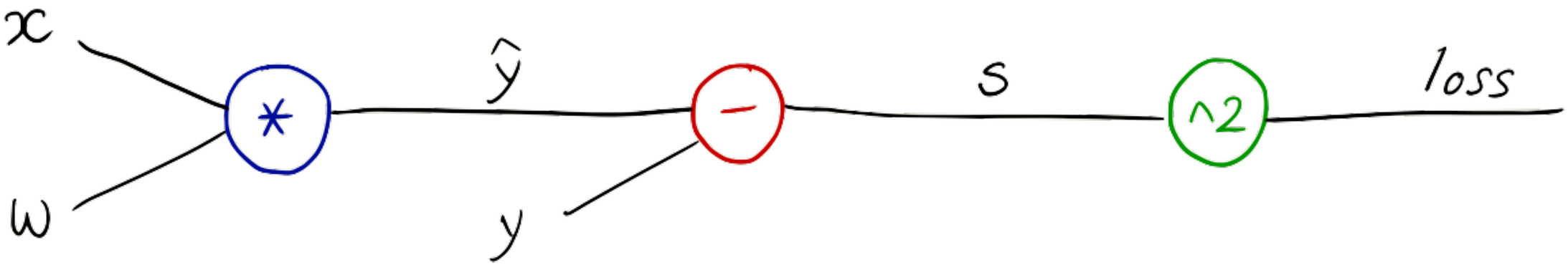
Output
(from numeric gradient computation)

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
    grad:  1.0 2.0 -1.478624
    grad:  2.0 4.0 -5.796206079999999
    grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
    grad:  1.0 2.0 -1.093164466688
    grad:  2.0 4.0 -4.285204709416961
    grad:  3.0 6.0 -8.87037374849311
progress: 2 1.4696334962911515
    grad:  1.0 2.0 -0.8081896081960389
    grad:  2.0 4.0 -3.1681032641284723
    grad:  3.0 6.0 -6.557973756745939
progress: 3 0.8032755585999681
    grad:  1.0 2.0 -0.59750427561463
    grad:  2.0 4.0 -2.3422167604093502
    grad:  3.0 6.0 -4.848388694047353
progress: 4 0.43905614881022015
    grad:  1.0 2.0 -0.44174208101320334
    grad:  2.0 4.0 -1.7316289575717576
    grad:  3.0 6.0 -3.584471942173538
progress: 5 0.2399802903801062
    grad:  1.0 2.0 -0.3265852213980338
    grad:  2.0 4.0 -1.2802140678802925
    grad:  3.0 6.0 -2.650043120512205
progress: 6 0.1311689630744999
    grad:  1.0 2.0 -0.241448373202223
    grad:  2.0 4.0 -0.946477622952715
    grad:  3.0 6.0 -1.9592086795121197
progress: 7 0.07169462478267678
    grad:  1.0 2.0 -0.17850567968888198
    grad:  2.0 4.0 -0.6997422643804168
```

Output
(computational graph)

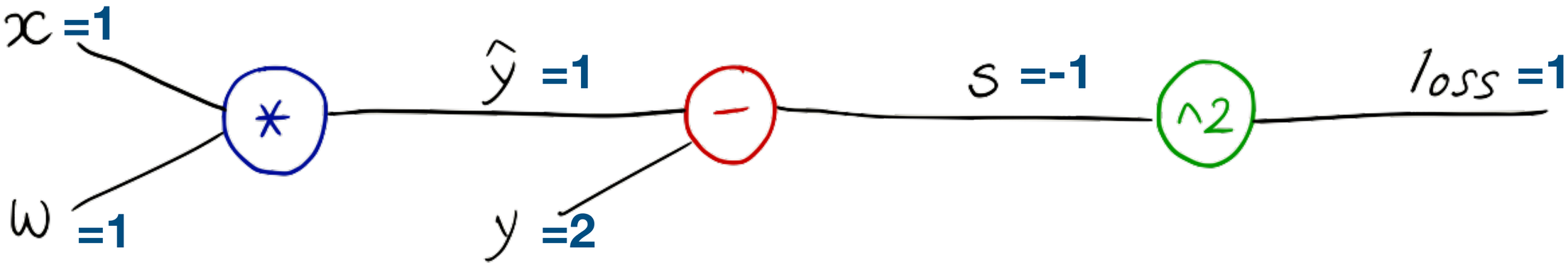

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
    grad:  1.0 2.0 -1.478624
    grad:  2.0 4.0 -5.796206079999999
    grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
    grad:  1.0 2.0 -1.093164466688
    grad:  2.0 4.0 -4.285204709416961
    grad:  3.0 6.0 -8.87037374849311
progress: 2 1.4696334962911515
    grad:  1.0 2.0 -0.8081896081960389
    grad:  2.0 4.0 -3.1681032641284723
    grad:  3.0 6.0 -6.557973756745939
progress: 3 0.8032755585999681
    grad:  1.0 2.0 -0.59750427561463
    grad:  2.0 4.0 -2.3422167604093502
    grad:  3.0 6.0 -4.848388694047353
progress: 4 0.43905614881022015
    grad:  1.0 2.0 -0.44174208101320334
    grad:  2.0 4.0 -1.7316289575717576
    grad:  3.0 6.0 -3.584471942173538
progress: 5 0.2399802903801062
    grad:  1.0 2.0 -0.3265852213980338
    grad:  2.0 4.0 -1.2802140678802925
    grad:  3.0 6.0 -2.650043120512205
progress: 6 0.1311689630744999
    grad:  1.0 2.0 -0.241448373202223
    grad:  2.0 4.0 -0.946477622952715
    grad:  3.0 6.0 -1.9592086795121197
progress: 7 0.07169462478267678
    grad:  1.0 2.0 -0.17850567968888198
    grad:  2.0 4.0 -0.6997422643804168
```

PyTorch forward/backward



Forward pass

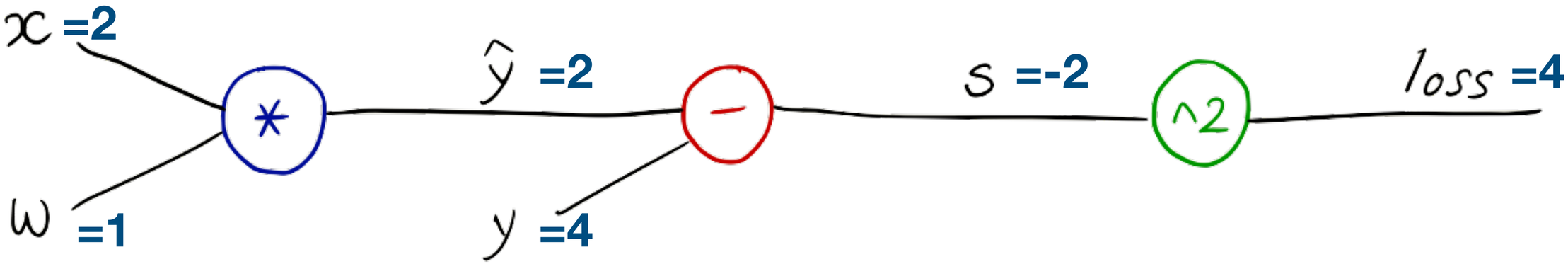
```
w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value
l = loss(x=1, y=1)
```



$$\frac{\partial loss}{\partial w} =$$

Back propagation

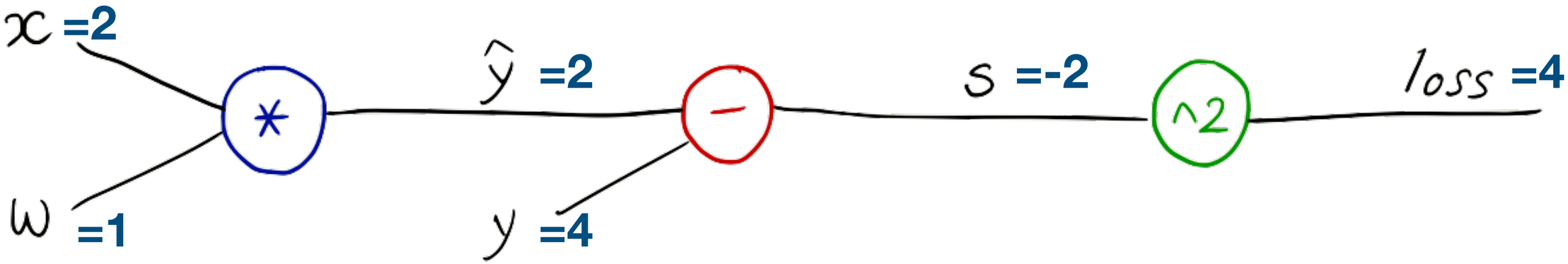
```
l.backward()

# Manually zero the gradients after running the backward pass and update w
w.grad.data.zero_()
```



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

Weight update (step)

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



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

Lecture 5: Linear regression in the PyTorch way

ML/DL for Everyone with PYTORCH

Lecture 5: Linear regression in PyTorch way

Sung Kim <hunkim+ml@gmail.com> HKUST

Code: https://github.com/hunkim/PyTorchZeroToAll

Data definition (3x1)

```
import torch
from torch.autograd import Variable

x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0]]))
y_data = Variable(torch.Tensor([[2.0], [4.0], [6.0]]))
```

Model class in PyTorch way

```
import torch
from torch.autograd import Variable

x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0]]))
y_data = Variable(torch.Tensor([[2.0], [4.0], [6.0]]))

class Model(torch.nn.Module):
  def __init__(self):
    """
    In the constructor we instantiate two nn.Linear module
    """
    super(Model, self).__init__()
    self.linear = torch.nn.Linear(1, 1) # One in and one out

  def forward(self, x):
    """
    In the forward function we accept a Variable of input data and we must return
    a Variable of output data. We can use Modules defined in the constructor as
    well as arbitrary operators on Variables.
    """
    y_pred = self.linear(x)
    return y_pred

# our model
model = Model()
```

Construct loss and optimizer

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

https://github.com/jcjohnson/pytorch-examples

Training: forward, loss, backward, step

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

https://github.com/jcjohnson/pytorch-examples

Testing Model

```python
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```python
# After training
hour_var = Variable(torch.Tensor([[4.0]]))
print("predict (after training)",  4, model.forward(hour_var).data[0][0])
```

https://github.com/jcjohnson/pytorch-examples

Output

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
# After training
hour_var = Variable(torch.Tensor([[4.0]]))
print("predict (after training)",  4, model.forward(hour_var).data[0][0])
```

```
470 1.52139027704834e-05
471 1.4996051504567731e-05
472 1.4781335266889073e-05
473 1.4567947800969705e-05
474 1.4360077329911292e-05
475 1.4153701158647891e-05
476 1.3949686035630293e-05
477 1.3749523532169405e-05
478 1.3551662959798705e-05
479 1.3357152056414634e-05
480 1.3165942618797999e-05
481 1.2975904610357247e-05
482 1.2790364962711465e-05
483 1.2605956726474687e-05
484 1.2424526175891515e-05
485 1.2245835932844784e-05
486 1.2070459888491314e-05
487 1.1897350304934662e-05
488 1.1724299838533625e-05
489 1.155646714323666e-05
490 1.1392002306820359e-05
491 1.1226966307731345e-05
492 1.1066998922615312e-05
493 1.090722162189195e-05
494 1.0750130059022922e-05
495 1.0595314961392432e-05
496 1.0444626241223887e-05
497 1.029352642945014e-05
498 1.0146304703084752e-05
499 9.999960639106575e-06
predict (after training) 4 7.996364593505859
```

https://github.com/jcjohnson/pytorch-examples

```python
import torch
from torch.autograd import Variable

x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0]]))
y_data = Variable(torch.Tensor([[2.0], [4.0], [6.0]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1) # One in and one out

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.linear(x)
        return y_pred

# our model
model = Model()


# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()


# After training
hour_var = Variable(torch.Tensor([[4.0]]))
print("predict (after training)",  4, model.forward(hour_var).data[0][0])
```

1 **Design your model using class**

2 **Construct loss and optimizer (select from PyTorch API)**

3 **Training cycle (forward, backward, update)**

Training CIFAR10 Classifier

```
########################################################################
# 1. Define a Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

########################################################################
# 2. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
# 3. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# network and optimize
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

1. **Design your model using class**

2. **Construct loss and optimizer (select from PyTorch API)**

3. **Training cycle (forward, backward, update)**

http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Build fun networks

- Neure Net components
  - CNN
  - RNN
  - Activations
- Losses
- Optimizers

torch.nn

- ⊞ Containers
- ⊞ Convolution Layers
- ⊞ Pooling Layers
- ⊞ Padding Layers
- ⊞ Non-linear Activations
- ⊞ Normalization layers
- ⊞ Recurrent layers
- ⊞ Linear layers
- ⊞ Dropout layers
- ⊞ Sparse layers
- ⊞ Distance functions
- ⊞ Loss functions
- ⊞ Vision layers

⊟ Convolution Layers

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d

⊟ Recurrent layers

- RNN
- LSTM
- GRU
- RNNCell
- LSTMCell
- GRUCell

⊟ Non-linear Activations

- ReLU
- ReLU6
- ELU
- SELU
- PReLU
- LeakyReLU
- Threshold
- Hardtanh
- Sigmoid
- Tanh
- LogSigmoid
- Softplus
- Softshrink
- Softsign
- Tanhshrink
- Softmin
- Softmax
- Softmax2d
- LogSoftmax

http://pytorch.org/docs/master/nn.html

Loss functions

- Loss functions
  - L1Loss
  - MSELoss
  - CrossEntropyLoss
  - NLLLoss
  - PoissonNLLLoss
  - NLLLoss2d
  - KLDivLoss
  - BCELoss
  - BCEWithLogitsLoss
  - MarginRankingLoss
  - HingeEmbeddingLoss
  - MultiLabelMarginLoss
  - SmoothL1Loss
  - SoftMarginLoss
  - MultiLabelSoftMarginLoss
  - CosineEmbeddingLoss
  - MultiMarginLoss
  - TripletMarginLoss

http://pytorch.org/docs/master/nn.html

Table 1: List of losses analysed in this paper. $\mathbf{y}$ is true label as one-hot encoding, $\hat{\mathbf{y}}$ is true label as +1/-1 encoding, $\mathbf{o}$ is the output of the last layer of the network, $\cdot^{(j)}$ denotes jth dimension of a given vector, and $\sigma(\cdot)$ denotes probability estimate.

symbol	name	equation
$\mathcal{L}_1$	L_1 loss	$\|\mathbf{y}-\mathbf{o}\|_1$
$\mathcal{L}_2$	L_2 loss	$\|\mathbf{y}-\mathbf{o}\|_2^2$
$\mathcal{L}_1 \circ \sigma$	expectation loss	$\|\mathbf{y}-\sigma(\mathbf{o})\|_1$
$\mathcal{L}_2 \circ \sigma$	regularised expectation loss[1]	$\|\mathbf{y}-\sigma(\mathbf{o})\|_2^2$
$\mathcal{L}_\infty \circ \sigma$	Chebyshev loss	$\max_j \lvert\sigma(\mathbf{o})^{(j)}-\mathbf{y}^{(j)}\rvert$
hinge	hinge [13] (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})$
hinge^2	squared hinge (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^2$
hinge^3	cubed hinge (margin) loss	$\sum_j \max(0, \frac{1}{2}-\hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^3$
log	log (cross entropy) loss	$-\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}$
log^2	squared log loss	$-\sum_j [\mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}]^2$
tan	Tanimoto loss	$\frac{-\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\|\sigma(\mathbf{o})\|_2^2+\|\mathbf{y}\|_2^2-\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}$
D_{CS}	Cauchy-Schwarz Divergence [3]	$-\log \frac{\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\|\sigma(\mathbf{o})\|_2\|\mathbf{y}\|_2}$

https://arxiv.org/pdf/1702.05659.pdf

torch.optim

- *class* torch.optim.Adadelta
- *class* torch.optim.Adagrad
- *class* torch.optim.Adam
- *class* torch.optim.Adamax
- *class* torch.optim.ASGD
- *class* torch.optim.RMSprop
- *class* torch.optim.Rprop
- *class* torch.optim.SGD

http://pytorch.org/docs/master/optim.html

Upcoming topics (TBA)

Intermediate

- Convolutional Neural Network
- Deep Residual Network
- Recurrent Neural Network
- Bidirectional Recurrent Neural Network
- Language Model (RNN-LM)
- Generative Adversarial Network

Advanced

- Image Captioning (CNN-RNN)
- Deep Convolutional GAN (DCGAN)
- Variational Auto-Encoder
- Neural Style Transfer

https://github.com/yunjey/pytorch-tutorial

Upcoming topics (TBA)

- Wasserstein GAN
- OptNet: Differentiable Optimization as a Layer in Neural Networks
- Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer
- Wide ResNet model in PyTorch
- Task-based End-to-end Model Learning
- An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition
- Scaling the Scattering Transform: Deep Hybrid Networks
- Adversarial Generator-Encoder Network
- Conditional Similarity Networks
- Multi-style Generative Network for Real-time Transfer
- Image-to-Image Translation with Conditional Adversarial Networks
- Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks
- Inferring and Executing Programs for Visual Reasoning
- On the Effects of Batch and Weight Normalization in Generative Adversarial Networks
- Train longer, generalize better: closing the generalization gap in large batch training of neural networks
- Neural Message Passing for Quantum Chemistry
- DiracNets: Training Very Deep Neural Networks Without Skip-Connections
- Deal or No Deal? End-to-End Learning for Negotiation Dialogues

…

…

…

https://github.com/ritchieng/the-incredible-pytorch

References

- http://pytorch.org/
- https://github.com/pytorch/examples
- https://github.com/ritchieng/the-incredible-pytorch
- https://github.com/yunjey/pytorch-tutorial
- https://github.com/znxlwm/pytorch-generative-model-collections

- https://www.facebook.com/groups/TensorFlowKR/ (in Korean)
- https://www.facebook.com/groups/PyTorchKR/ (in Korean)

ML/DL for Everyone with PYTORCH

TBA

Sung Kim <hunkim+ml@gmail.com> HKUST
Code: https://github.com/hunkim/PyTorchZeroToAll