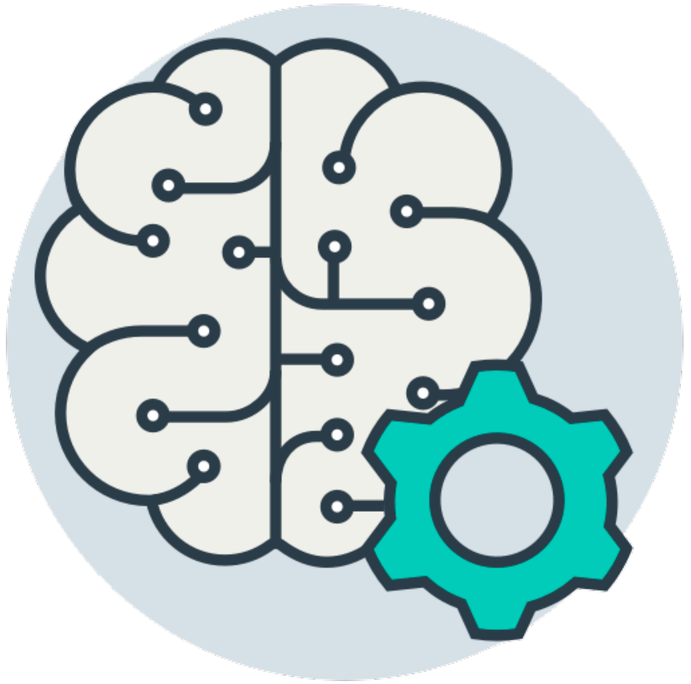
ML/DL for Everyone with PYTORCH

Call for Comments

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

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

What is ML?

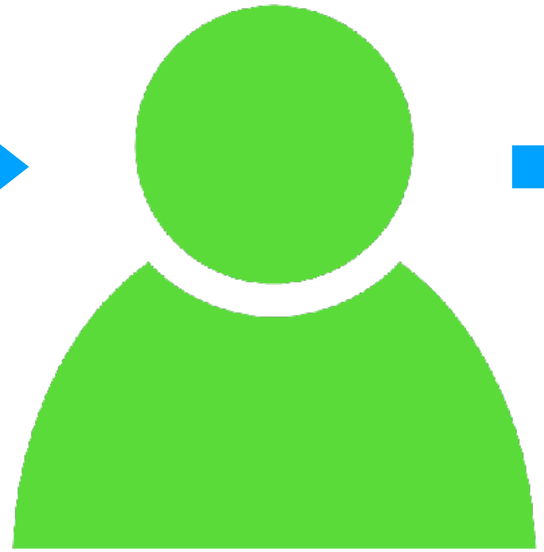


What is Human Intelligence?

What is Human Intelligence?

What to eat for lunch?

Information

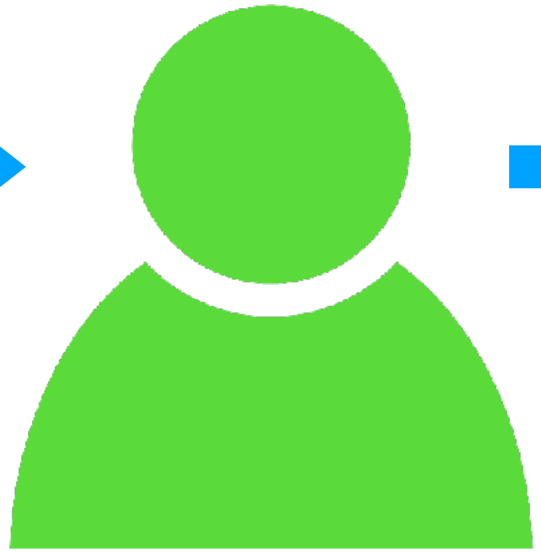


Infer

What is Human Intelligence?

What to dress?

Information

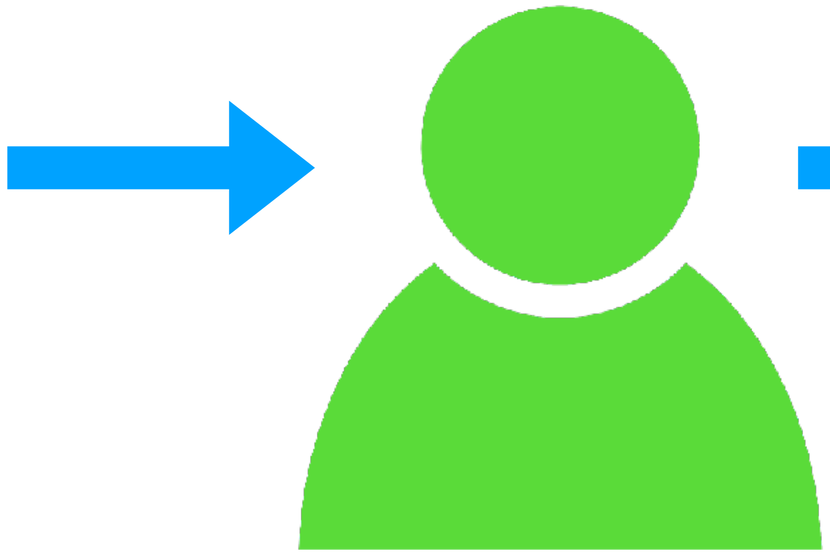


Infer

What is Human Intelligence?

What is this picture?

Image information

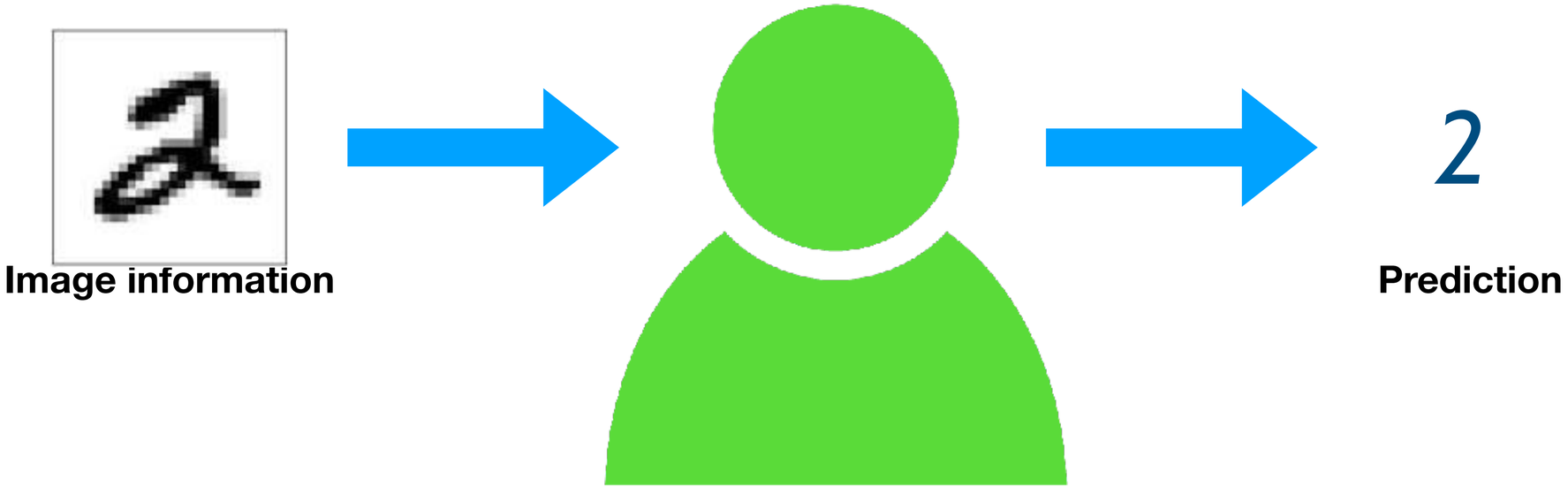


CAT

Prediction

What is Human Intelligence?

What is this number?

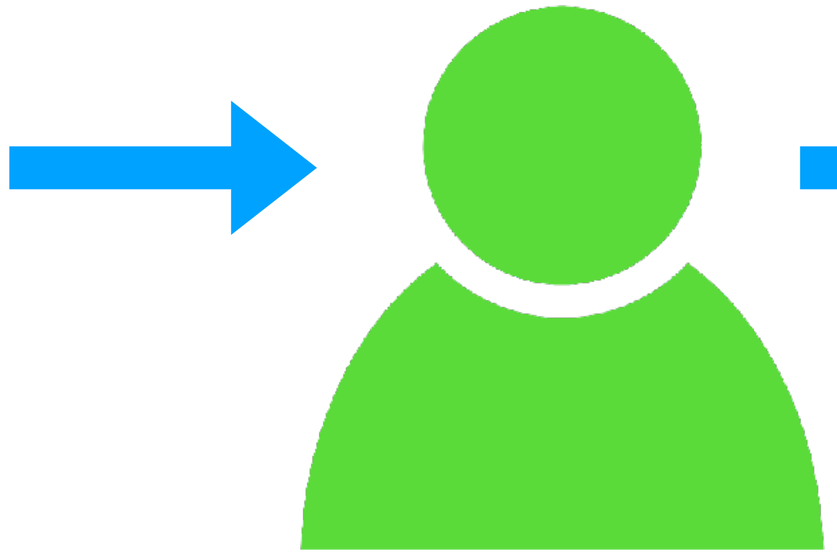


What is Human Intelligence?

What would be the grade if I study 4 hours?

4 hours

information



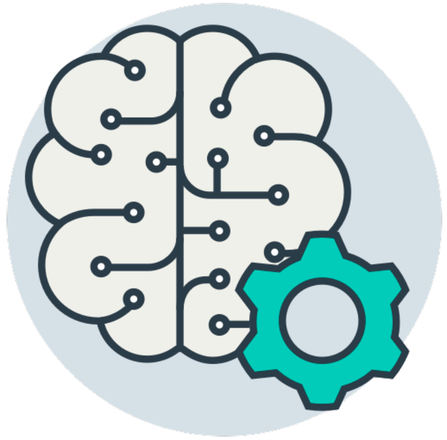
? points

Prediction

Machine Learning

What to dress?

Information

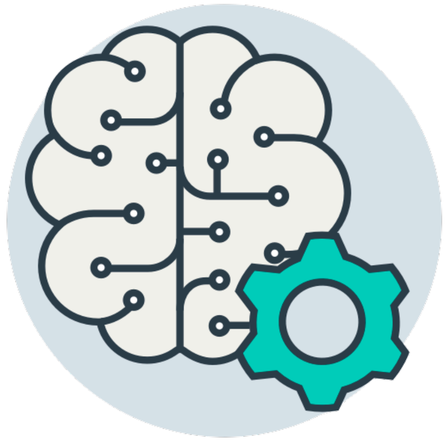


Infer

Machine Learning

What is this picture?

Image information

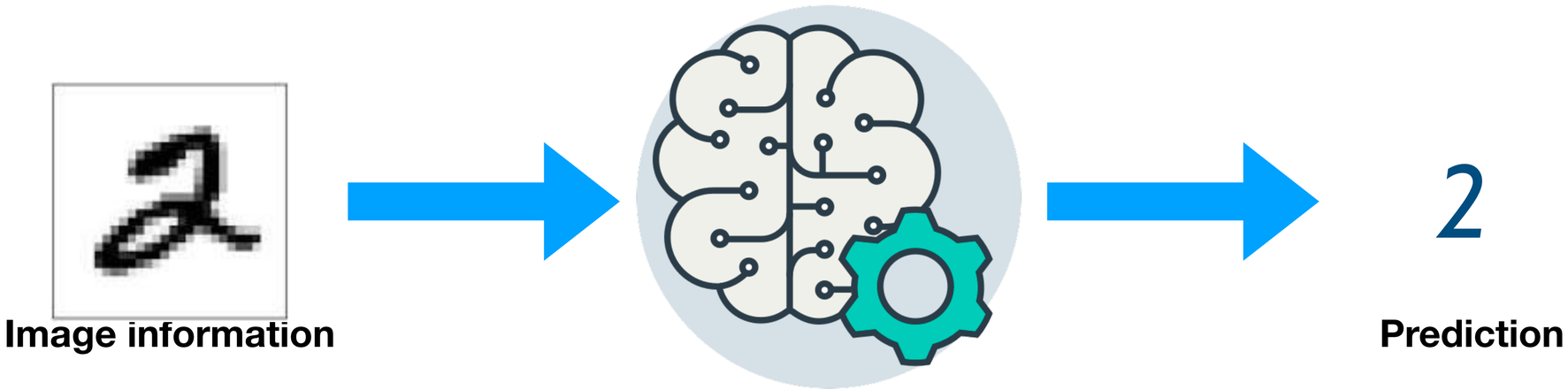


CAT

Prediction

Machine Learning

What is this number?

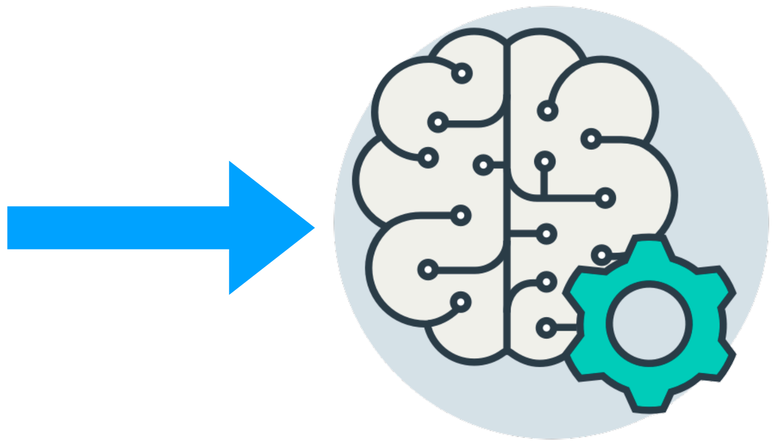


Machine Learning

What would be the grade if I study 4 hours?

4 hours

information

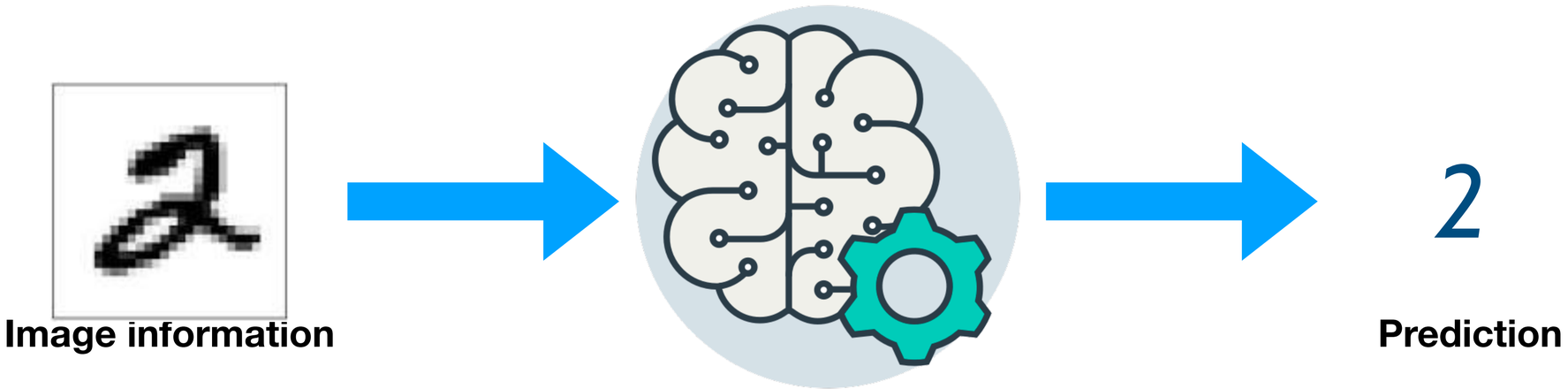


? points

Prediction

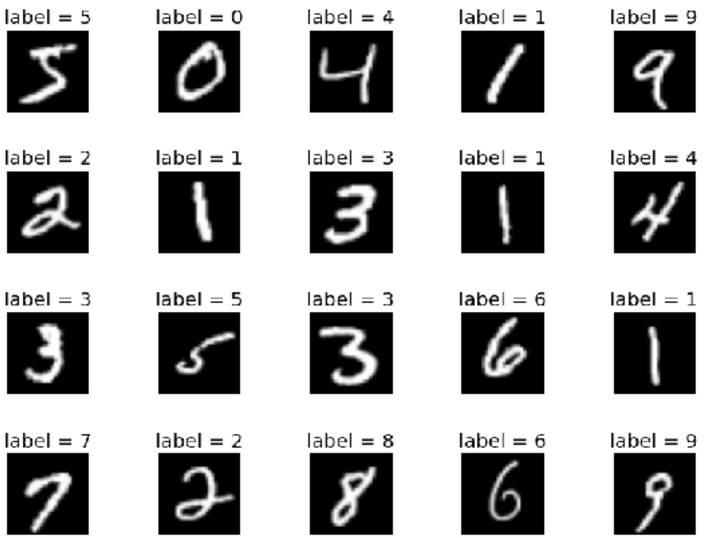
Machine Learning

Machine need lots of training



Machine Learning

Machine need lots of training



Labeled dataset

training

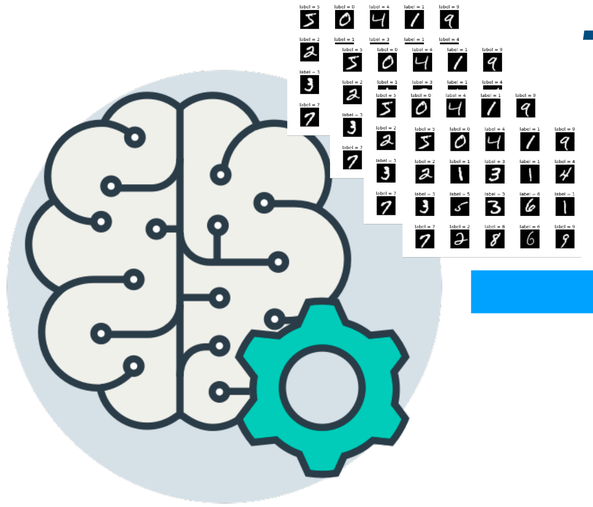
Model

Machine Learning

Predict (test) with trained model

Image information

Test dataset



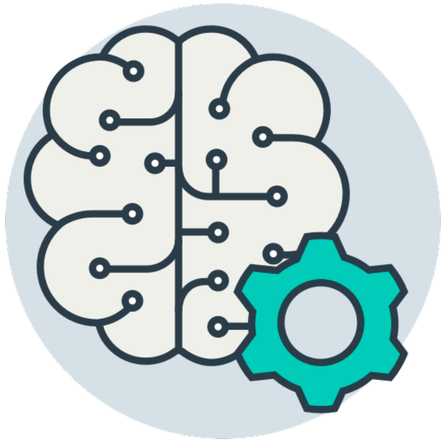
Trained model

2

Prediction

Machine Learning

What would be the grade if I study 4 hours?

4 hours →  → *? points*

Prediction

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

Training dataset

Test dataset

Install PYTORCH

Get Started.

Select your preferences, then run the PyTorch install command.

Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

OS	Linux	**OSX**	
Package Manager	conda	**pip**	Source
Python	2.7	3.5	**3.6**
CUDA	7.5	8.0	**None**

Run this command:

```
pip3 install http://download.pytorch.org/whl/torch-0.2.0.post3-cp36-cp36m-macosx_10_7_x86_64.whl
pip3 install torchvision
# OSX Binaries dont support CUDA, install from source if CUDA is needed
```

Lecture 2: Linear Model

ML/DL for Everyone with PYTORCH

Lecture 2: Linear Model

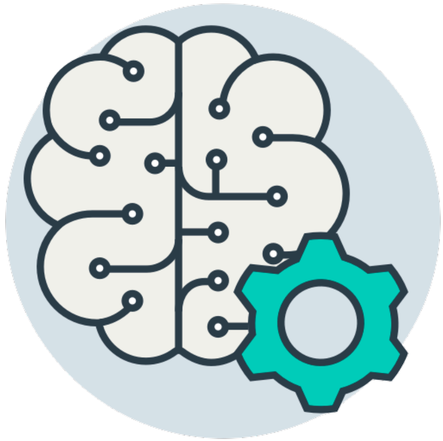
Call for Comments

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

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

Machine Learning

What would be the grade if I study 4 hours?

4 hours →  → *? points*

Prediction

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

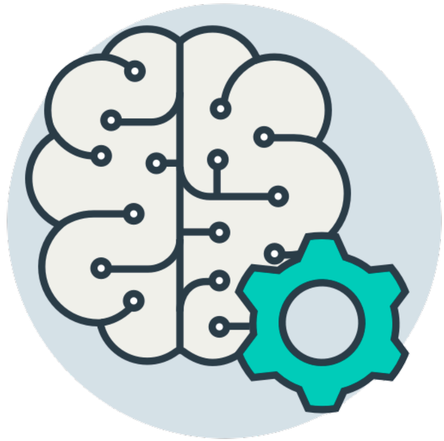
Training dataset

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	?

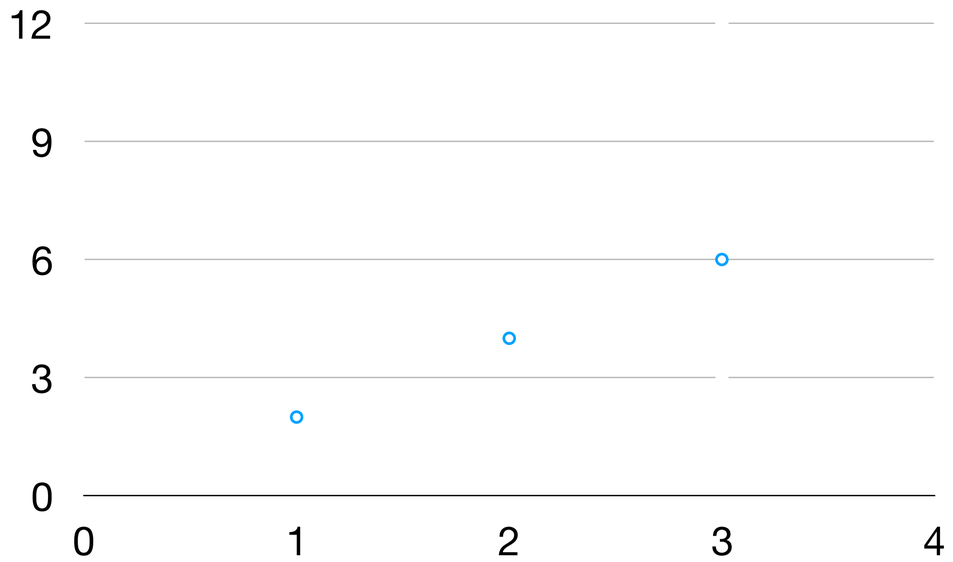


$$\hat{y} = x * w + b$$

Linear Regression

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

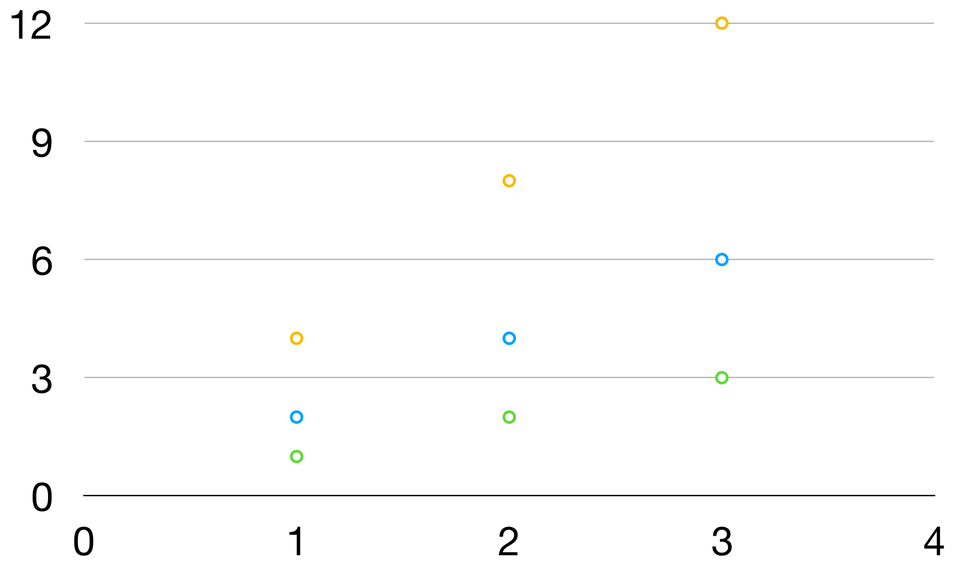
Hours (x)	Points (y)
1	2
2	4
3	6



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)

$$loss = (\hat{y} - y)^2 = (x * w - y)^2 \qquad loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

MSE, mean square error

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
	MSE=56/3=18.7	MSE=14/3=4.7	MSE=0	MSE=14/3=4.7	MSE=56/3=18.7

Loss graph

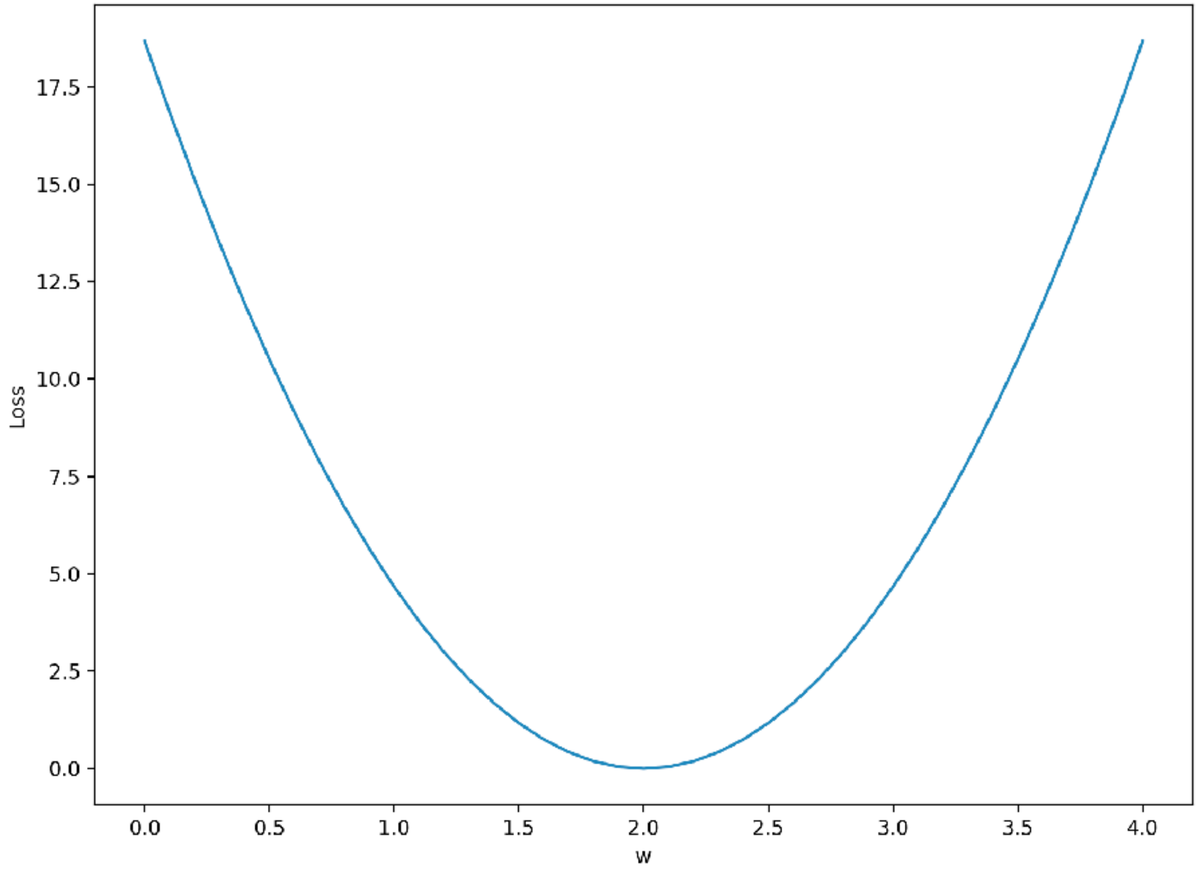
$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

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

$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

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)
```

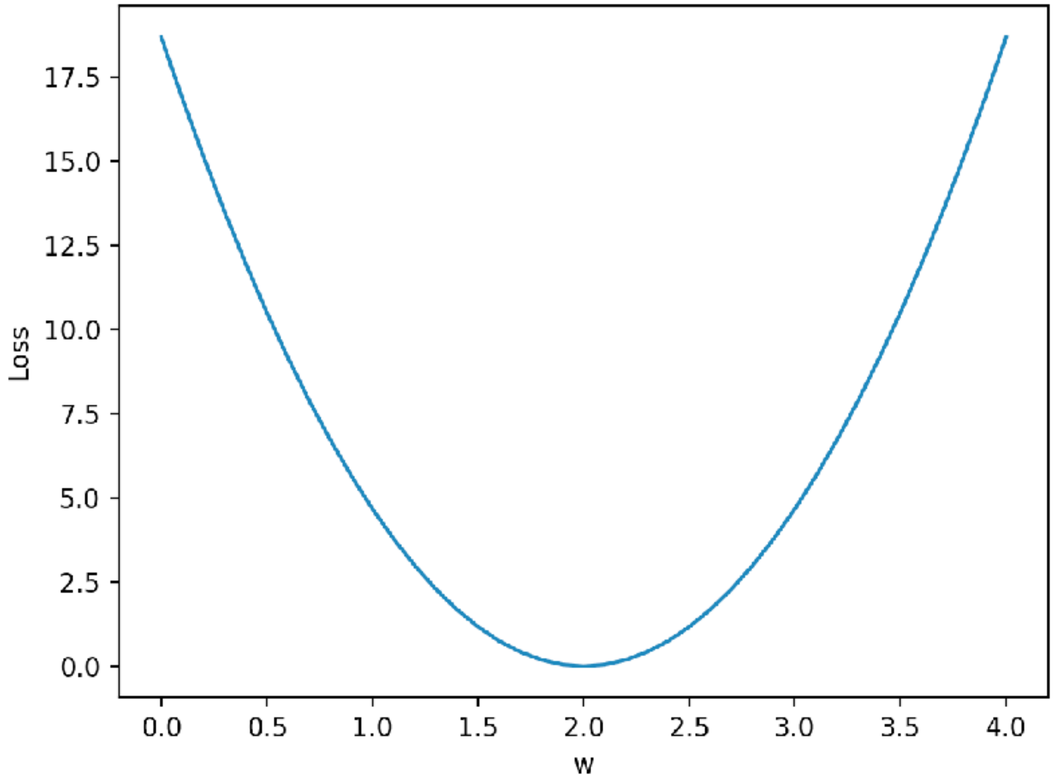
Compute loss for w

```
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 Descent

ML/DL for Everyone with PYTORCH

Lecture 3: Gradient Descent

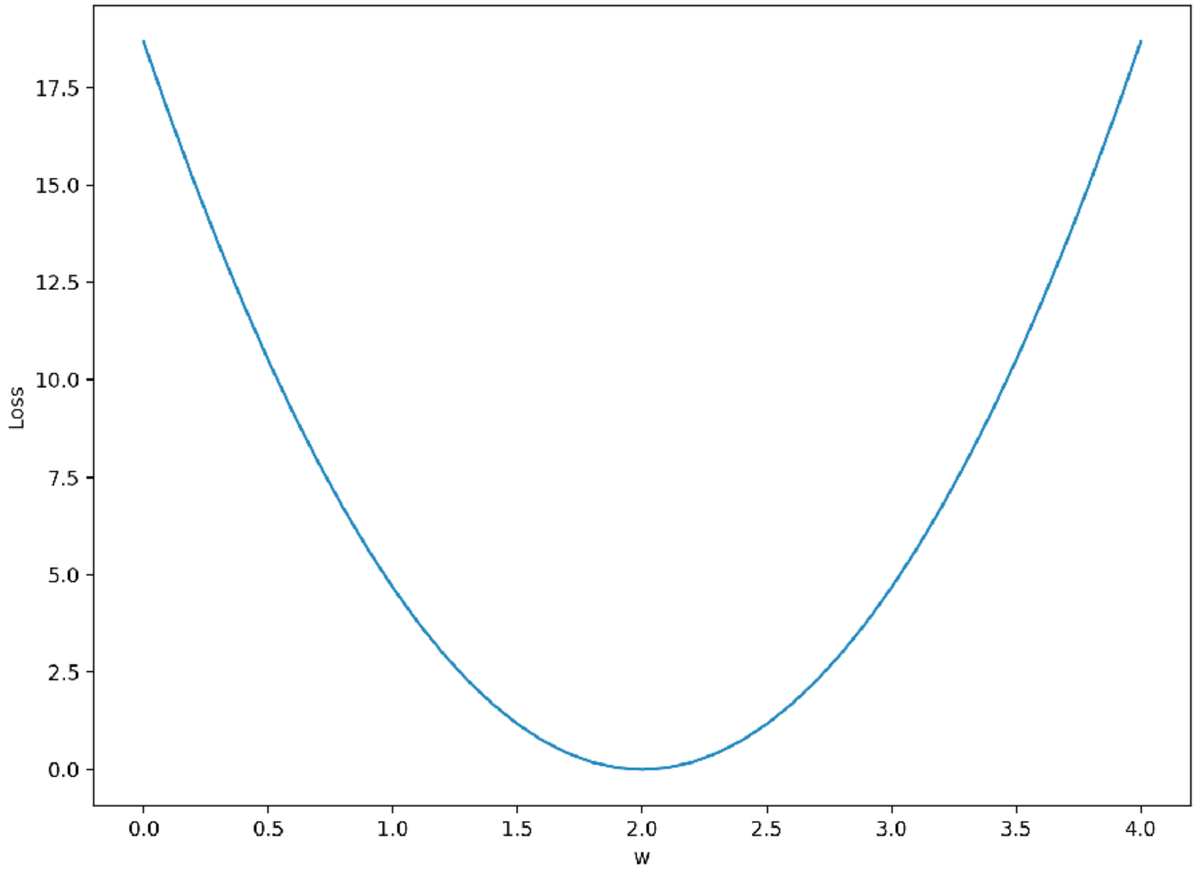
Call for Comments

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

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

Loss graph

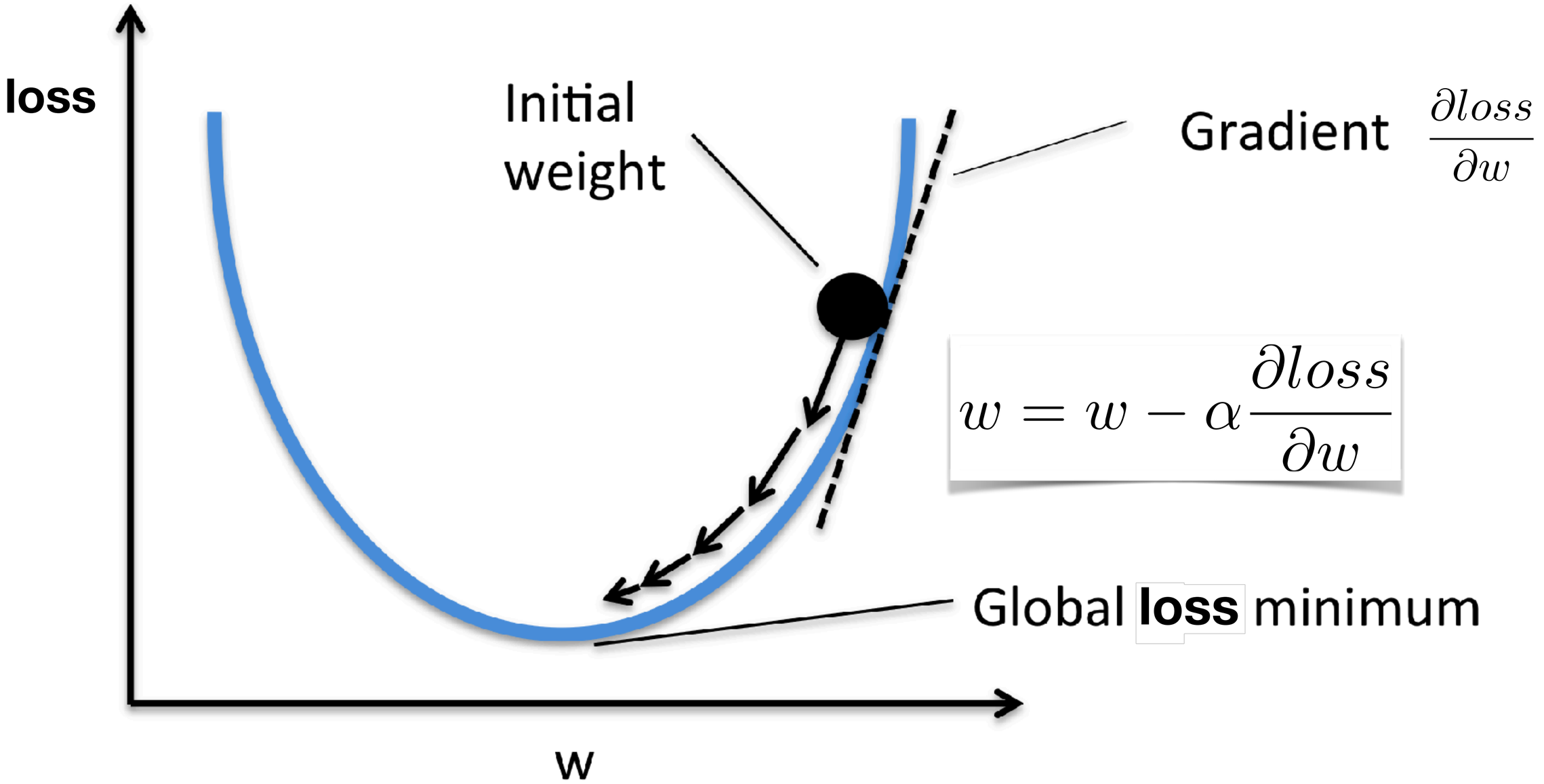
Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
mean=56/3=18.7	mean=14/3=4.7	mean=0	mean=14/3=4.7	mean=56/3=18.7



$$loss(w) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

$$\arg\min_{w} loss(w)$$

Gradient descent algorithm



Derivative

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

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

Derivative

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

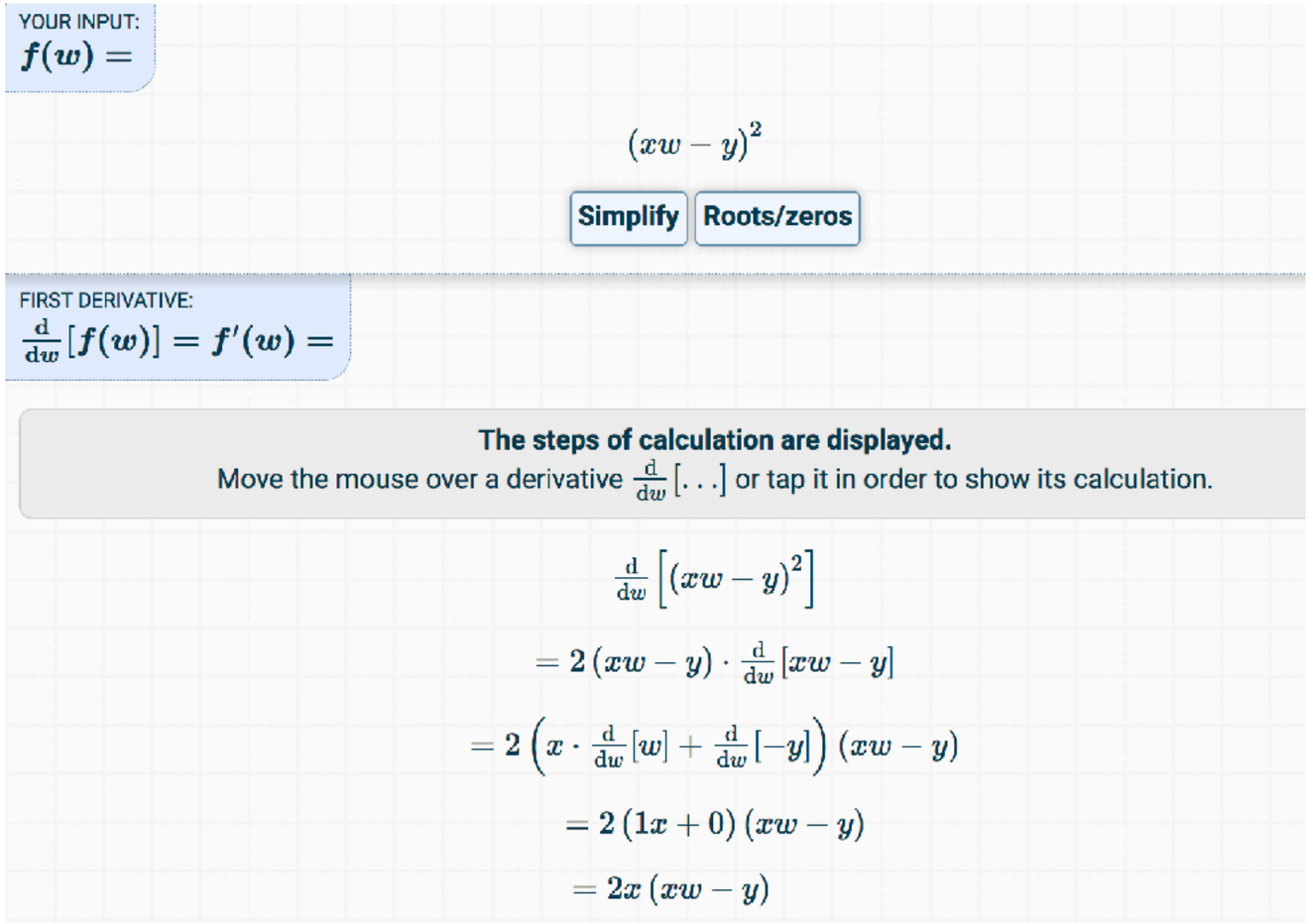
$$w = w - \alpha \frac{\partial loss}{\partial w}$$

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

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

Call for Comments

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

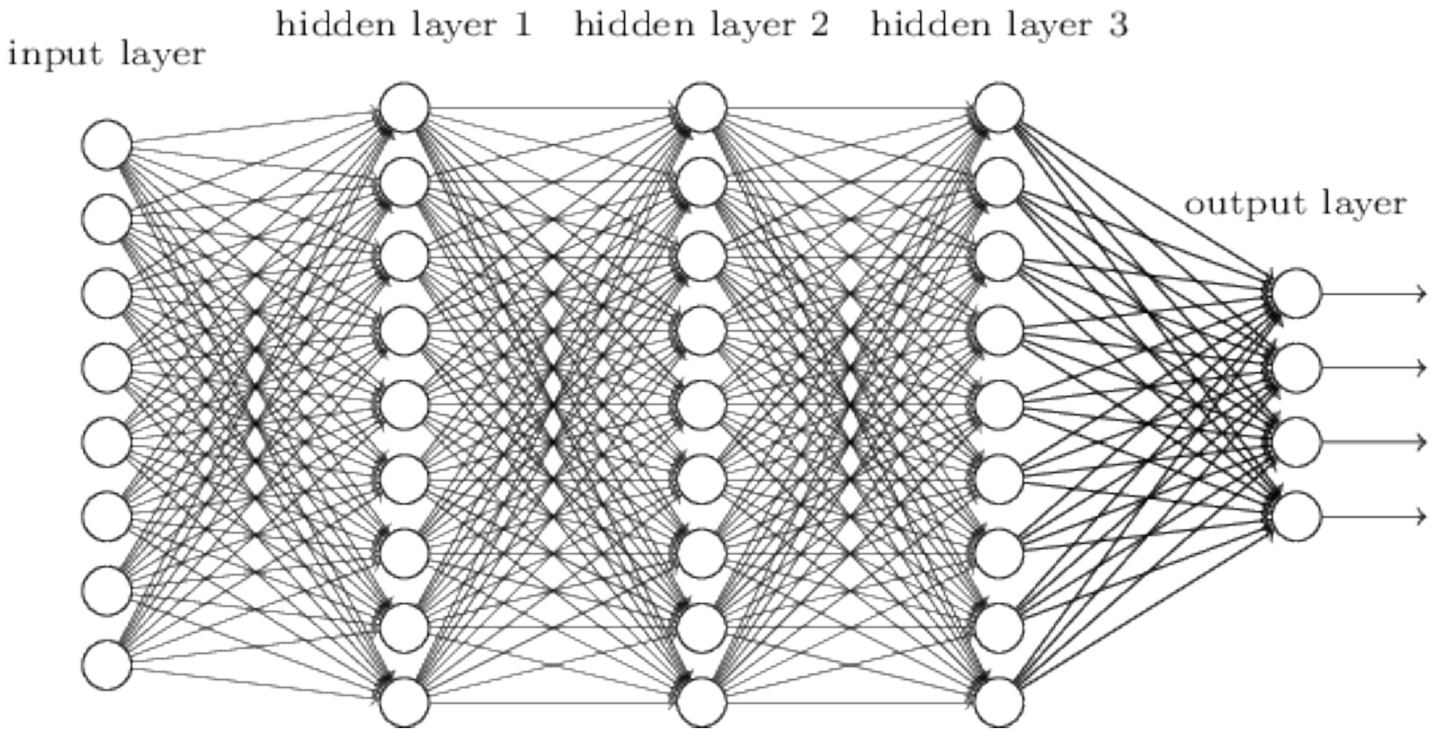
Code: https://github.com/hunkim/PyTorchZeroToAll

Computing gradient in simple network

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

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

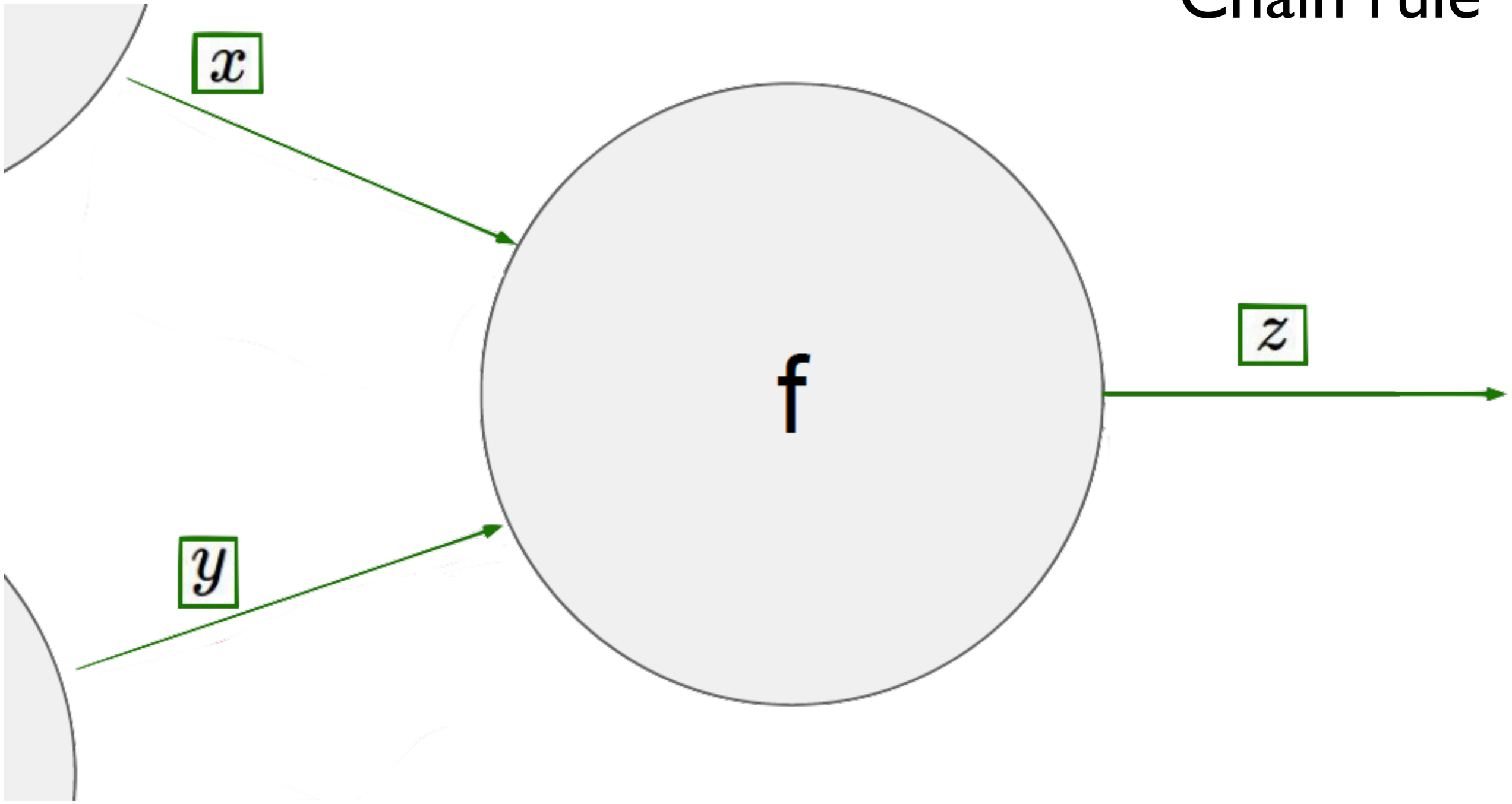
Complicated network?



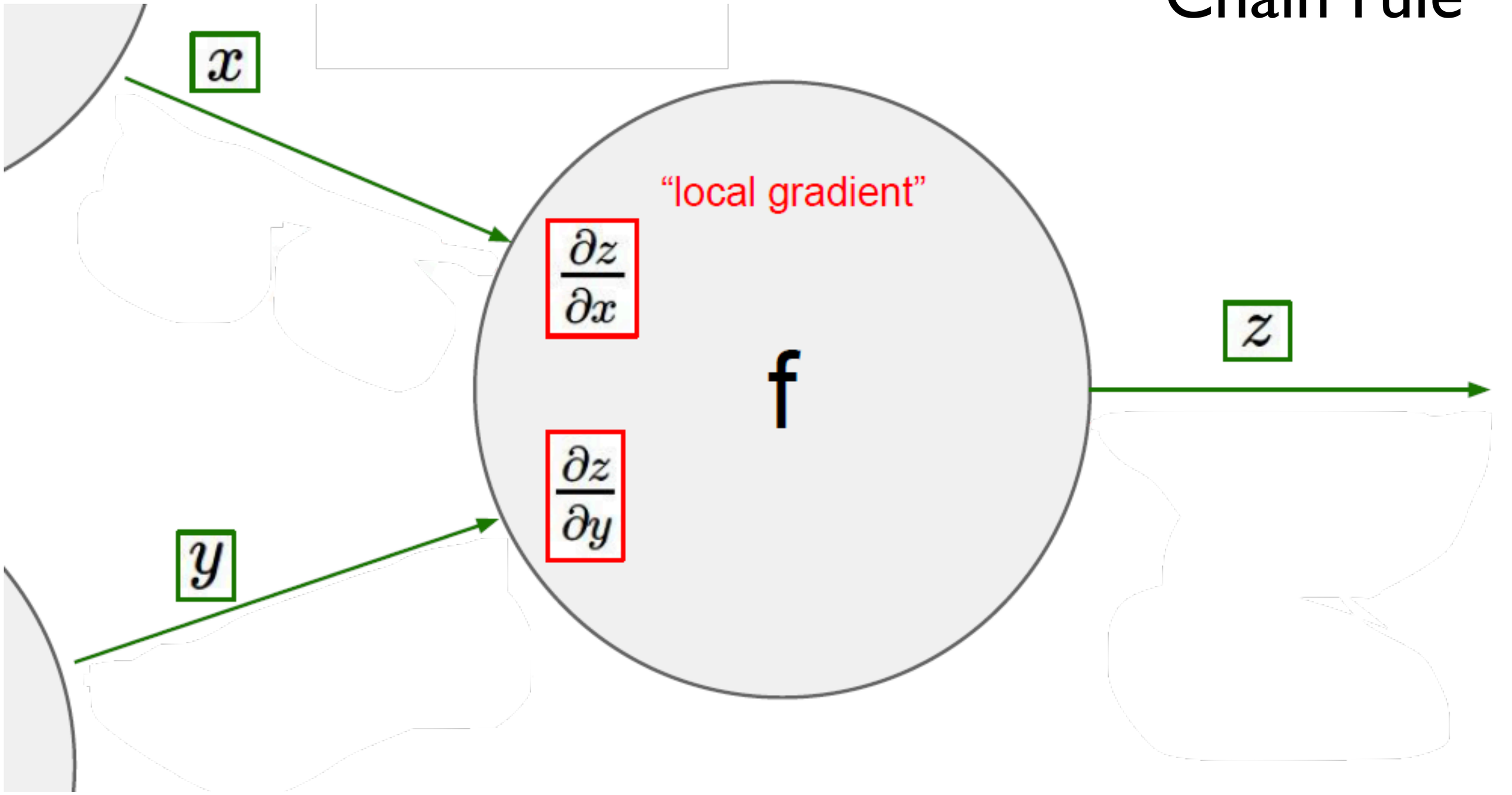
$$\frac{\partial loss}{\partial w} = ?$$

Better way? Computational graph + chain rule

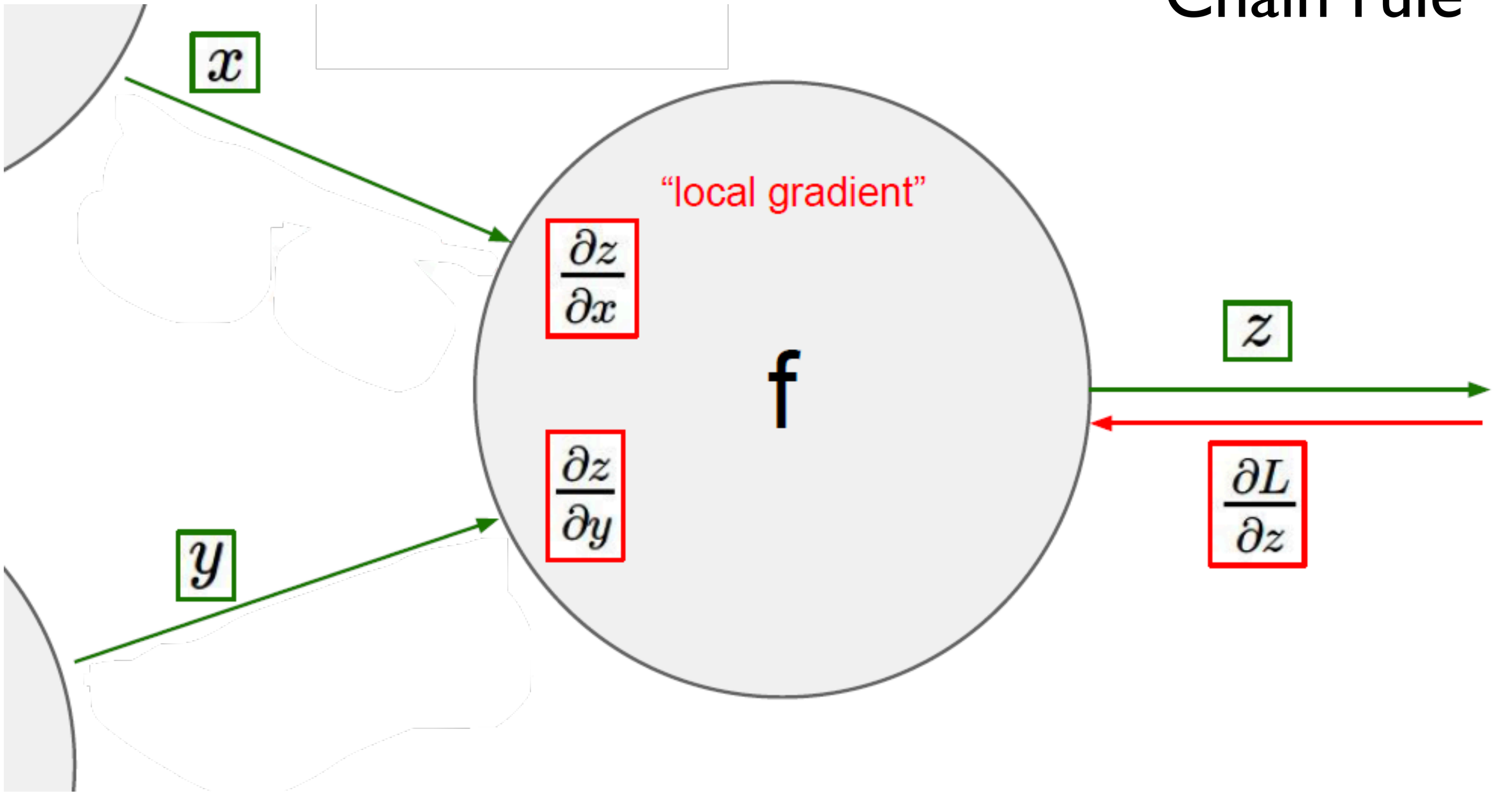
Chain rule



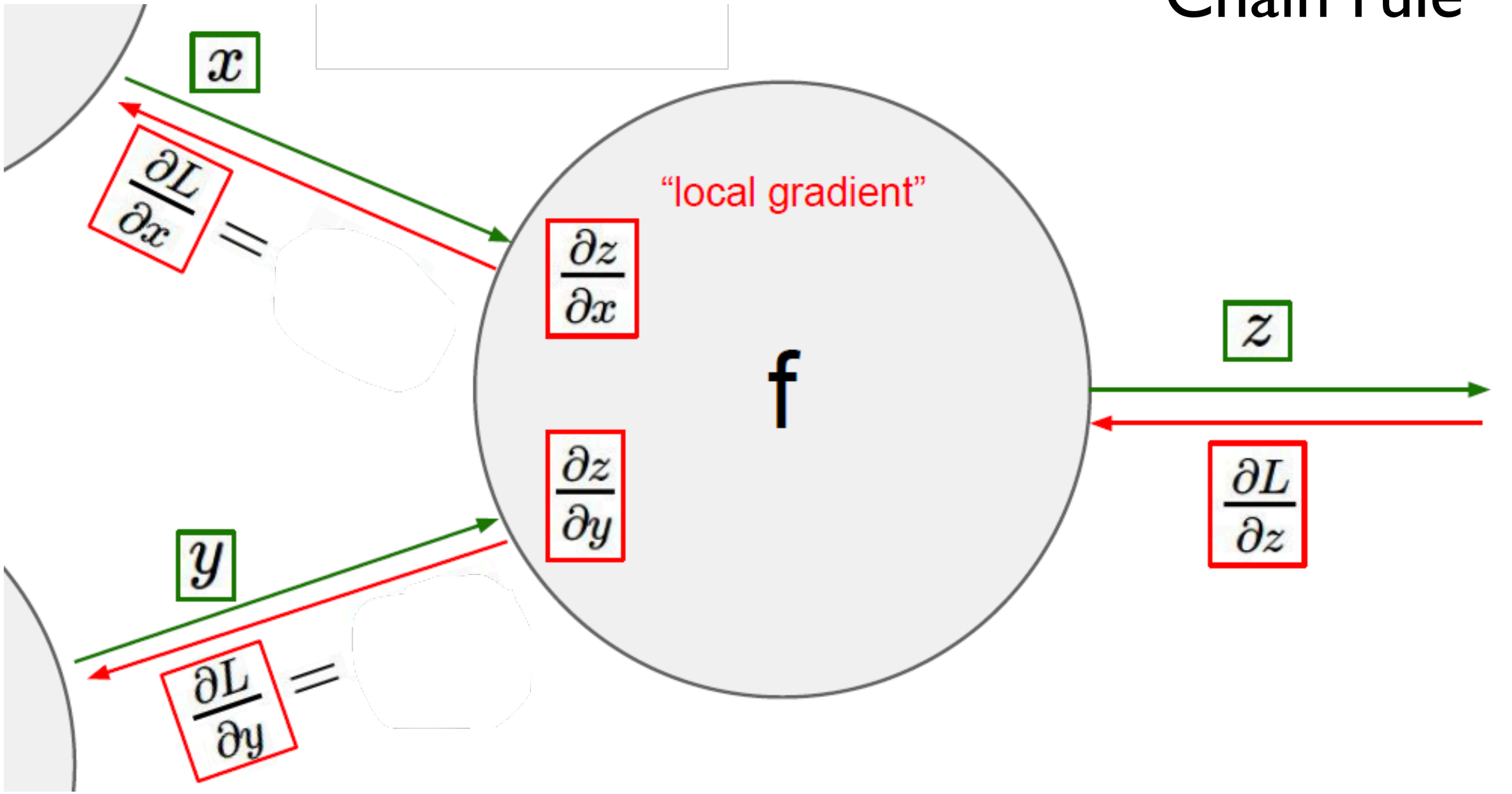
Chain rule


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

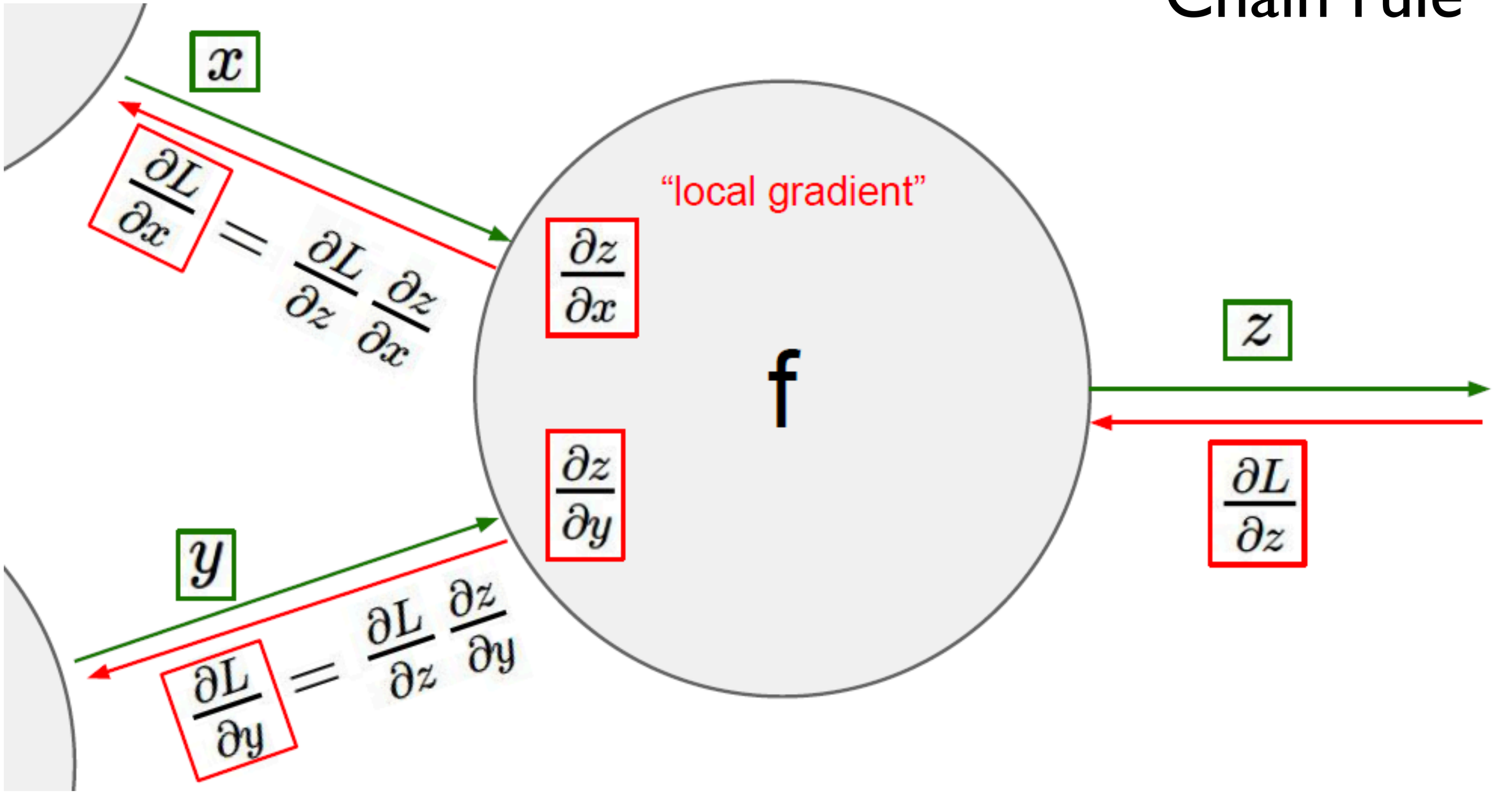
Chain rule


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

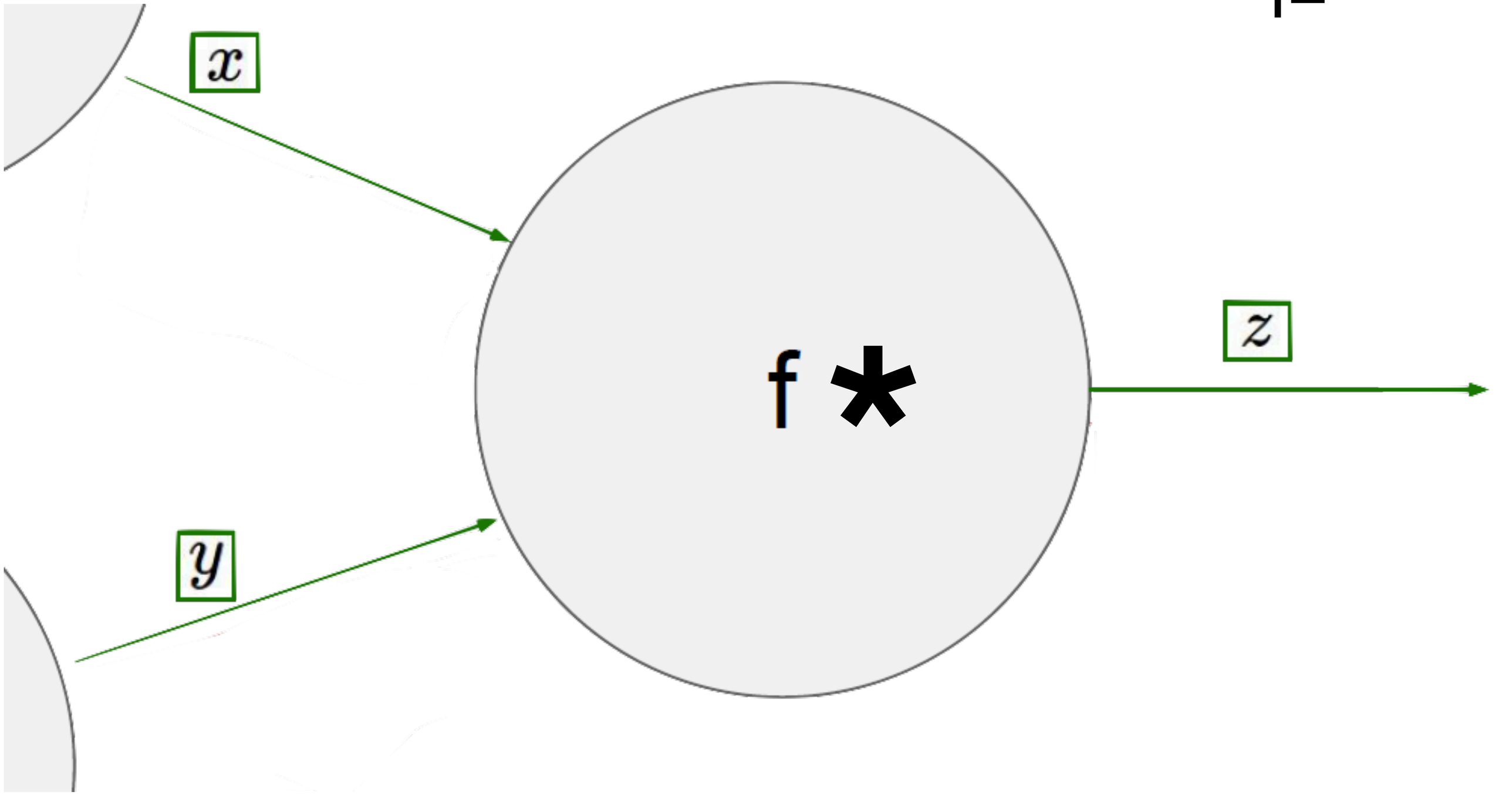
Chain rule

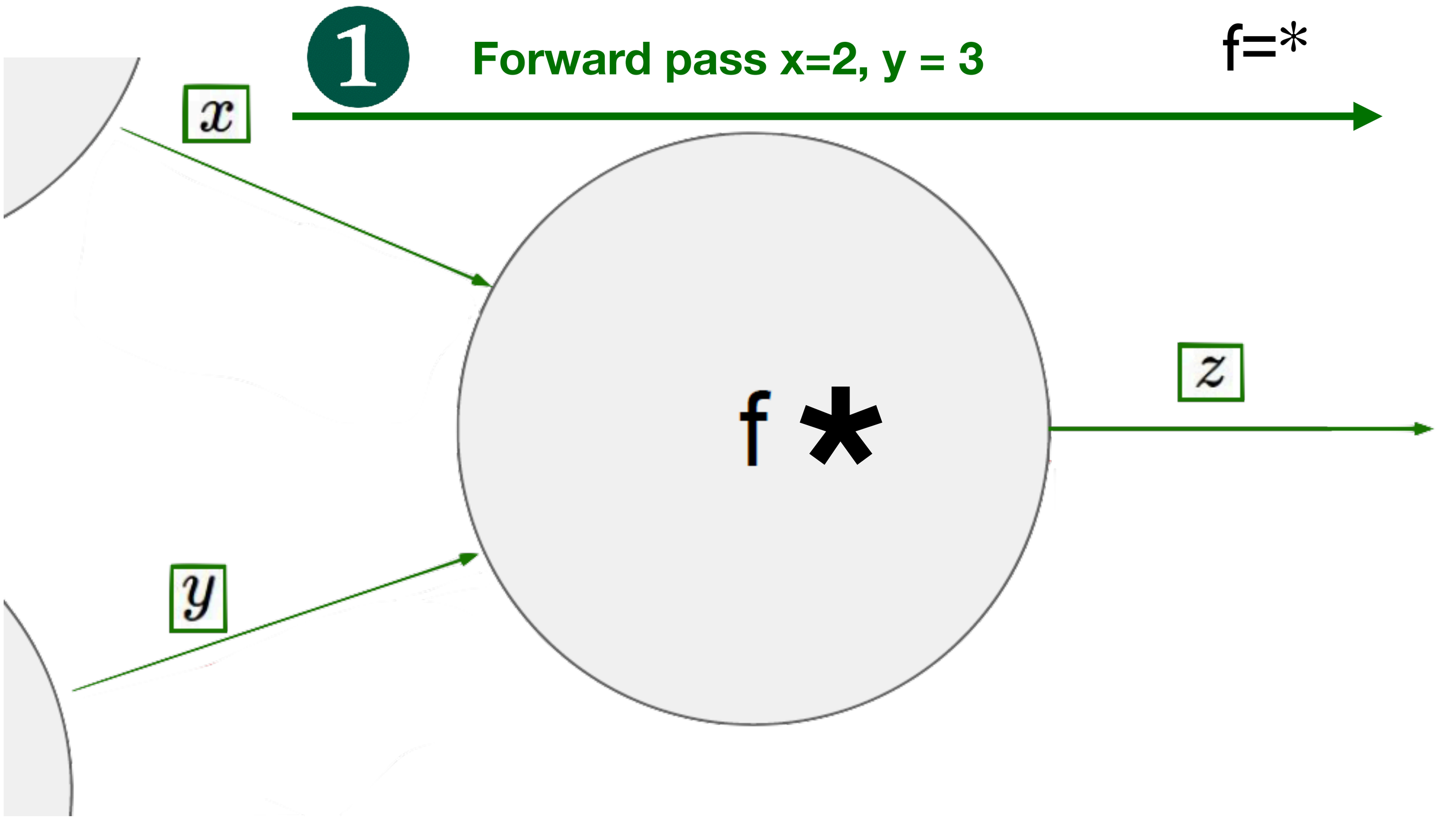

x
$\frac{\partial L}{\partial x}$ =
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y}$ =

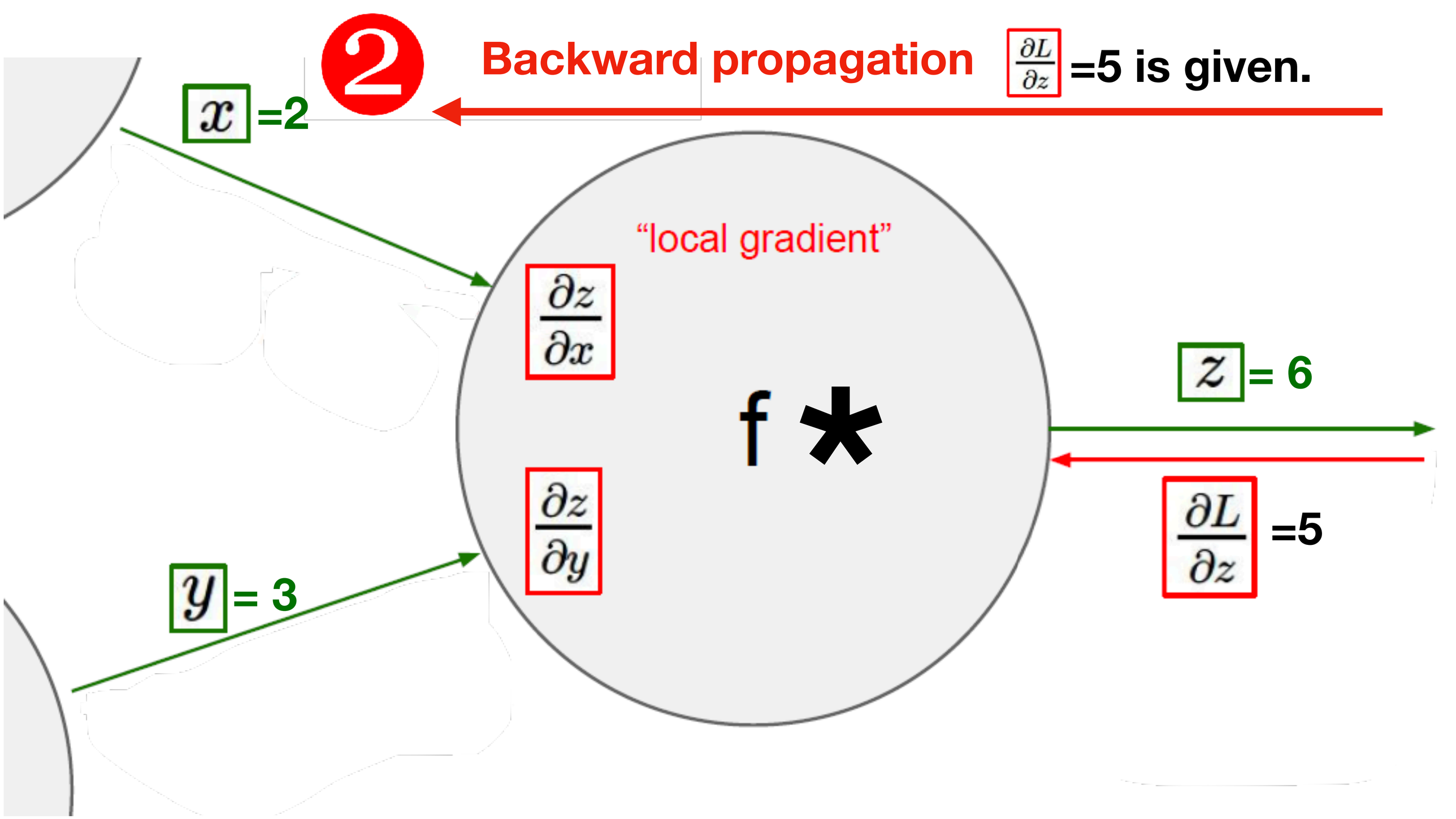
Chain rule

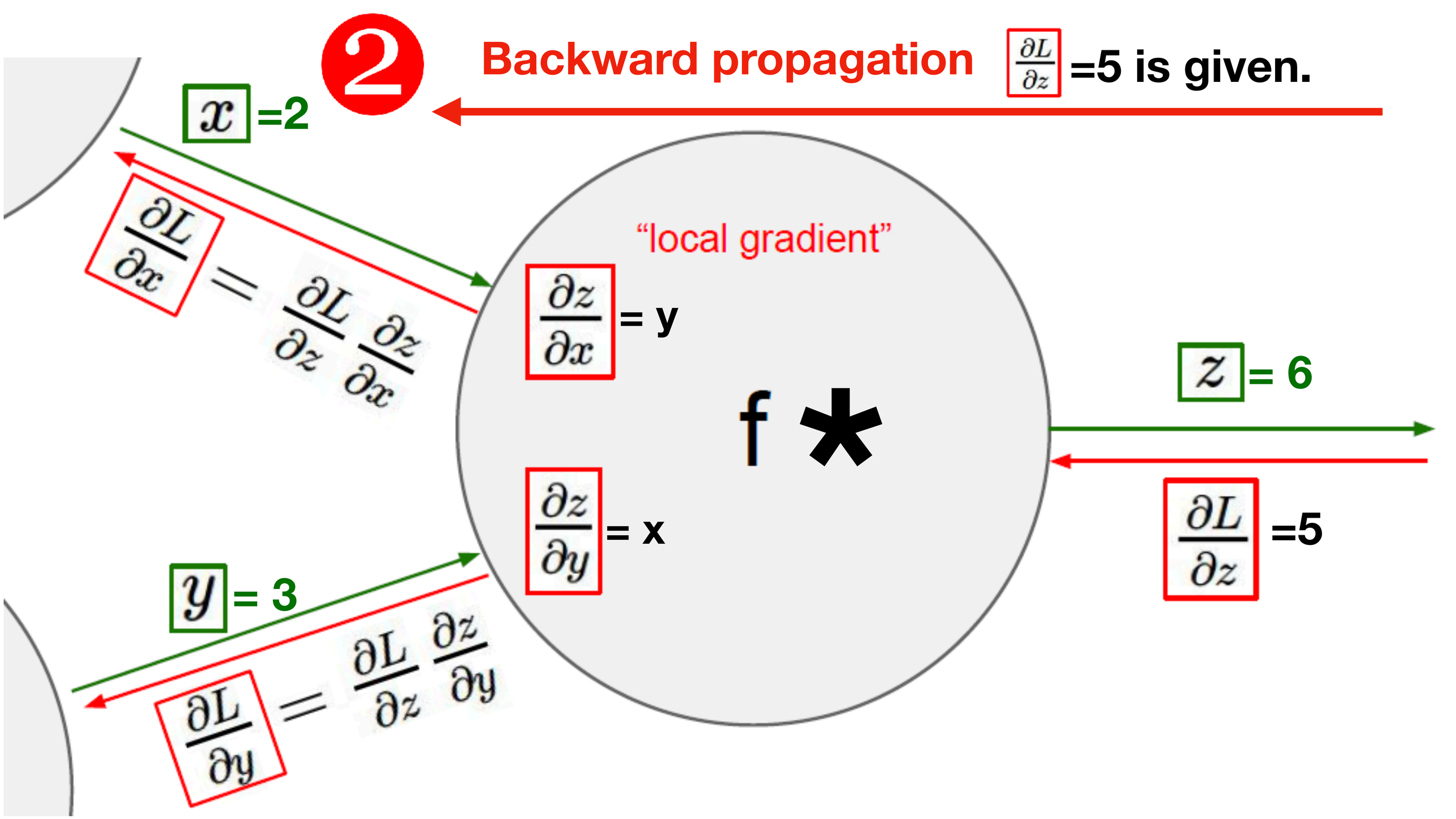

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial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

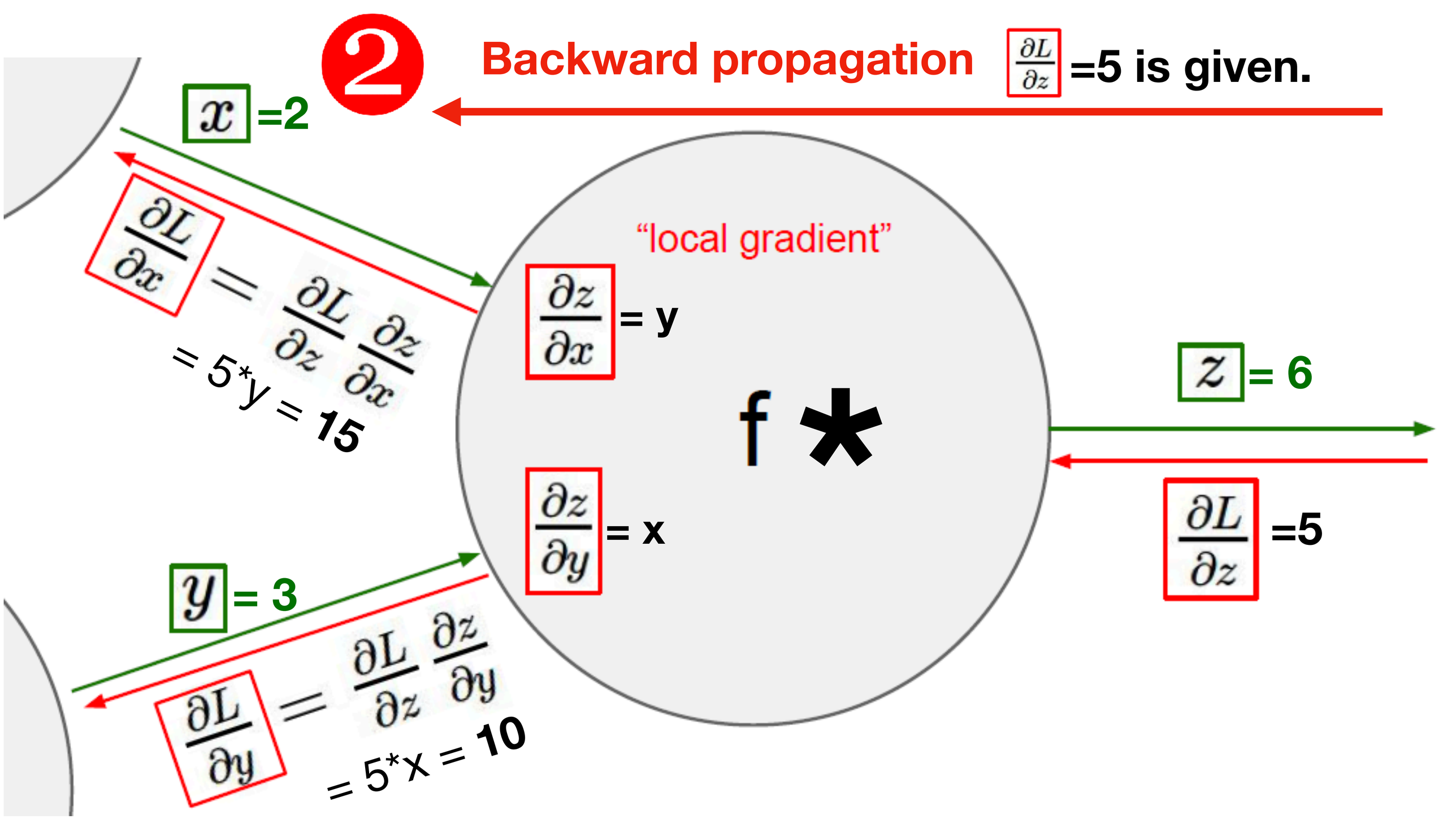
f=*




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
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

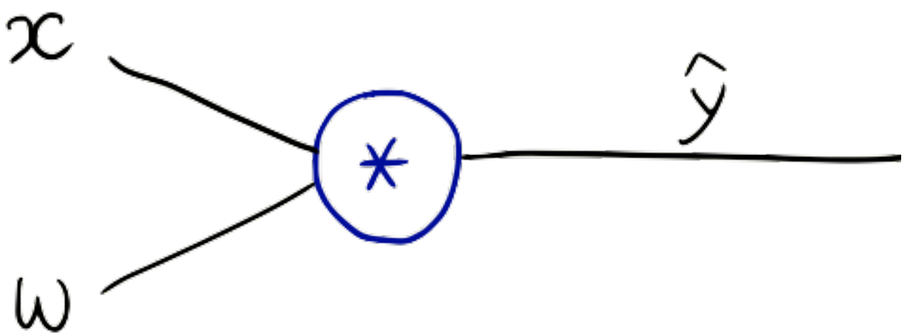

2
Backward propagation
$\frac{\partial L}{\partial z}$ =5 is given.
x =2
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
= 5*y = 15
"local gradient"
$\frac{\partial z}{\partial x}$ = y
f *
$\frac{\partial z}{\partial y}$ = x
z = 6
$\frac{\partial L}{\partial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

Computational graph

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

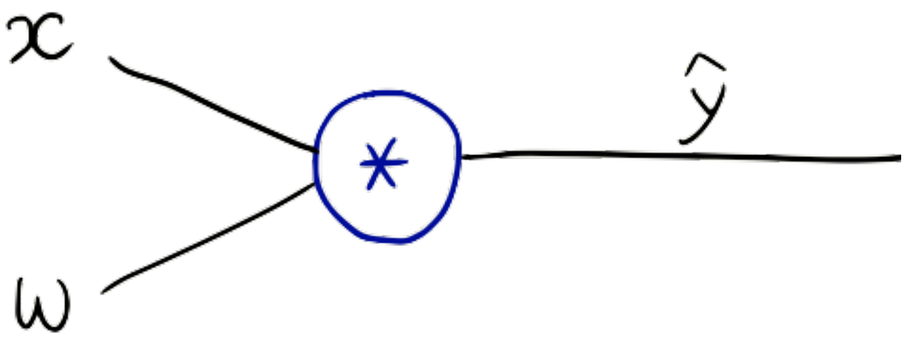
Computational graph

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



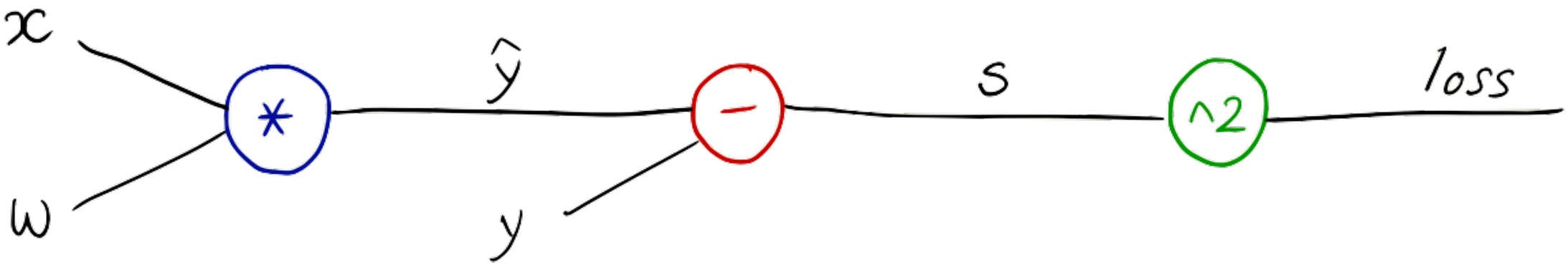
Computational graph

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

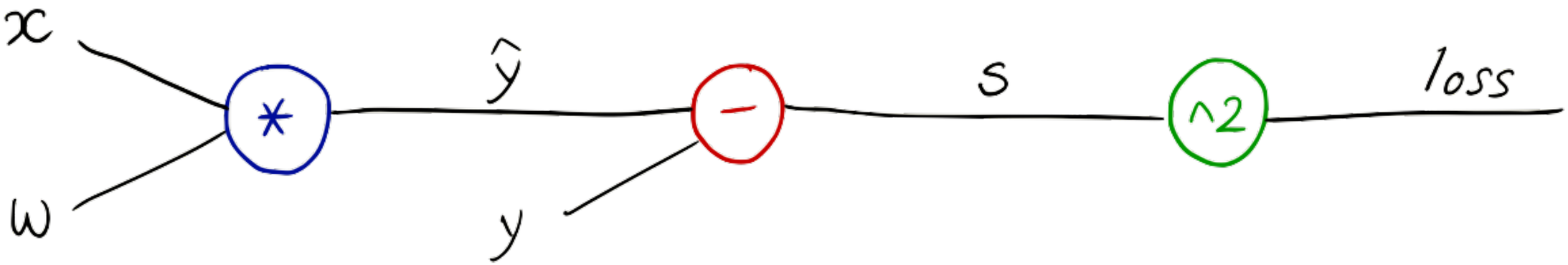


Computational graph

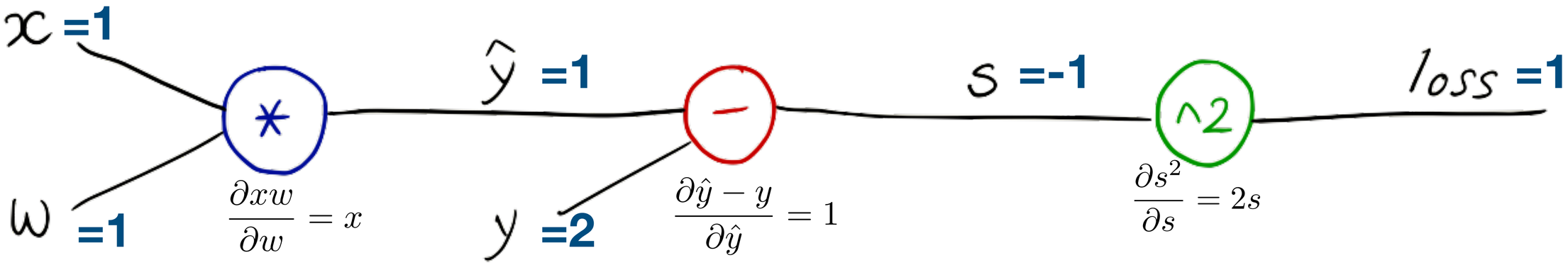
$$\hat{y} = x * w \qquad loss = (\hat{y} - y)^2 = (x * w - y)^2$$



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

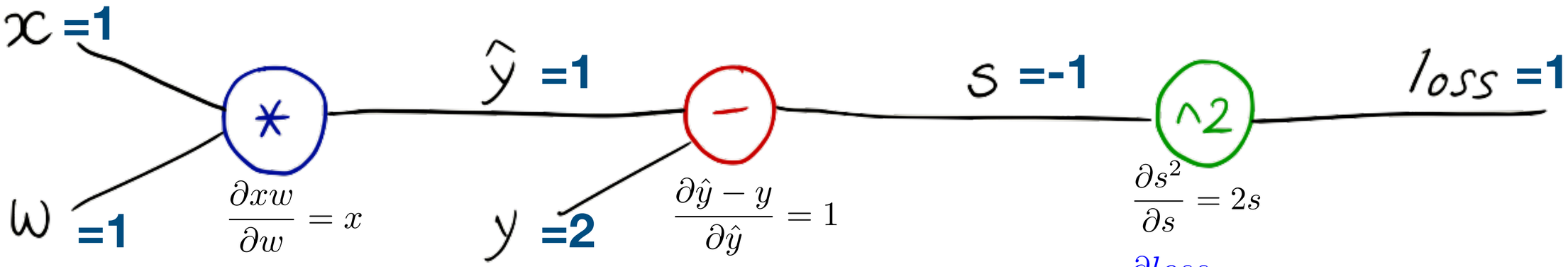


2 Backward propagation



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

2 Backward propagation



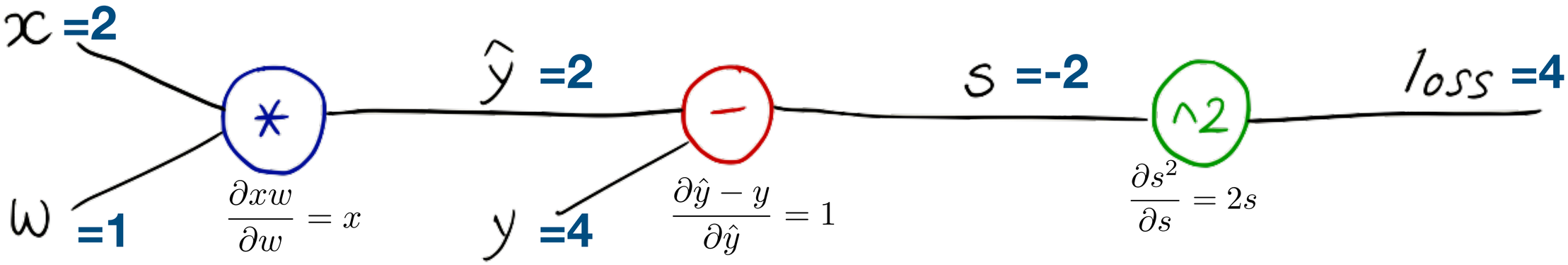
$$\frac{\partial loss}{\partial s} = 2s = -2$$

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

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

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

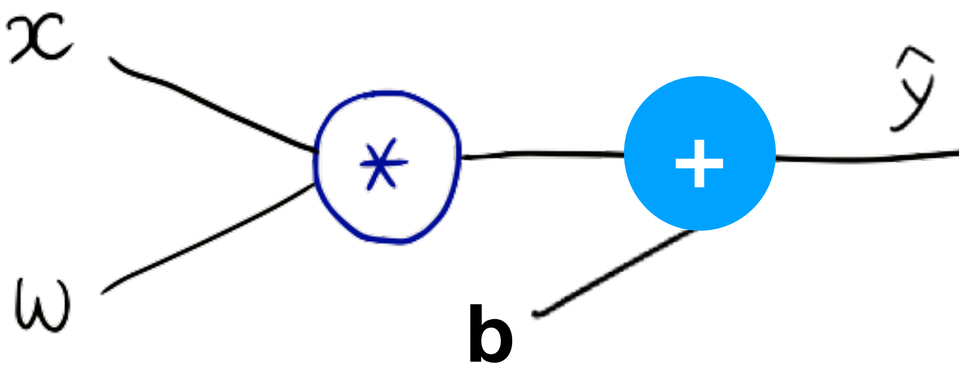


$$\frac{\partial loss}{\partial w} =$$

Exercise 2: x = 1, y=2, *w=1*, *b*=2

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

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



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))
```

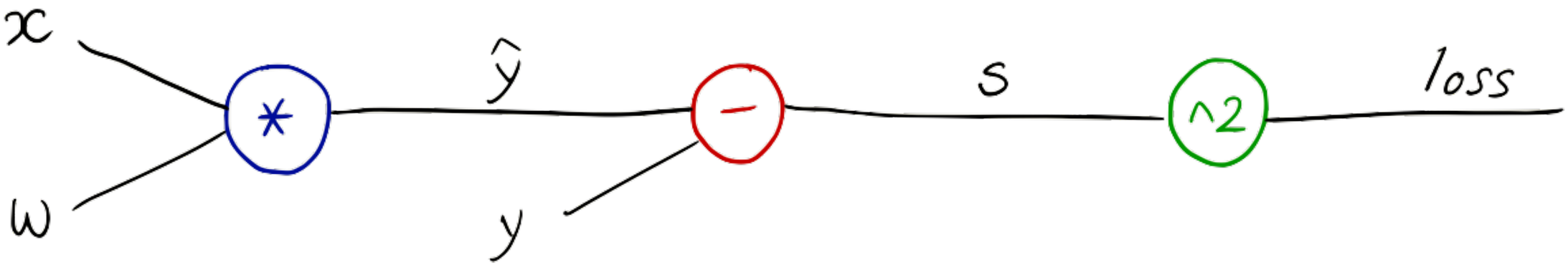
Output
(from numeric gradient computation)

```
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)

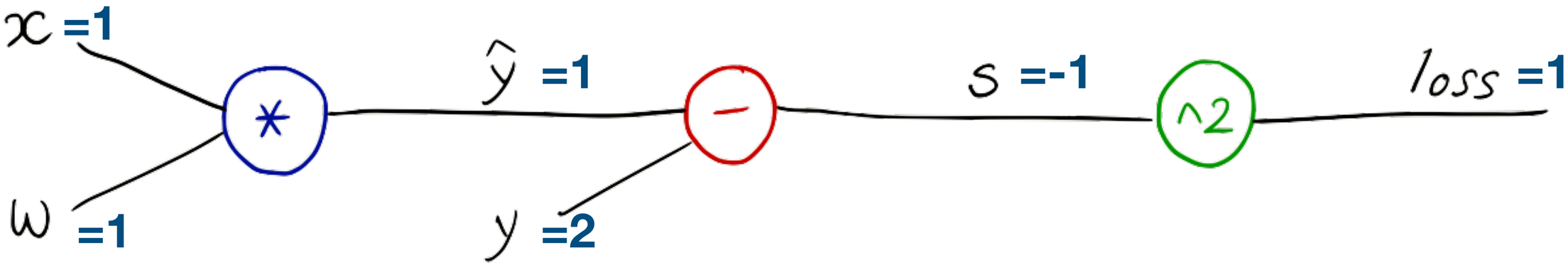

```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.84
    grad:  3.0 6.0 -16.2288
progress: 0 4.919240100095999
    grad:  1.0 2.0 -1.478624
    grad:  2.0 4.0 -5.796206079999999
    grad:  3.0 6.0 -11.998146585599997
progress: 1 2.688769240265834
    grad:  1.0 2.0 -1.093164466688
    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

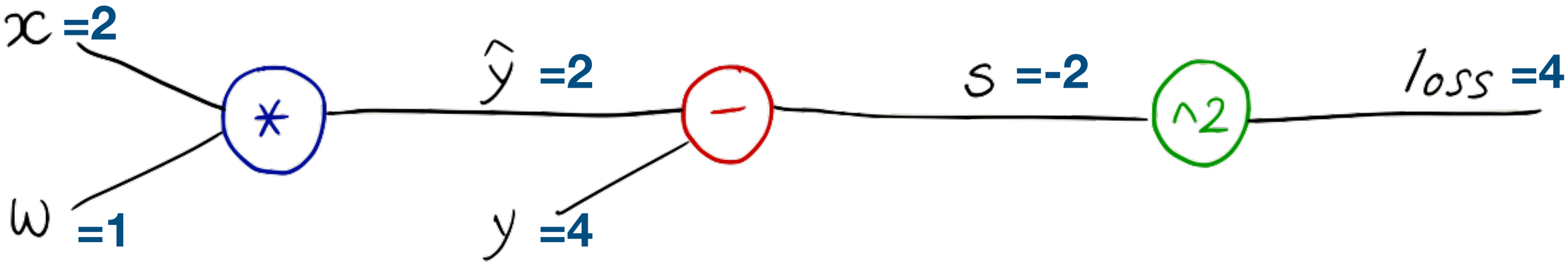
```
w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value
l = loss(x=1, y=1)
```



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

Back propagation

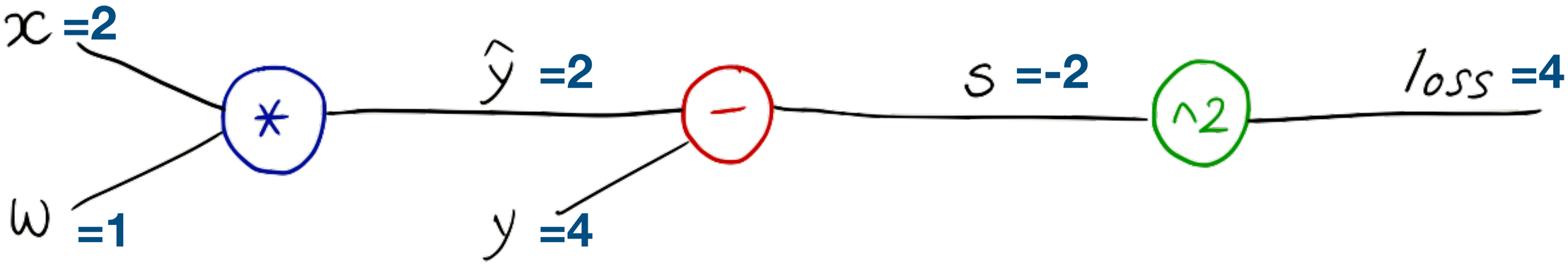
```
l.backward()

# 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

Call for Comments

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**

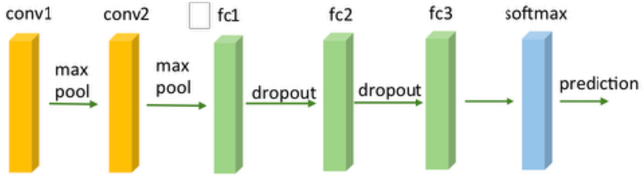
2 **Construct loss and optimizer (select from PyTorch API)**

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()
```

1 **Design your model using class**




```
########################################################################
# 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)
```

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

```
########################################################################
# 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')
```

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

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

Building fun models

- 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

Three simple steps

Design your model using class

Construct loss and optimizer (select from PyTorch API)

Training cycle (forward, backward, update)

WHAT NEXT

Lecture 6: Logistic regression

ML/DL for Everyone with PYTORCH

Lecture 6:
Logistic Regression

Call for Comments

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

Binary prediction (0 or 1) is very useful!

- Spent n hours for study, pass or fail?
- GPA and GRE scores for the HKUST PHD program, admit or not?
- Soccer game against Japan, win or lose?
- She/he looks good, propose or not?
- …

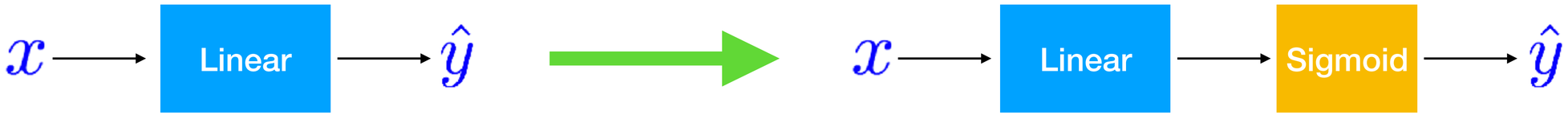
Linear model

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

Logistic regression: pass/fail (0 or 1)

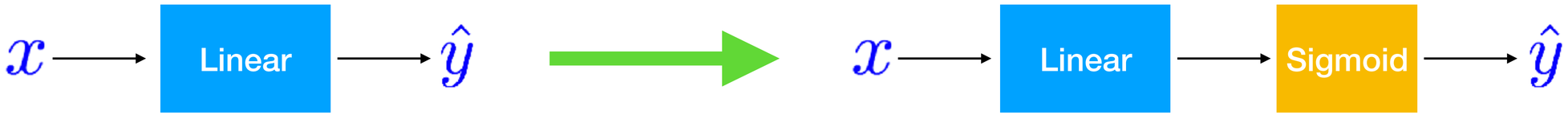
Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?

Logistic regression: pass/fail (0 or 1)

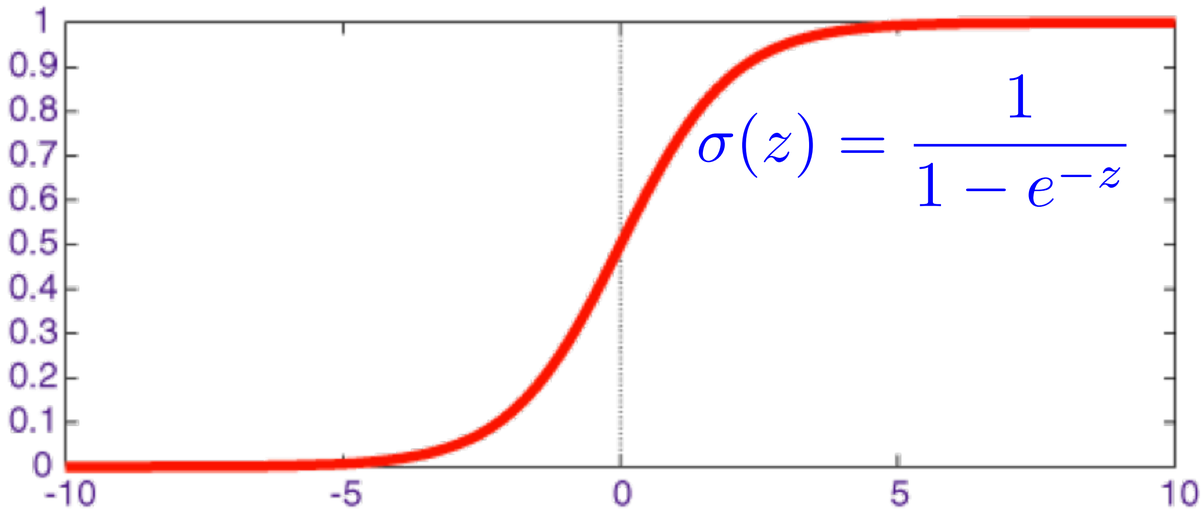


Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?

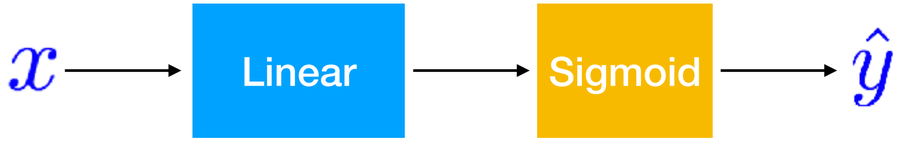
Meet sigmoid



Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



Meet sigmoid

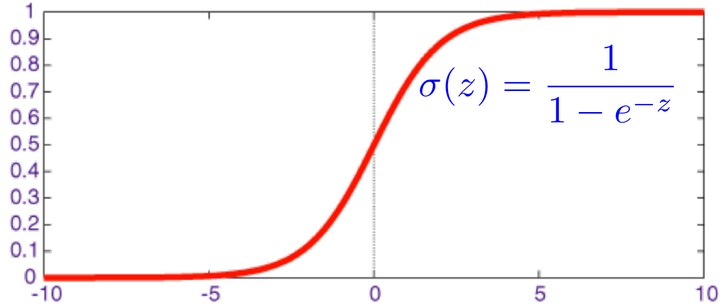


$$\hat{y} = x * w + b$$

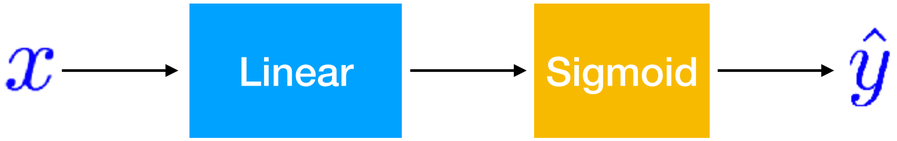
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

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

Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



Meet Cross Entropy Loss



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

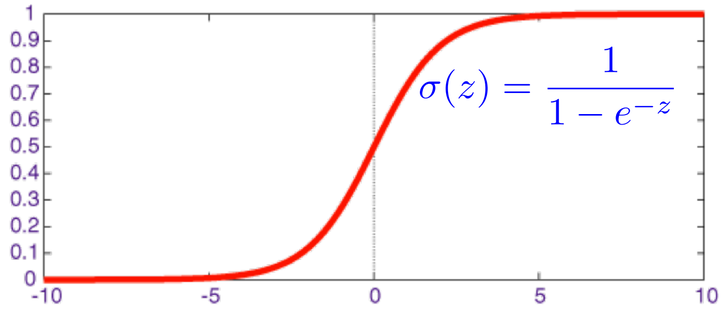
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

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

$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

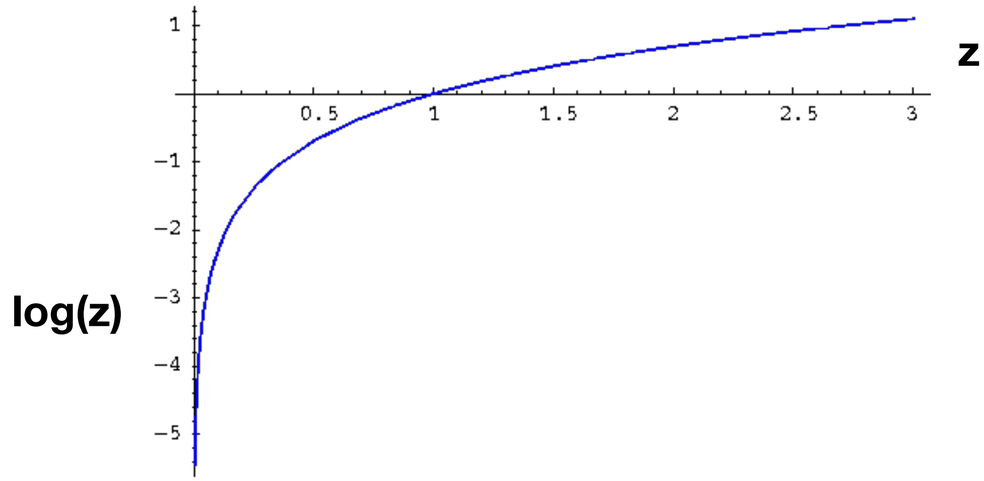
Hours (x)	Points	fail/pass
1	2	0
2	4	0
3	6	1
4	?	?



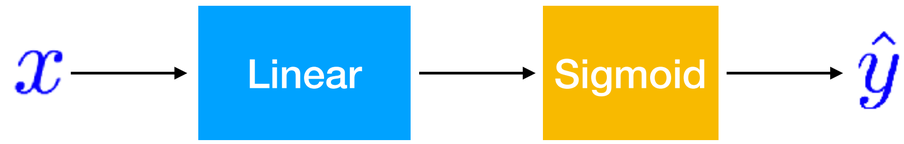
(Binary) Cross Entropy Loss

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

y	y_pred	loss
0	0.2	
0	0.8	
1	0.1	
1	0.9	



Logistic regression



$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

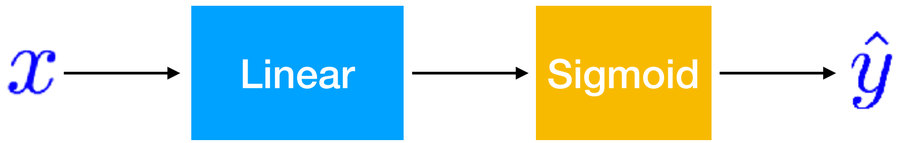
$$\hat{y} = \sigma(x * w + b)$$

class torch.nn.Sigmoid [source]

Applies the element-wise function $f(x) = 1/(1 + exp(-x))$

```
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
        self.sigmoid = torch.nn.Sigmoid()

    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.sigmoid(self.linear(x))
        return y_pred
```

Logistic regression



$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

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

class torch.nn.Sigmoid [source]

Applies the element-wise function $f(x) = 1/(1 + exp(-x))$

```
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
        self.sigmoid = torch.nn.Sigmoid()

    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.sigmoid(self.linear(x))
        return y_pred
```

$$loss = -\frac{1}{N}\sum_{n=1}^{N} y_n \log \hat{y}_n + (1 - y_n) \log(1 - \hat{y}_n)$$

class torch.nn.BCELoss(*weight=None, size_average=True*) [source]

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

$$loss(o, t) = -1/n \sum_i (t[i] * log(o[i]) + (1 - t[i]) * log(1 - o[i]))$$

```
criterion = torch.nn.BCELoss(size_average=True)
```

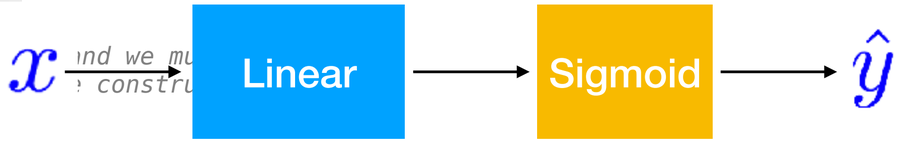
```
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

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
        self.sigmoid = torch.nn.Sigmoid()

    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.sigmoid(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.BCELoss(size_average=True)
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([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

Logistic regression

Logistic regression

```
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

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 a
        self.sigmoid = torch.nn.Sigmoid()

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

# our model
model = Model()

# Construct our loss function and an Optimizer. The ca
# in the SGD constructor will contain the learnable pa
# nn.Linear modules which are members of the model.
criterion = torch.nn.BCELoss(size_average=True)
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([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, 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)**

Logistic regression

```python
x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0], [4.0]]))
y_data = Variable(torch.Tensor([[0.], [0.], [1.], [1.]]))

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
        self.sigmoid = torch.nn.Sigmoid()

    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.sigmoid(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.BCELoss(size_average=True)
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([[0.5]]))
print("predict (after training)", 0.5, model.forward(hour_var).data[0][0])
hour_var = Variable(torch.Tensor([[7.0]]))
print("predict (after training)", 7.0, model.forward(hour_var).data[0][0])
```

```
0 1.6369143724441528
1 1.6119738817214966
2 1.5872894525527954
3 1.5628681182861328
4 1.5387169122695923
5 1.514843225479126
6 1.4912540912628174
7 1.467956781387329
8 1.4449583292007446
9 1.4222657680511475
10 1.3998862504959106
...
…
484 0.5245369672775269
485 0.5243527293205261
486 0.5241686701774597
487 0.5239847302436829
488 0.5238009095191956
489 0.5236172080039978
490 0.5234336256980896
491 0.523250162601471
492 0.5230668187141418
493 0.5228836536407471
494 0.5227005481719971
495 0.5225176215171814
496 0.5223348140716553
497 0.5221521258354187
498 0.5219695568084717
499 0.5217871069908142
predict (after training) 0.5 0.3970
predict (after training) 7.0 0.9398

Process finished with exit code 0
```

WHAT NEXT?

Lecture 7:
Wide and Deep

ML/DL for Everyone with PYTORCH

Lecture 7:
Wide & Deep

Call for Comments

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

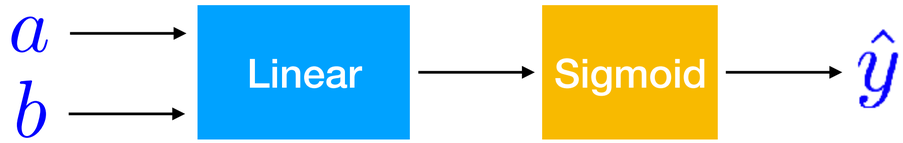
HKUST PHD Program Application

GPA (a)	Admission?
2.1	0
4.2	1
3.1	0
3.3	1


```
x_data =[[2.1],   y_data =[[0.0],
         [4.2],            [1.0],
         [3.1],            [0.0],
         [3.3]]            [1.0]]
```

GPA enough?
How about experience and others?

GPA (a)	Experience (b)	Admission?
2.1	0.1	0
4.2	0.8	1
3.1	0.9	0
3.3	0.2	1



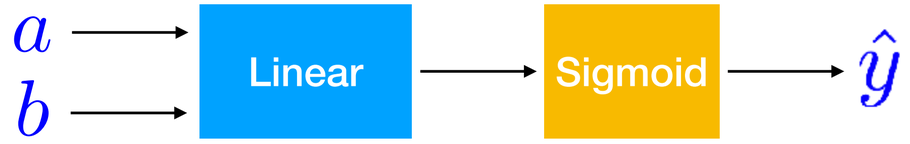

```
x_data =[[2.1, 0.1],
         [4.2, 0.8],
         [3.1, 0.9],
         [3.3, 0.2]]
```

```
y_data =[[0.0],
         [1.0],
         [0.0],
         [1.0]]
```

Matrix Multiplication

```
x_data =[[2.1, 0.1],
         [4.2, 0.8],
         [3.1, 0.9],
         [3.3, 0.2]]
```

```
y_data =[[0.0],
         [1.0],
         [0.0],
         [1.0]]
```

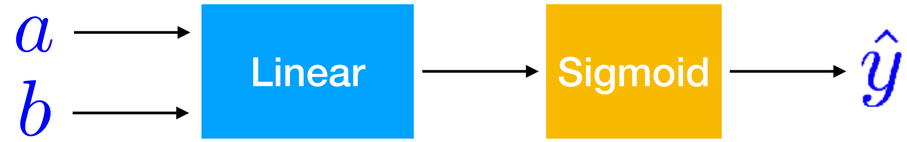


$$\underbrace{\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \cdots & \cdots \\ a_n & b_n \end{bmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{w \in \mathbb{R}^{2 \times 1}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \cdot\cdot \\ y_n \end{bmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

Matrix Multiplication

```
x_data =[[2.1, 0.1],
         [4.2, 0.8],
         [3.1, 0.9],
         [3.3, 0.2]]
```

```
y_data =[[0.0],
         [1.0],
         [0.0],
         [1.0]]
```

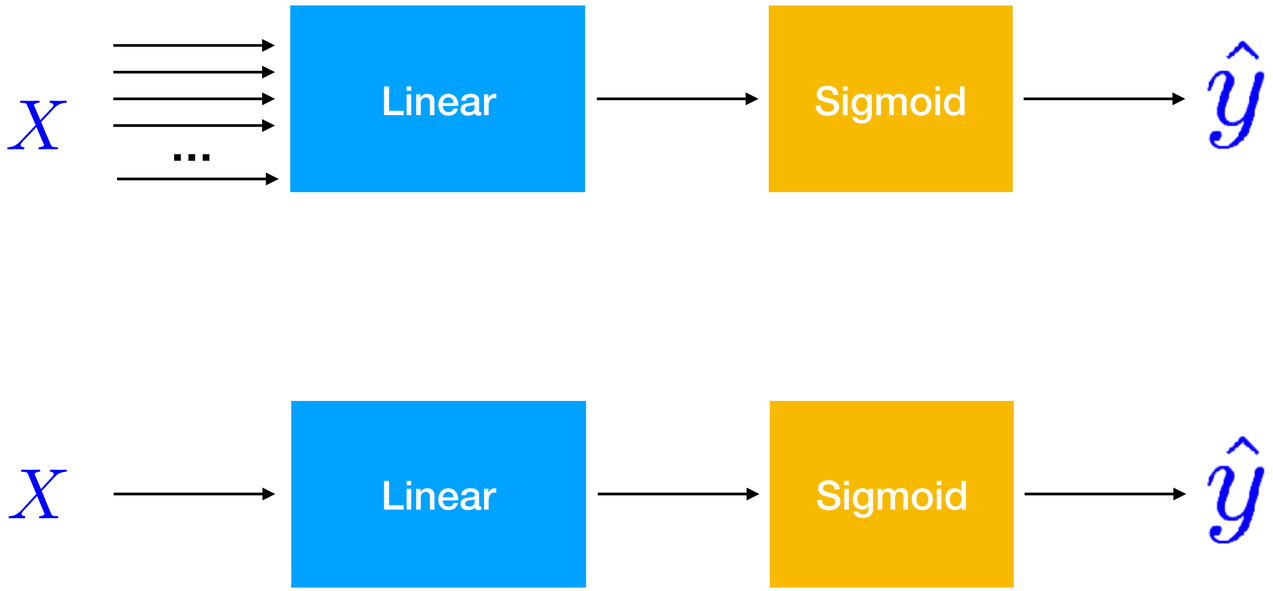


$$\underbrace{\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \cdots & \cdots \\ a_n & b_n \end{bmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{w \in \mathbb{R}^{2 \times 1}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \cdot\cdot \\ y_n \end{bmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

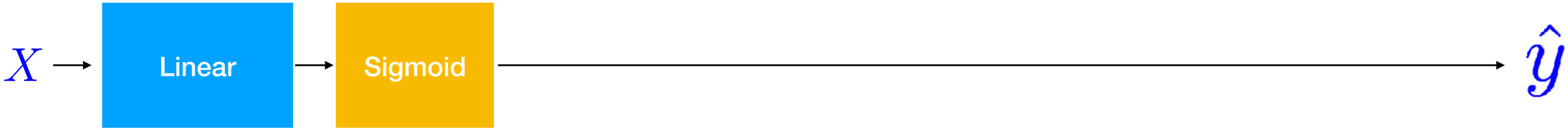
$$XW = \hat{Y}$$

```
linear = torch.nn.Linear(2, 1)
y_pred = linear(x_data)
```

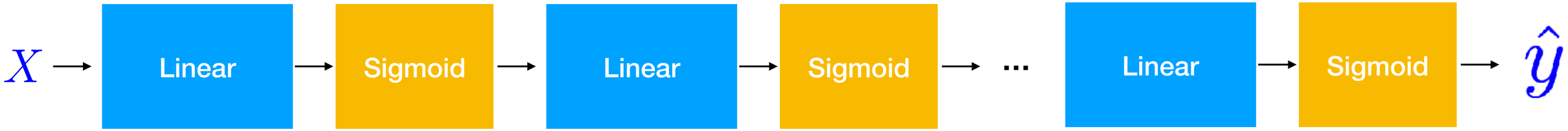
Go Wide!



Go Deep!



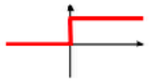
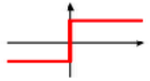
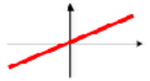
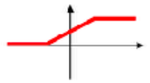
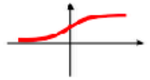
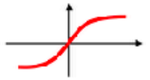
Go Deep!




```
sigmoid = torch.nn.Sigmoid()

l1 = torch.nn.Linear(2, 2)
l2 = torch.nn.Linear(2, 2)
l3 = torch.nn.Linear(2, 1)

out1    = sigmoid(l1(x_data))
out2    = sigmoid(l2(out1))
y_pred = sigmoid(l3(out2)
```

Sigmoid Activation Functions

Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0. \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

http://rasbt.github.io/mlxtend/user_guide/general_concepts/activation-functions/

⊟ Non-linear Activations

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

Classifying Diabetes

-0.411765	0.165829	0.213115	0	0	-0.23696	-0.894962	-0.7	1
-0.647059	-0.21608	-0.180328	-0.353535	-0.791962	-0.0760059	-0.854825	-0.833333	0
0.176471	0.155779	0	0	0	0.052161	-0.952178	-0.733333	1
-0.764706	0.979899	0.147541	-0.0909091	0.283688	-0.0909091	-0.931682	0.0666667	0
-0.0588235	0.256281	0.57377	0	0	0	-0.868488	0.1	0
-0.529412	0.105528	0.508197	0	0	0.120715	-0.903501	-0.7	1
0.176471	0.688442	0.213115	0	0	0.132638	-0.608027	-0.566667	0
0.176471	0.396985	0.311475	0	0	-0.19225	0.163962	0.2	1

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)

x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

print(x_data.data.shape) # torch.Size([759, 8])
print(y_data.data.shape) # torch.Size([759, 1])
```

Wide & Deep

```
class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    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.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        return y_pred
```

Classifying Diabetes

```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    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.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        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.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Training loop
for epoch in range(100):
        # 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()
```

1 **Design your model using class**

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

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

Lecture 8: DataLoader

ML/DL for Everyone with PYTORCH

Lecture 8: DataLoader

Call for Comments

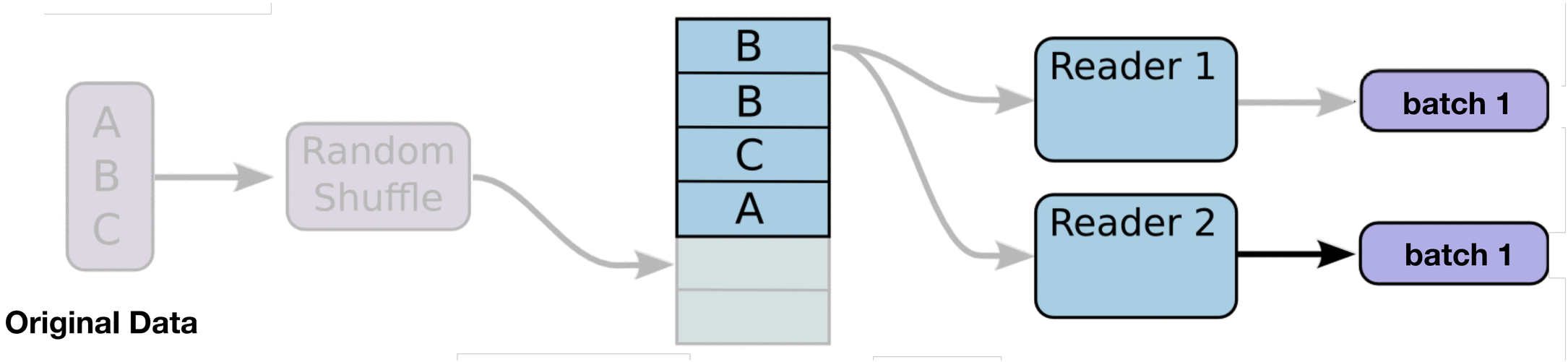
Sung Kim <hunkim+ml@gmail.com> HKUST

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

Manual data feed


```
xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
x_data = Variable(torch.from_numpy(xy[:, 0:-1]))
y_data = Variable(torch.from_numpy(xy[:, [-1]]))

...

# Training loop
for epoch in range(100):
        # 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()
```

DataLoader




```
for i, data in enumerate(train_loader, 0):
    # get the inputs
    inputs, labels = data

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

    # Run your training process
    print(epoch, i, "inputs", inputs.data, "labels", labels.data)
```

Custom DataLoader

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):



    def __getitem__(self, index):
        return

    def __len__(self):
        return


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

1 download, read data, etc.

2 return one item on the index

3 return the data length

Custom DataLoader

```
class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

Classifying Diabetes

```python
# References
# https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/pytorch_basics/main.py
# http://pytorch.org/tutorials/beginner/data_loading_tutorial.html#dataset-class
import torch
import numpy as np
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader


class DiabetesDataset(Dataset):
    """ Diabetes dataset."""

    # Initialize your data, download, etc.
    def __init__(self):
        xy = np.loadtxt('data-diabetes.csv', delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len


dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

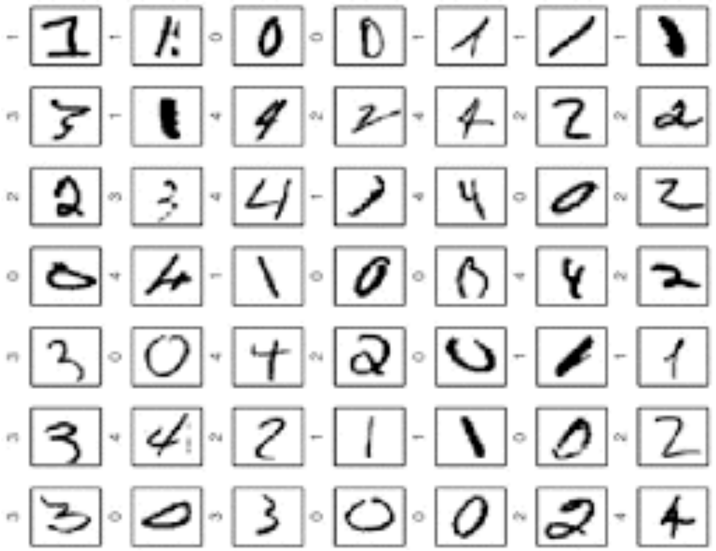
```python
class Model(torch.nn.Module):

    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.l1 = torch.nn.Linear(8, 6)
        self.l2 = torch.nn.Linear(6, 4)
        self.l3 = torch.nn.Linear(4, 1)

        self.sigmoid = torch.nn.Sigmoid()

    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.
        """
        out1 = self.sigmoid(self.l1(x))
        out2 = self.sigmoid(self.l2(out1))
        y_pred = self.sigmoid(self.l3(out2))
        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.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

# Training loop
for epoch in range(2):
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

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

        # Forward pass: Compute predicted y by passing x to the model
        y_pred = model(inputs)

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

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

The following dataset loaders are available

- MNIST and FashionMNIST
- COCO (Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10
- SVHN
- PhotoTour

MNIST dataset loading

```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
      transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False,
      transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)



for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    …
```



The following dataset loaders are available

- MNIST and FashionMNIST
- COCO (Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10
- SVHN
- PhotoTour

Lecture 9:
Softmax Classifier

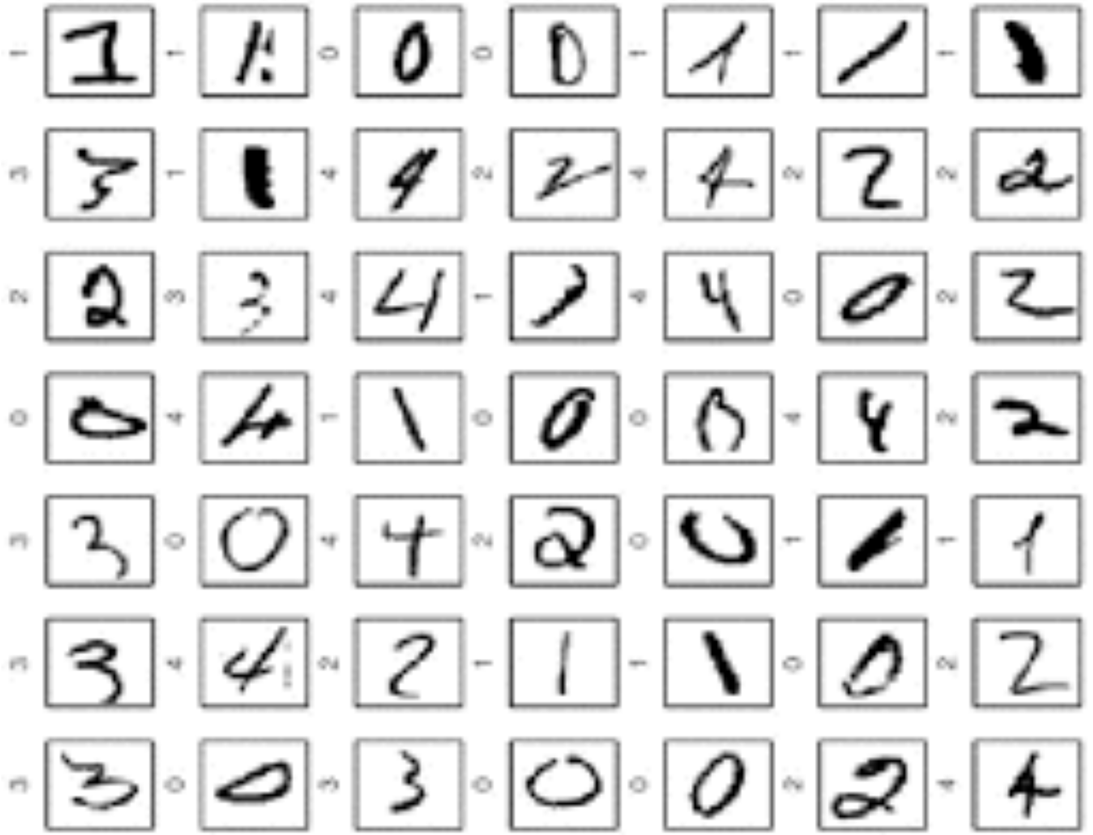
ML/DL for Everyone with PYTORCH

Lecture 9:
Softmax Classifier

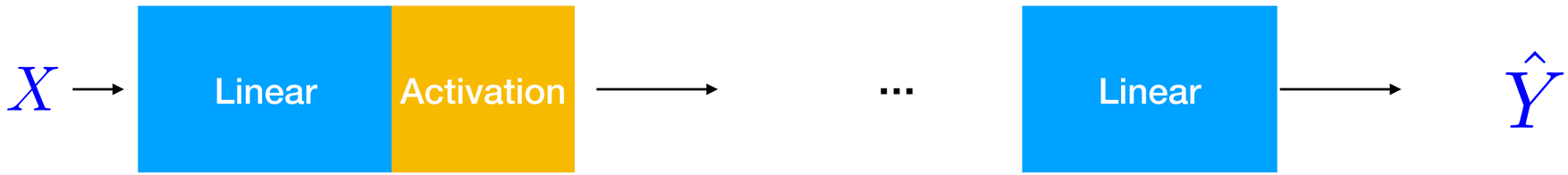
Call for Comments

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

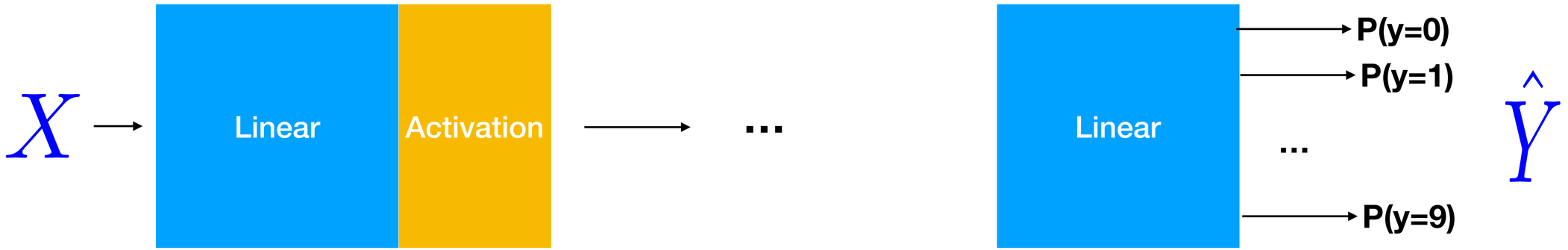
MNIST: 10 labels



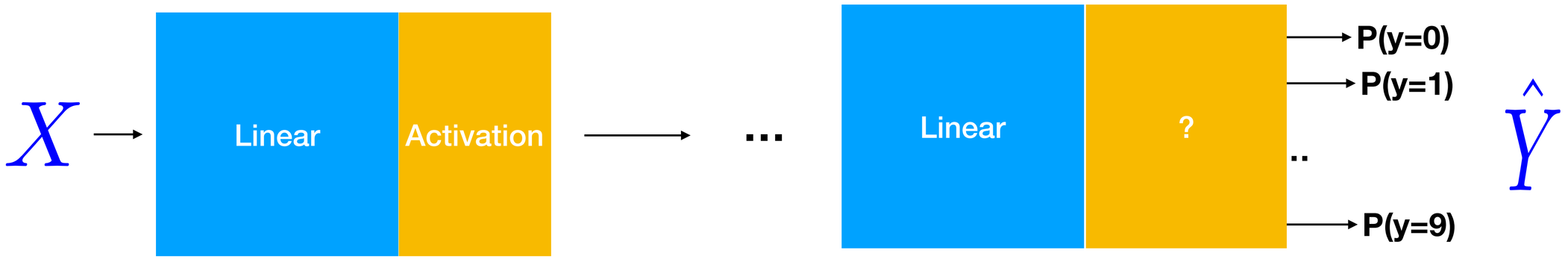
10 labels: 10 outputs



10 labels: 10 outputs

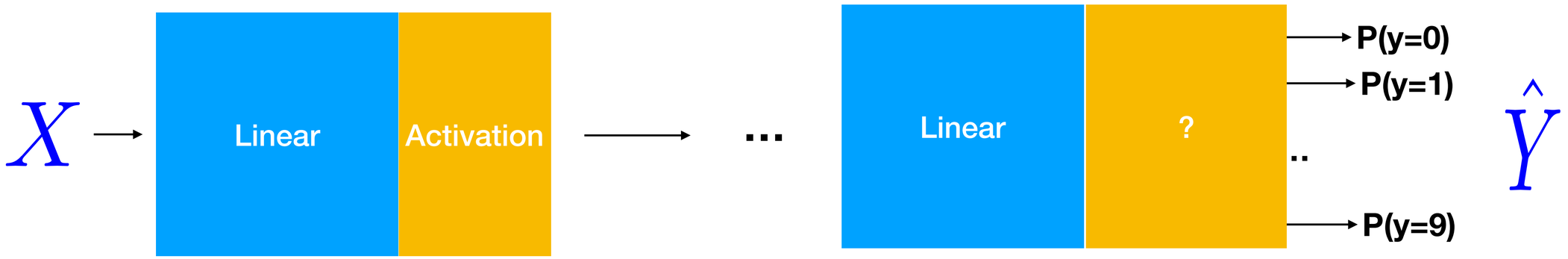


10 outputs



$$\underbrace{\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \cdots & \cdots \\ a_n & b_n \end{bmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{w \in \mathbb{R}^{2 \times 1}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \cdot\cdot \\ y_n \end{bmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

10 outputs

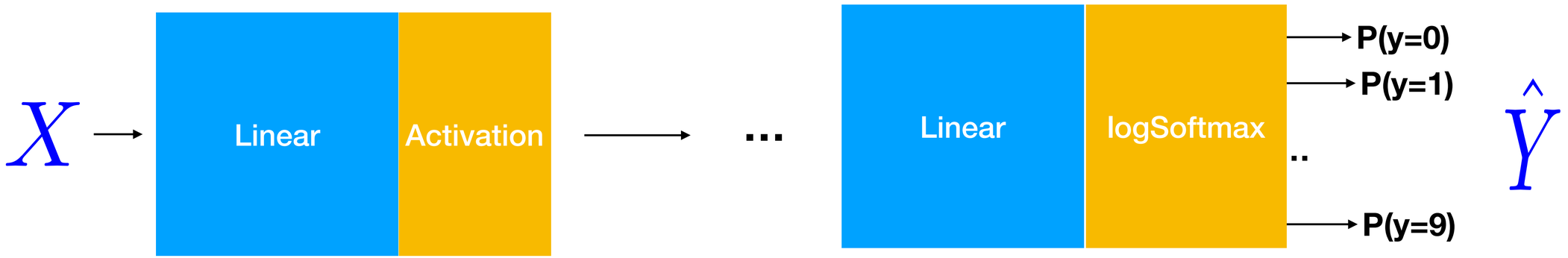


$$\underbrace{\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \cdots & \cdots \\ a_n & b_n \end{bmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}}_{w \in \mathbb{R}^{2 \times 1}} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \cdot\cdot \\ y_n \end{bmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

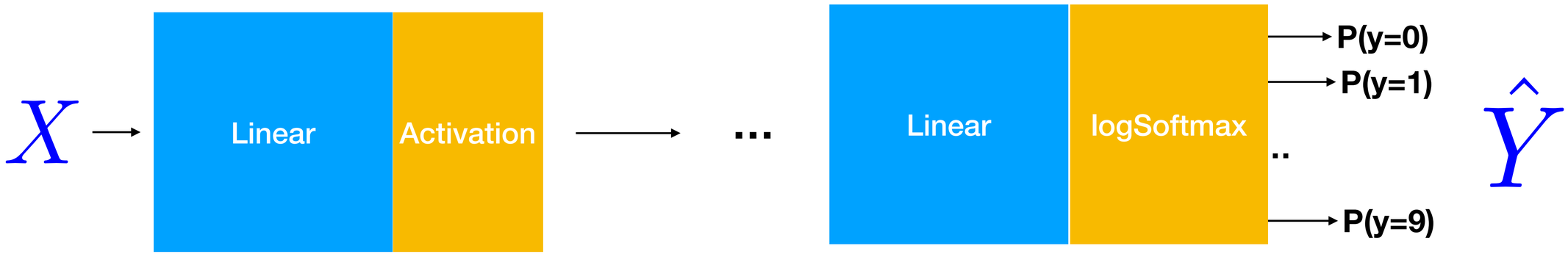
$$\underbrace{\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \\ \cdots & \cdots \\ a_n & b_n \end{bmatrix}}_{x \in \mathbb{R}^{N \times 2}} \begin{bmatrix} ? \end{bmatrix} = y \in R^{N \times 10}$$

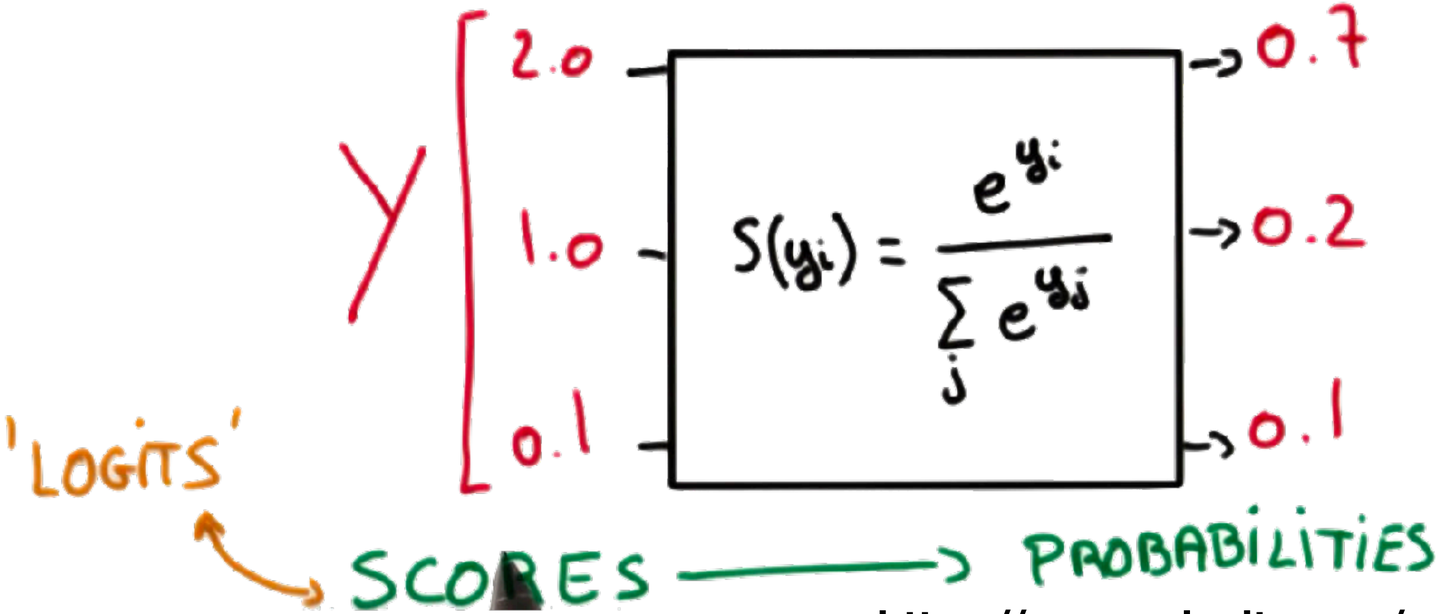
$$w \in R^{N \times ?}$$

(log)Softmax



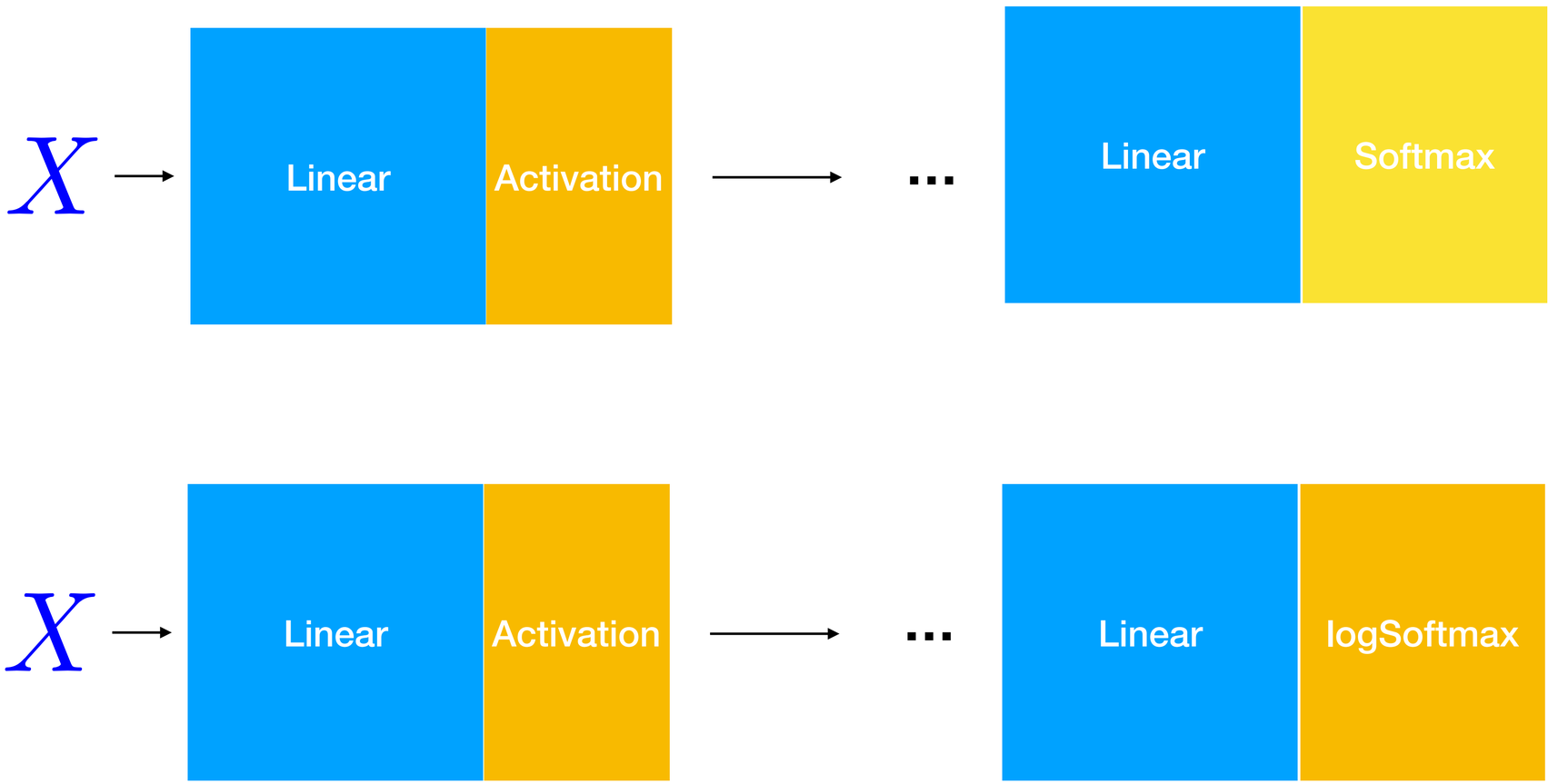
Softmax





https://www.udacity.com/course/deep-learning--ud730

Softmax VS LogSoftmax



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

Softmax VS LogSoftmax

class `torch.nn.Softmax` [source]

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range (0,1) and sum to 1

Softmax is defined as $f_i(x) = exp(x_i)/sum_j exp(x_j)$

Note

This module doesn't work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use Logsoftmax instead (it's faster).

class `torch.nn.LogSoftmax` [source]

Applies the Log(Softmax(x)) function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as

$$f_i(x) = log(exp(x_i)/sum_j exp(x_j))$$

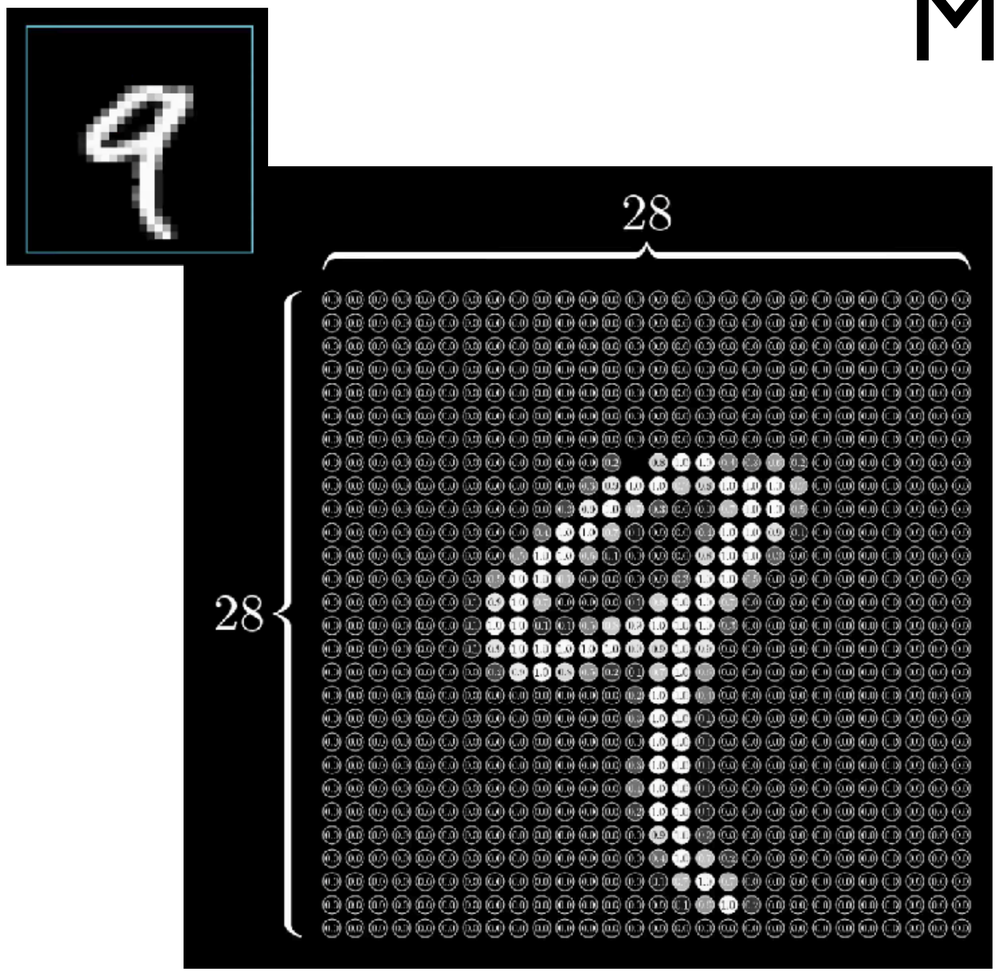
http://pytorch.org/docs/master/nn.html

LogSoftmax with NLLLoss

```
>>> m = nn.LogSoftmax()
>>> loss = nn.NLLLoss()
>>> # input is of size nBatch x nClasses = 3 x 5
>>> input = autograd.Variable(torch.randn(3, 5), requires_grad=True)
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
>>> output = loss(m(input), target)
>>> output.backward()
```

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

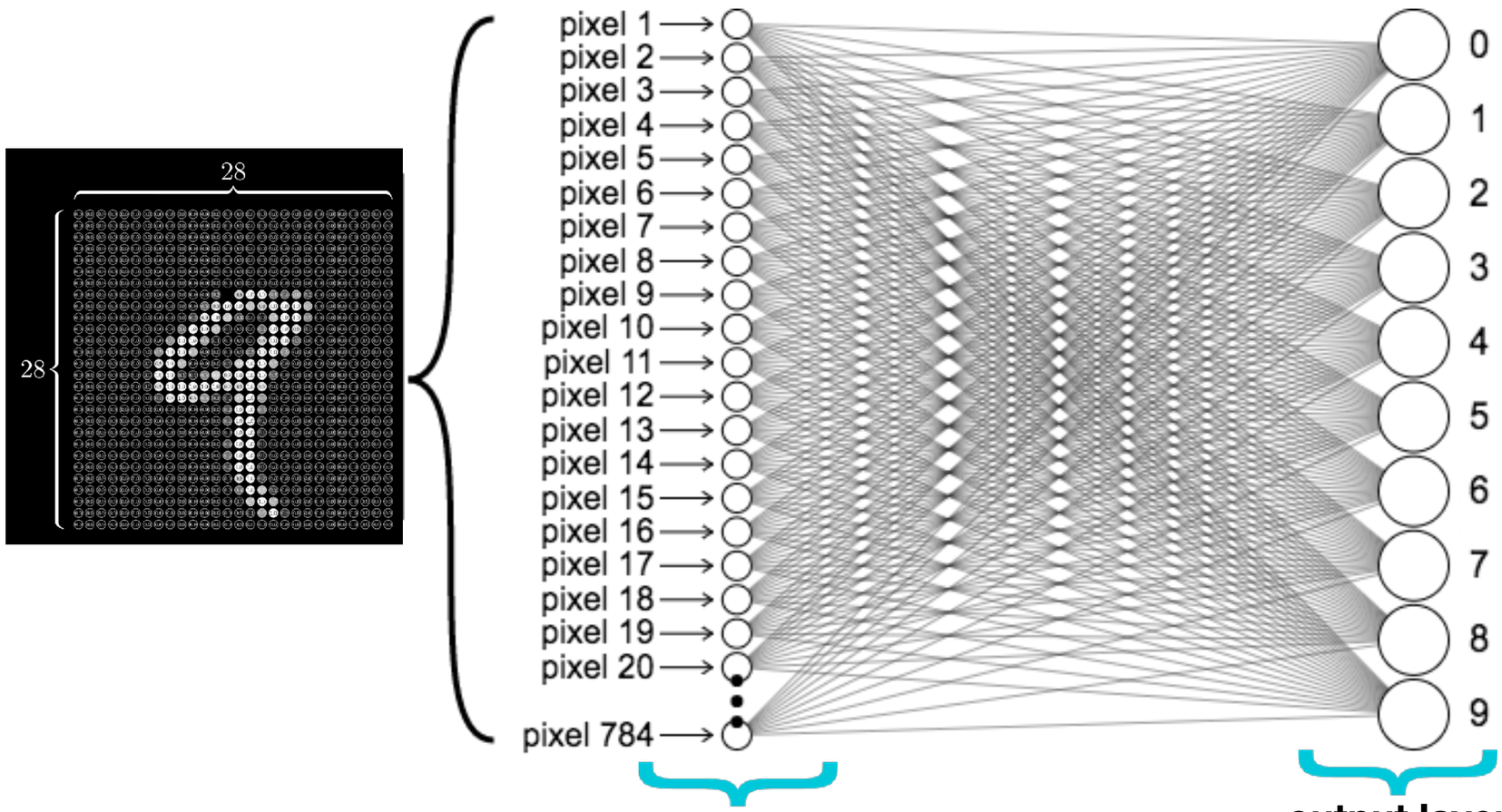
MNIST input



28x28 pixels = 748

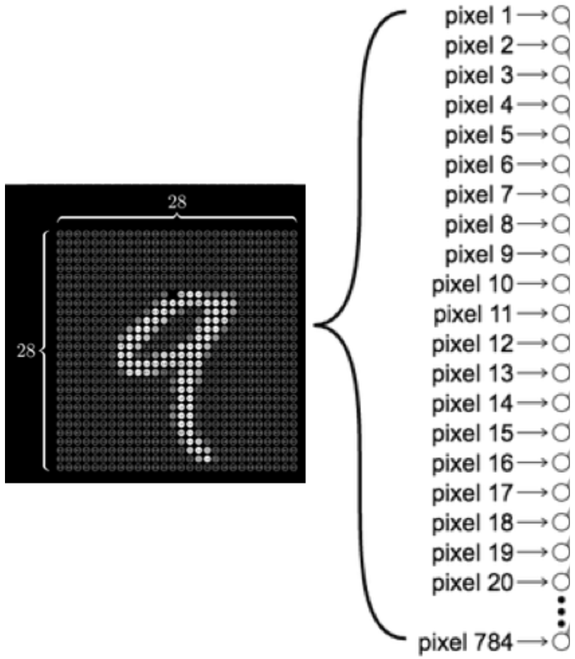
https://www.youtube.com/watch?v=aircAruvnKk&t=1s

MNIST Network



https://ml4a.github.io/ml4a/looking_inside_neural_nets

MNIST Network



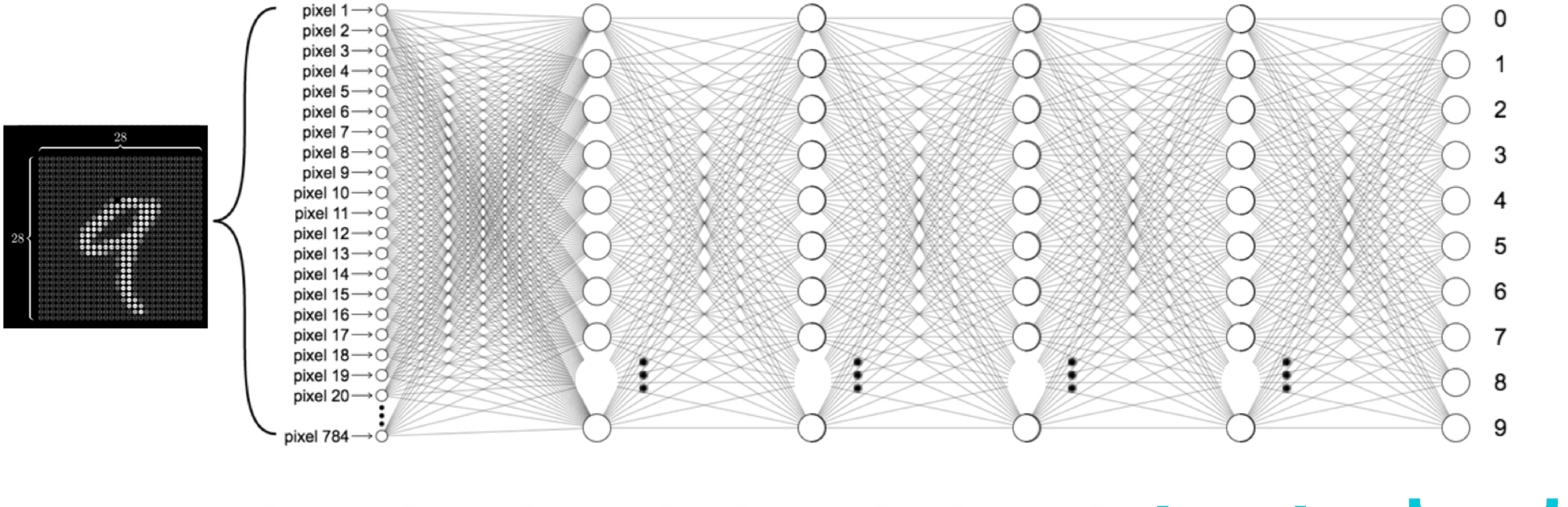
Input layer 784

Hidden layers

output layer
10 (labels)

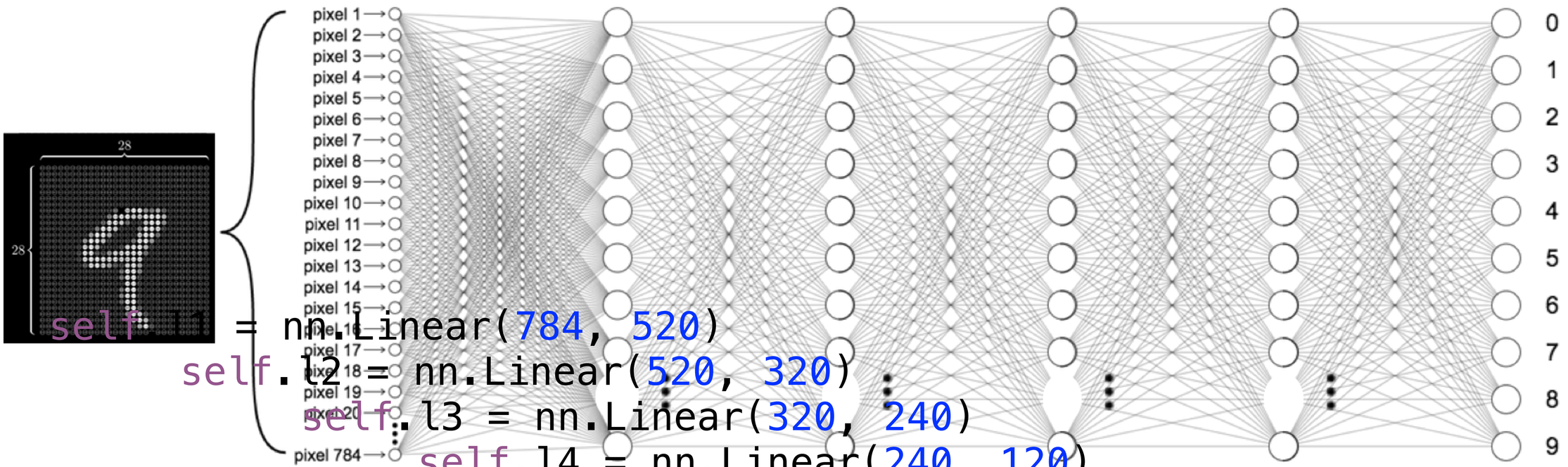
https://ml4a.github.io/ml4a/looking_inside_neural_nets

MNIST Network



https://ml4a.github.io/ml4a/looking_inside_neural_nets

MNIST Network

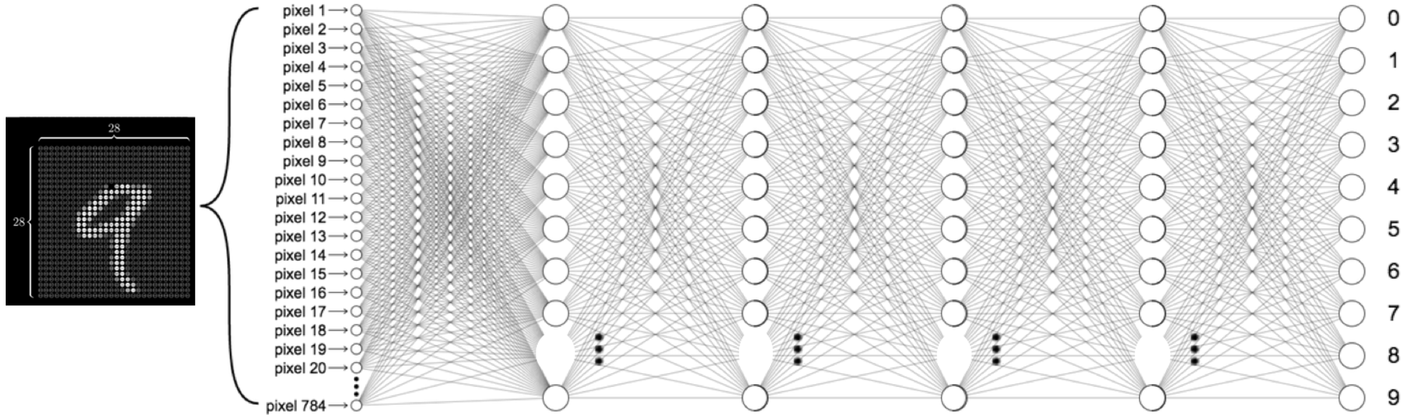



```
self.l1 = nn.Linear(784, 520)
self.l2 = nn.Linear(520, 320)
self.l3 = nn.Linear(320, 240)
self.l4 = nn.Linear(240, 120)
self.l5 = nn.Linear(120, 10)
```

https://ml4a.github.io/ml4a/looking_inside_neural_nets

Softmax & NLL loss

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        x = F.relu(self.l5(x))
        return F.log_softmax(x)
```



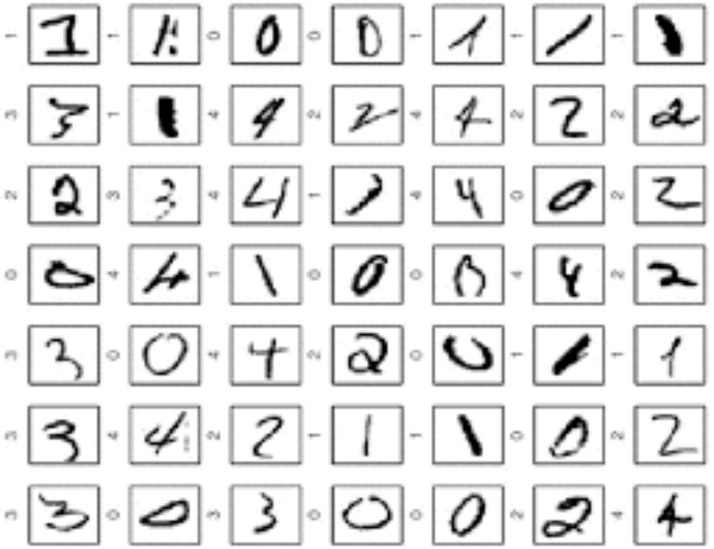
Softmax & NLL loss

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = x.view(-1, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        x = F.relu(self.l5(x))
        return F.log_softmax(x)
```

```
for batch_idx, (data, target) in enumerate(train_loader):
    data, target = Variable(data), Variable(target)
    optimizer.zero_grad()
    output = model(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    optimizer.step()
```

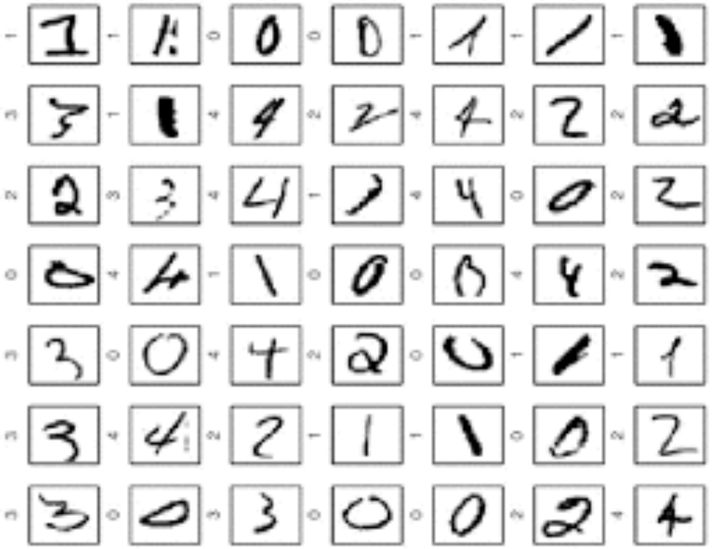
```python
# Training settings
batch_size = 64

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)

class Net(nn.Module):
   def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        x = F.relu(self.l5(x))
        return F.log_softmax(x)

model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 10 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

MNIST Softmax



```
# Training settings
batch_size = 64

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)

class Net(nn.Module):
   def __init__(self):
        super(Net, self).__init__()
        self.l1 = nn.Linear(784, 520)
        self.l2 = nn.Linear(520, 320)
        self.l3 = nn.Linear(320, 240)
        self.l4 = nn.Linear(240, 120)
        self.l5 = nn.Linear(120, 10)

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten the data (n, 1, 28, 28)-> (n, 784)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
        x = F.relu(self.l4(x))
        x = F.relu(self.l5(x))
        return F.log_softmax(x)

model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 10 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

Accuracy?




```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=batch_size, shuffle=True)
```

```
def train(epoch):
    ...
```

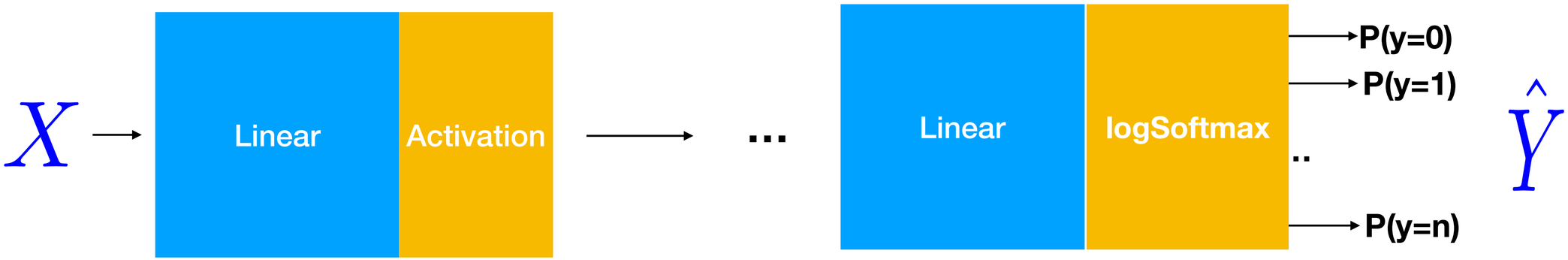
```
def test():
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        # sum up batch loss
        test_loss += F.nll_loss(output, target, size_average=False).data[0]
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```python
def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 10 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))


def test():
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        # sum up batch loss
        test_loss += F.nll_loss(output, target, size_average=False).data[0]
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.
        format(test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

for epoch in range(1, 10):
    train(epoch)
    test()
```

Accuracy?

```
Train Epoch: 9 [46720/60000 (78%)]    Loss: 0.790513
Train Epoch: 9 [47360/60000 (79%)]    Loss: 0.335216
Train Epoch: 9 [48000/60000 (80%)]    Loss: 0.675538
Train Epoch: 9 [48640/60000 (81%)]    Loss: 0.359488
Train Epoch: 9 [49280/60000 (82%)]    Loss: 0.276906
Train Epoch: 9 [49920/60000 (83%)]    Loss: 0.412109
Train Epoch: 9 [50560/60000 (84%)]    Loss: 0.556780
Train Epoch: 9 [51200/60000 (85%)]    Loss: 0.332712
Train Epoch: 9 [51840/60000 (86%)]    Loss: 0.514475
Train Epoch: 9 [52480/60000 (87%)]    Loss: 0.515686
Train Epoch: 9 [53120/60000 (88%)]    Loss: 0.462904
Train Epoch: 9 [53760/60000 (90%)]    Loss: 0.571690
Train Epoch: 9 [54400/60000 (91%)]    Loss: 0.446774
Train Epoch: 9 [55040/60000 (92%)]    Loss: 0.441682
Train Epoch: 9 [55680/60000 (93%)]    Loss: 0.438245
Train Epoch: 9 [56320/60000 (94%)]    Loss: 0.470004
Train Epoch: 9 [56960/60000 (95%)]    Loss: 0.474394
Train Epoch: 9 [57600/60000 (96%)]    Loss: 0.527718
Train Epoch: 9 [58240/60000 (97%)]    Loss: 0.614899
Train Epoch: 9 [58880/60000 (98%)]    Loss: 0.512663
Train Epoch: 9 [59520/60000 (99%)]    Loss: 0.474054

Test set: Average loss: 0.5403, Accuracy: 7820/10000 (78%)
```

Multiple label prediction?
No problem! Use logSoftmax + NLLLoss



NLLLoss

With NLLLoss

Lecture 9: CNN

ML/DL for Everyone with PYTORCH

Lecture 10:
CNN

Call for Comments

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

ML/DL for Everyone with PYTORCH

Lecture 11:
RNN

Call for Comments

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

ML/DL for Everyone with PYTORCH

Lecture 12: NSML

Call for Comments

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

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

Many more fun networks

- Language Model (RNN-LM)
- Generative Adversarial Network
- 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