Computer Vision

Lecture 03: Machine Learning Basics-2

Machine learning

- **x**: data, **y***: prediction, **f**: function (ie. machine)

$$\mathbf{x} \longrightarrow \boxed{\mathbf{f}} \longrightarrow \mathbf{y}^*$$

- Find **f** from data=learning a machine **f** from (**x**, **y**). ie. machine learning

Regression

- Regression
  - Data **x**, ground-truth **y**.
  - The ground-truth **y** is continuous.
  - It is trained in the supervised way.

Regression vs. Classification

- In classification, **y** label does not have meaning by itself.
- E.g. class 1, 2, 3, … → Class index can be inter-changed.
- In regression, **y** label itself has some meaning.
- E.g. mid-term exam score, weight/height, …

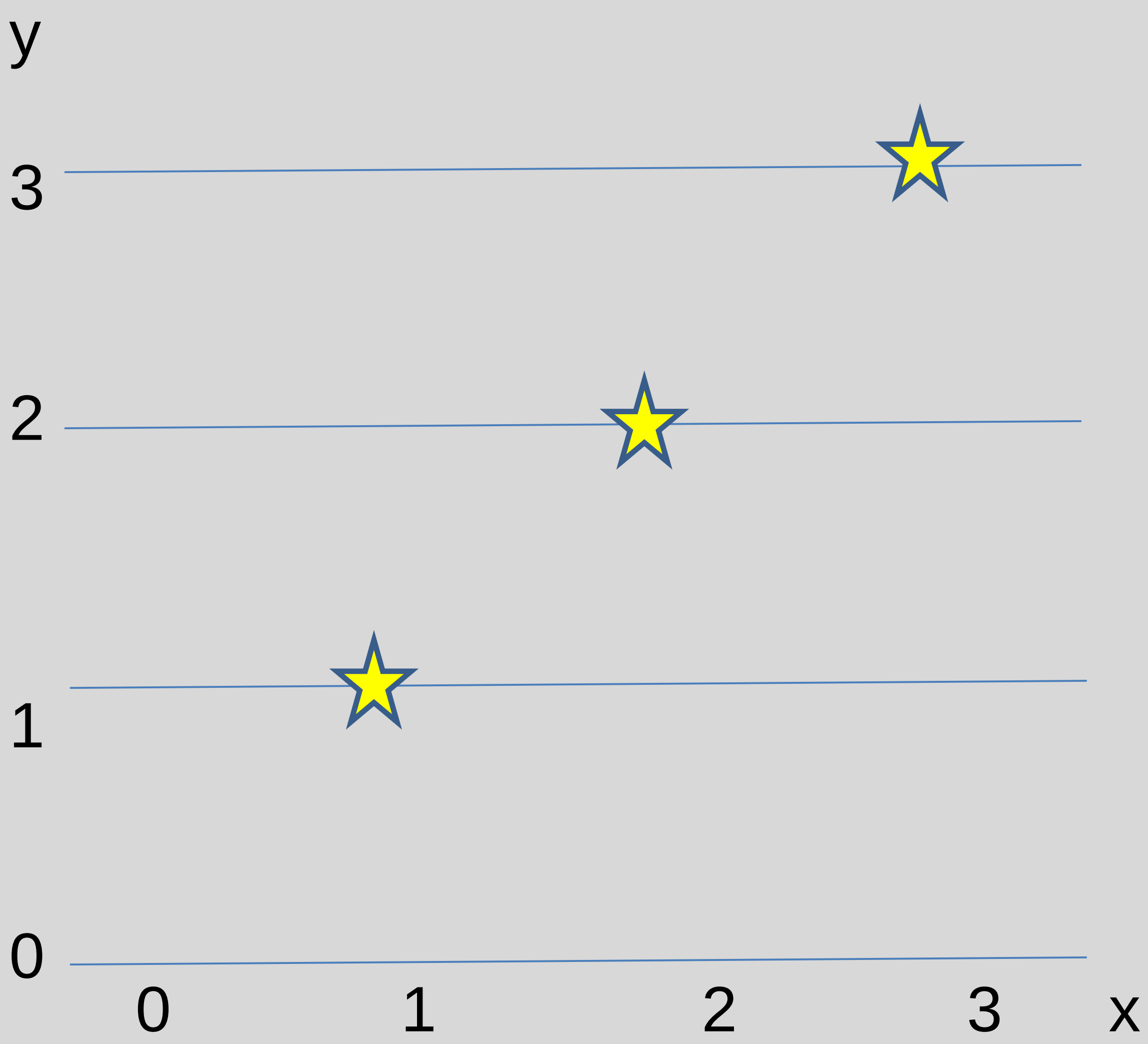
Linear regression

x (attendance score)	y (mid-term score)
10	90
9	70
1	20
4	50
8	60

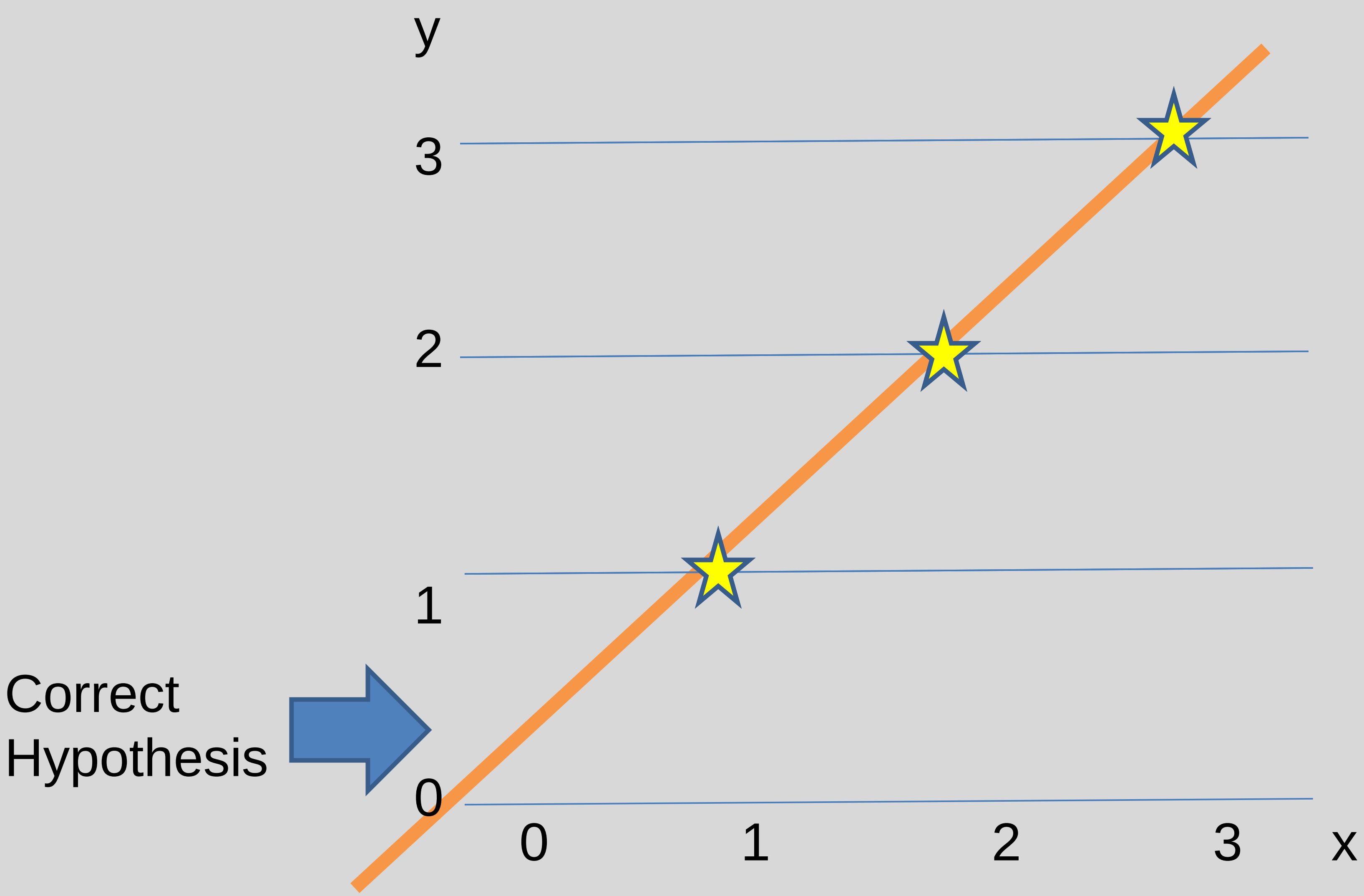
Linear regression

x	y
1	1
2	2
3	3

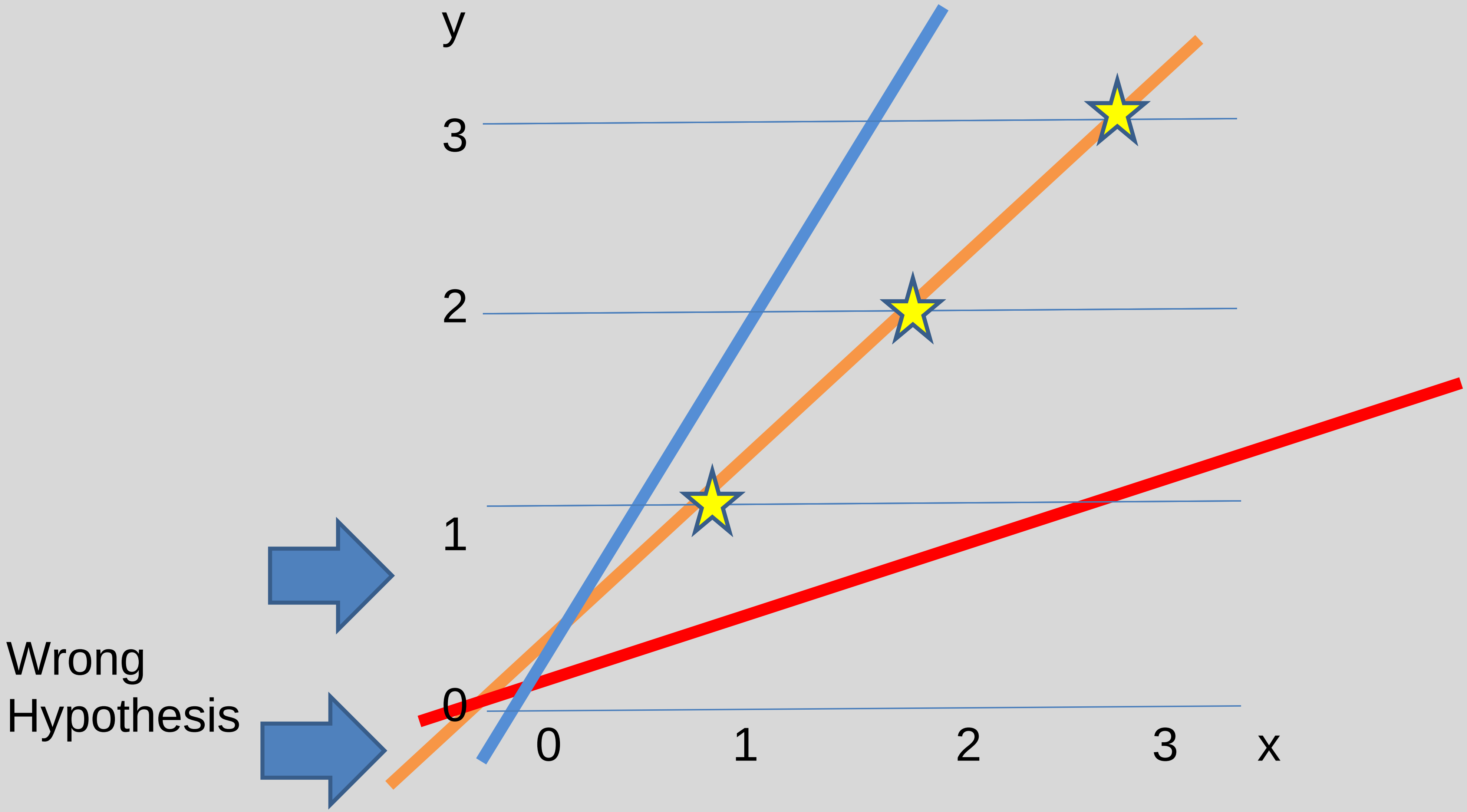
Linear regression



Linear regression



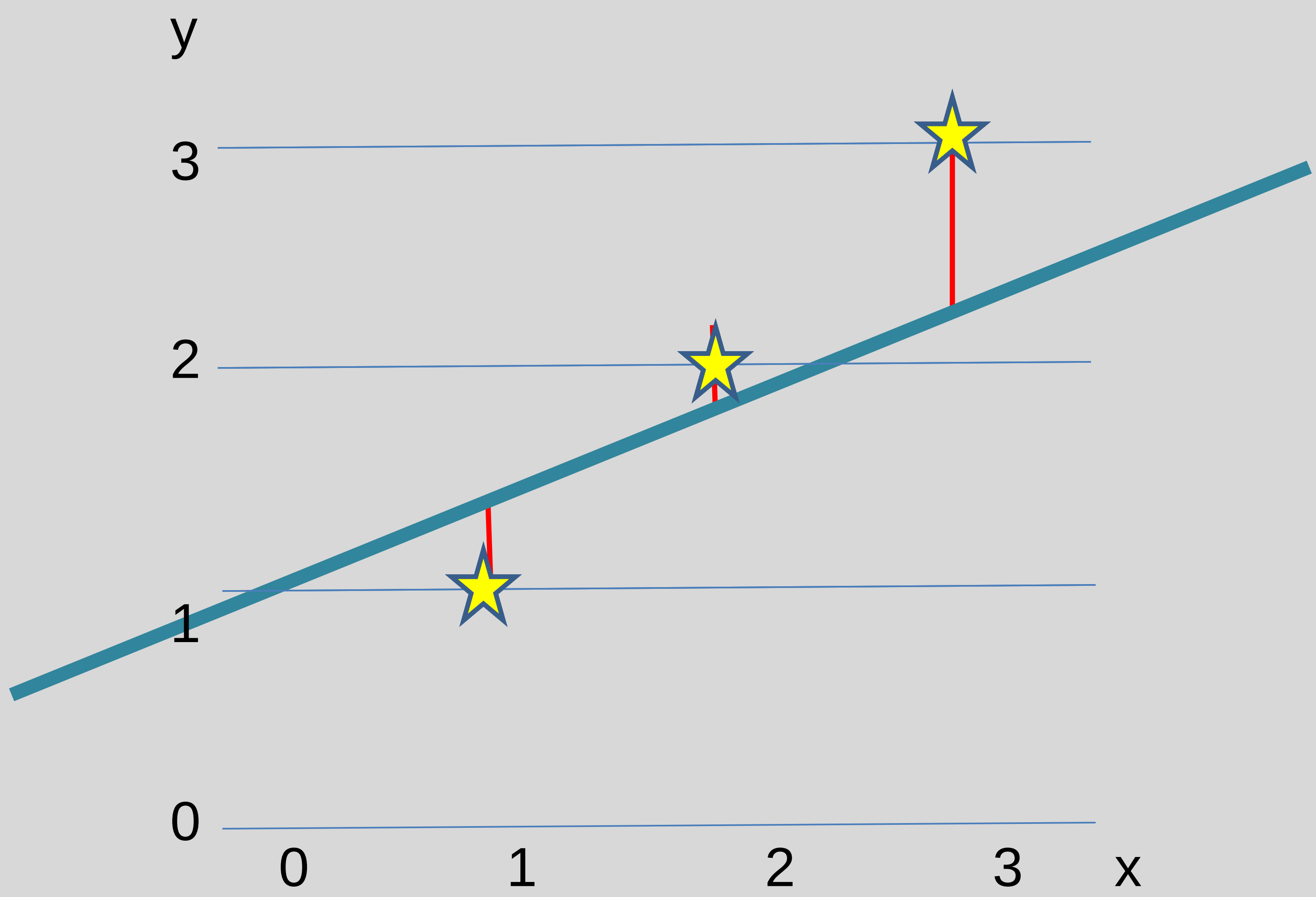
Linear regression



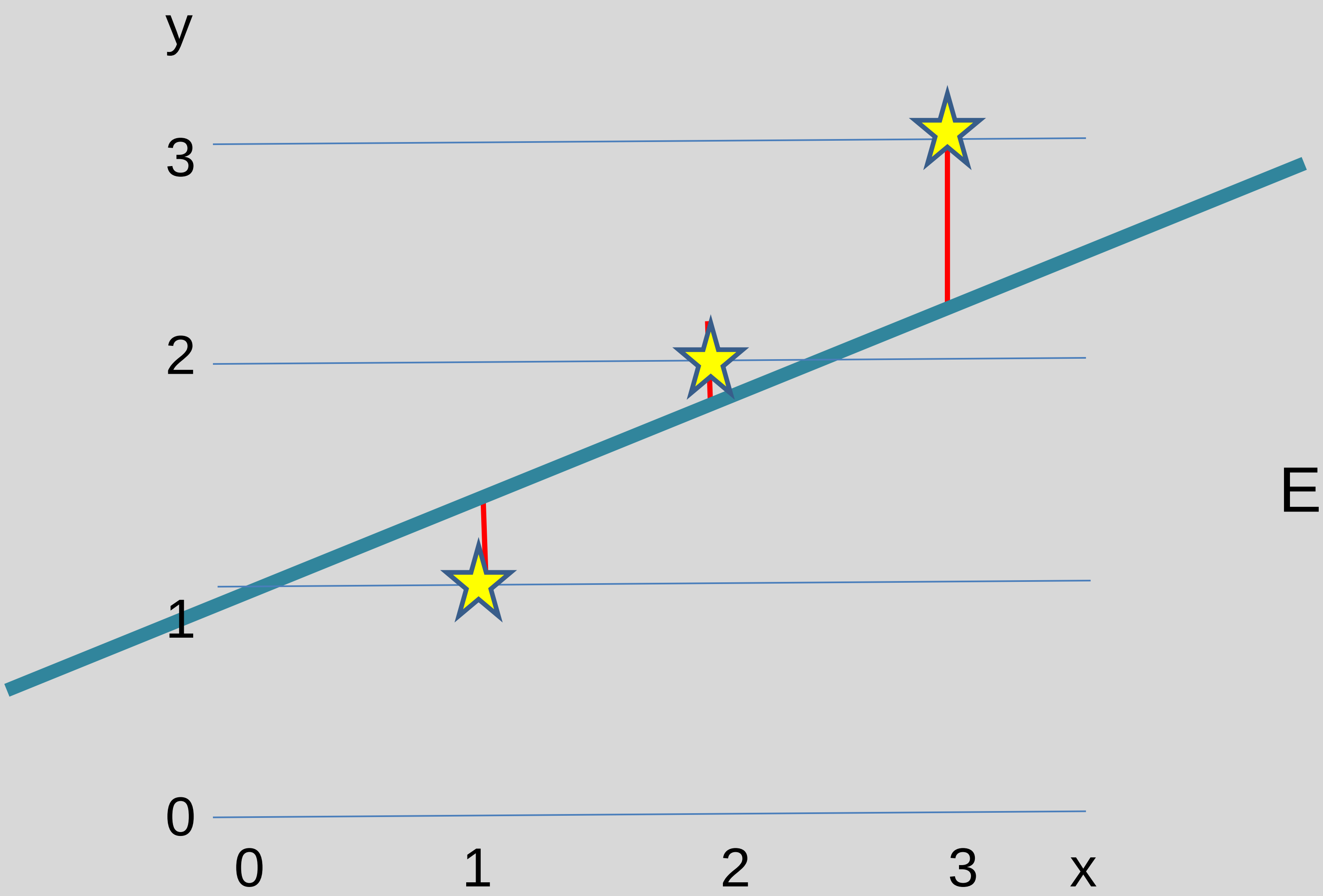
Linear regression

$$\text{Hypothesis}: H(x) = y = Wx + b$$

Linear regression



Linear regression



$$\text{Error} = \frac{1}{3}\sum_{i=1}^{3}(H(x_i) - y_i)^2$$

$$H(x) = Wx + b$$

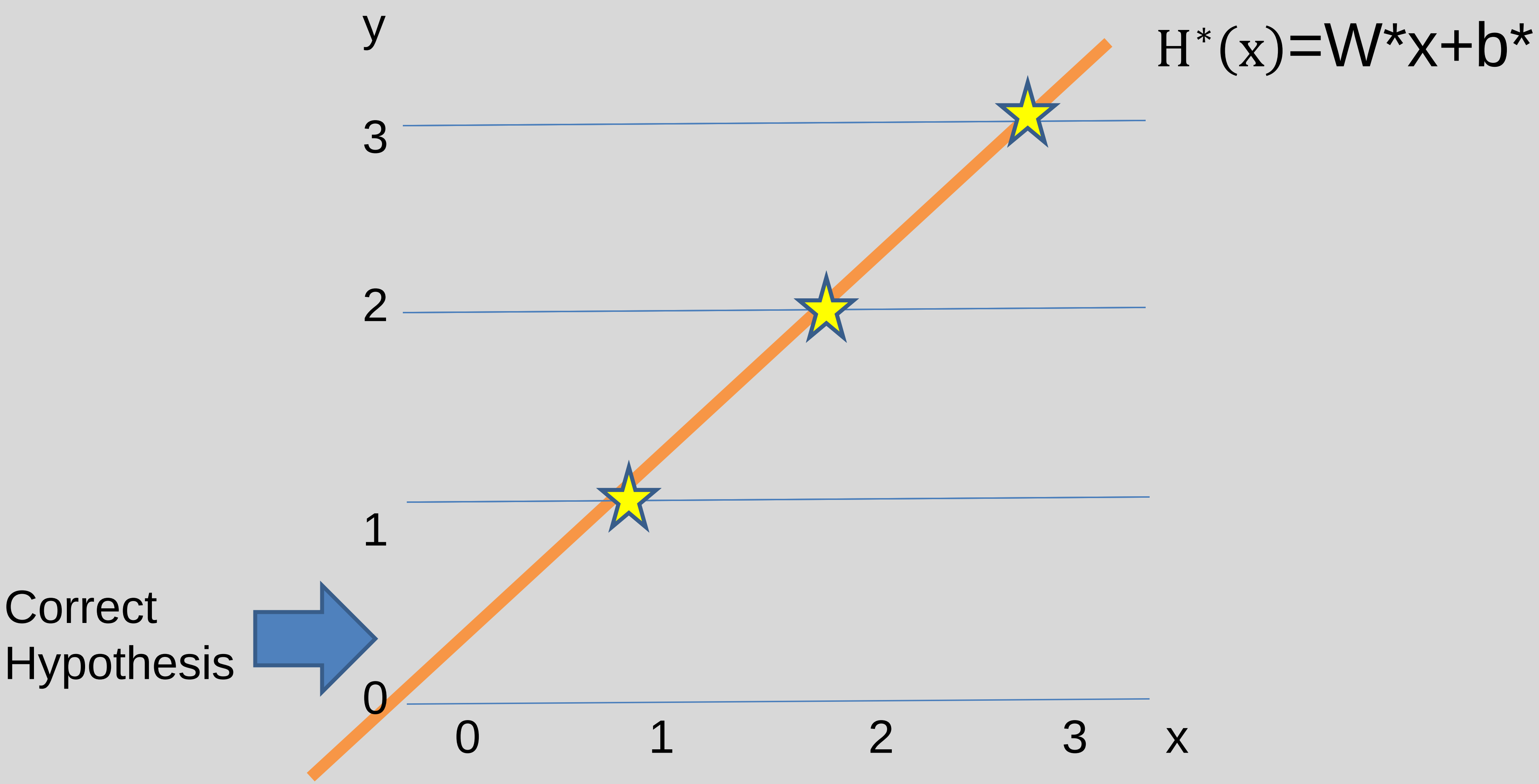
Linear regression

$$\text{Error}(W,b) = \frac{1}{3}\sum_{i=1}^{3}(H(x_i) - y_i)^2,\ H(x) = Wx+b$$

$$W^*, b^* = \text{minimize}_{W,b}\,\text{Error}(W, b)$$

$$\Rightarrow\quad H^*(x) = W^*x + b^*$$

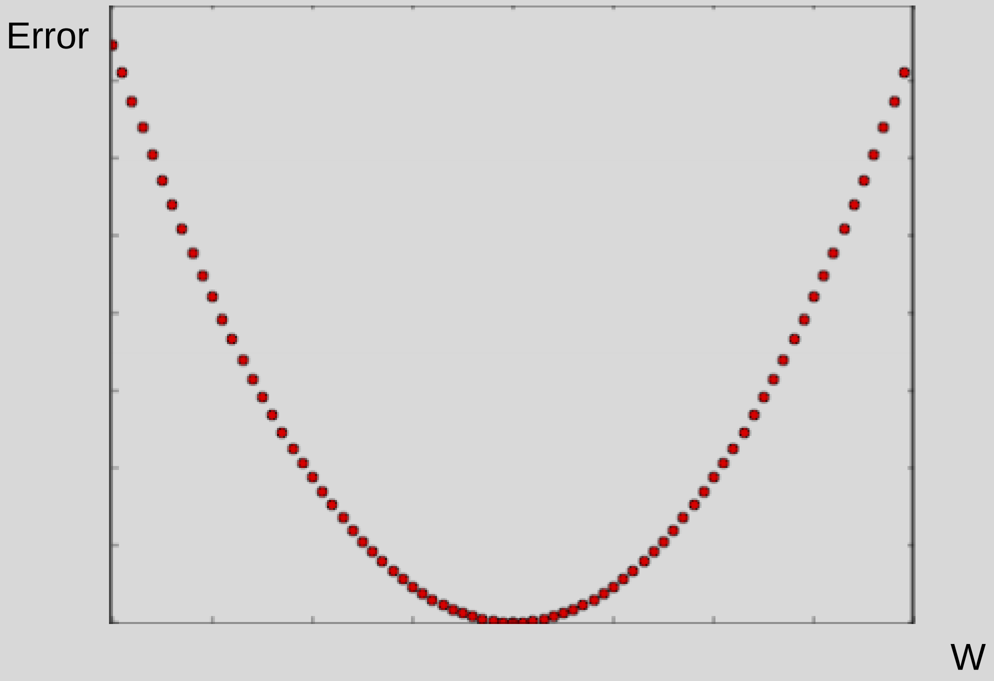
Linear regression



Linear regression

How to optimize it?

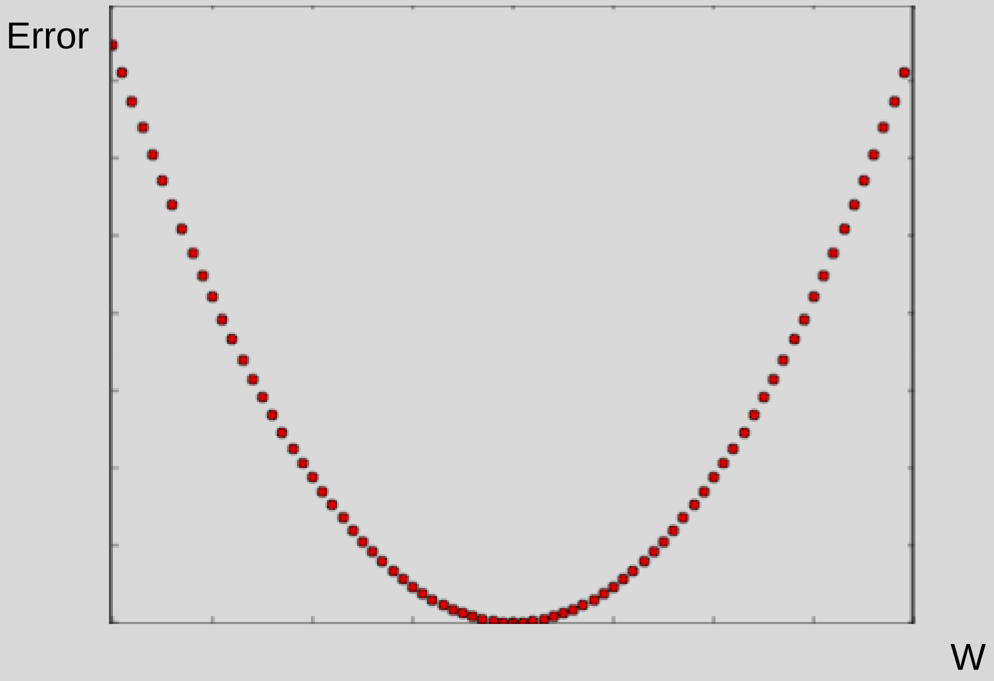
$$\text{Error}(W,b) = \frac{1}{3}\sum_{i=1}^{3}(Wx_i + b - y_i)^2,$$



Linear regression

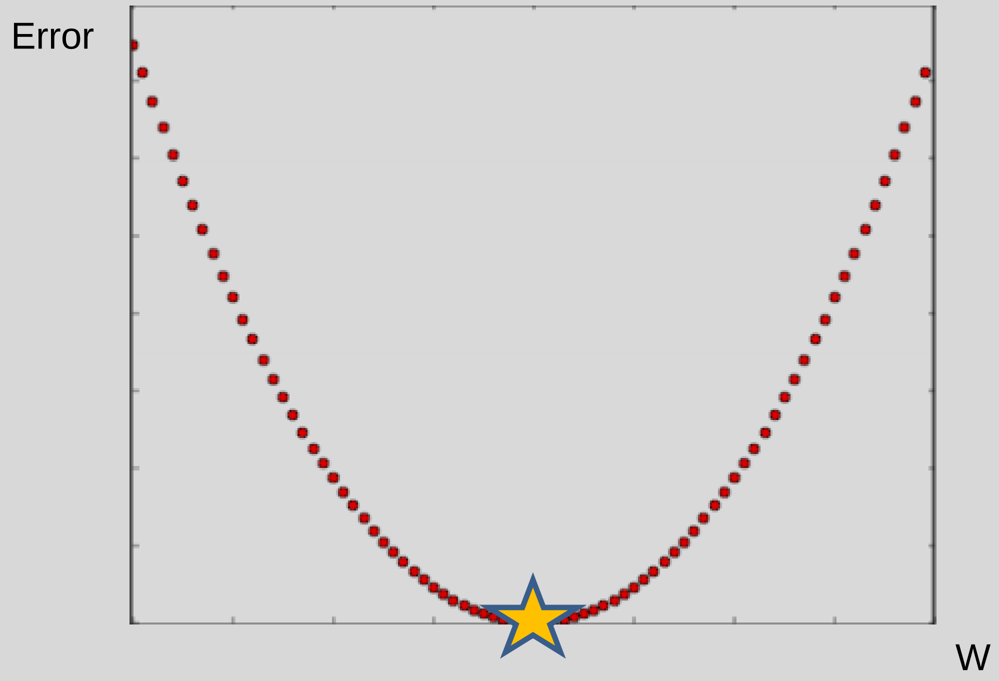
How to optimize it? → Gradient descent algorithm

$$\text{Error}(W,b) = \frac{1}{3}\sum_{i=1}^{3}(Wx_i + b - y_i)^2,$$



Convex function

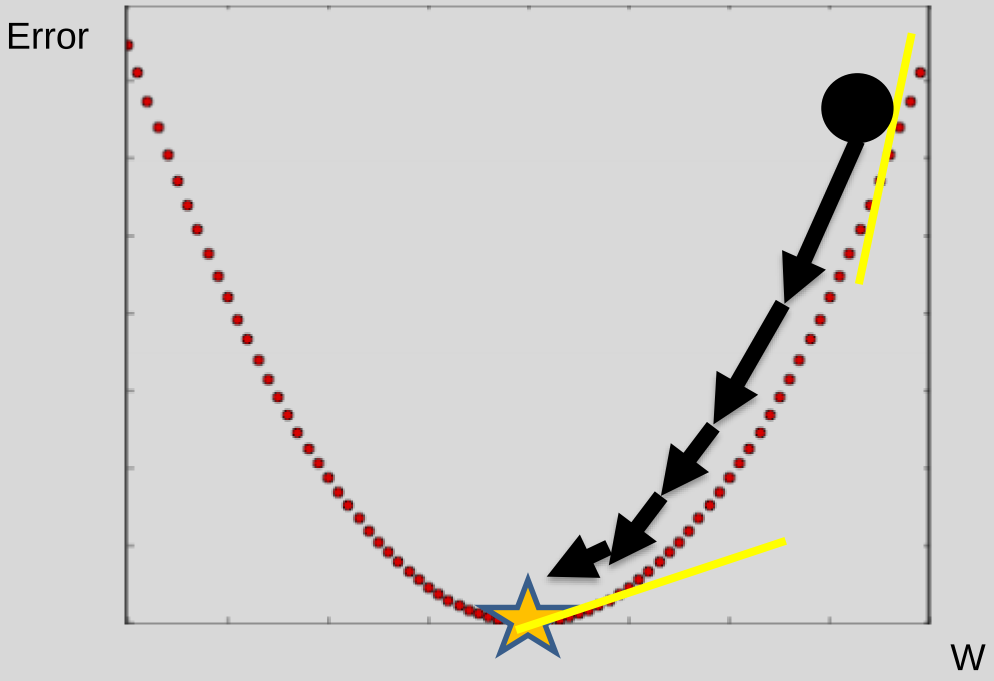
For the convex function, we manually can find the global minima by differentiating it.



Local minima = Global minima

Convex function

We can also iteratively reach the global minima.



Local minima = Global minima

Gradient descent

$$\text{Error(W, b)} = \frac{1}{3}\sum_{\mathrm{i}=1}^{3}(\mathrm{W}\mathrm{x}_\mathrm{i} + \mathrm{b} - \mathrm{y}_\mathrm{i})^2,$$

$$W^{(t+1)} = W^{(t)} - \epsilon\frac{\partial}{\partial W}\text{Error(W, b)}$$

$$b^{(t+1)} = b^{(t)} - \epsilon\frac{\partial}{\partial b}\text{Error(W, b)}$$

ϵ : Learning rate (small value e.g. 0.001)

Multi-variate linear regression

x1 (attendance score)	x2 (quiz score)	x3 (assignment 1 score)	y (mid-term score)
10	8	5	90
9	7	7	70
1	4	3	20
4	6	6	50
8	3	7	60

Multi-variate linear regression

$$\mathrm{H}(\boldsymbol{x}) = \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \mathrm{b}$$

$$H(x_1, x_2, x_3) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

W and **x** become vectors.

PyTorch library

https://numpy.org/

NumPy

Python library for multi-dimensional array.

https://pytorch.org/

PyTorch

Python library for multi-dimensional array.

+

Simple gradient computation. (loss.backward())
Simple GPU usage.

Implementing linear regression using Numpy

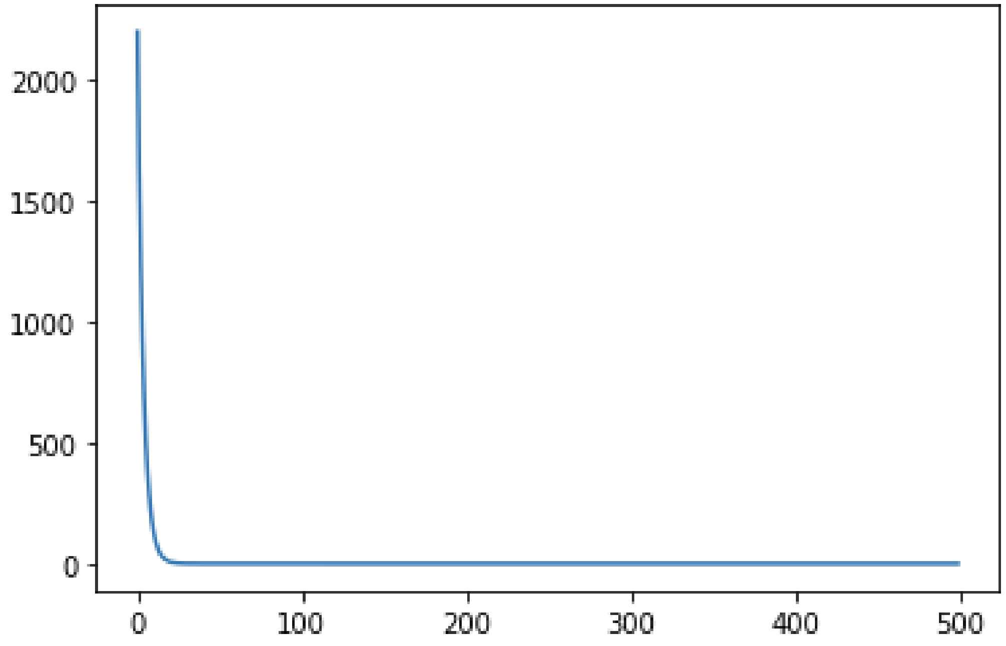
```
import numpy as np

w_true = np.array([1, 2, 3])
b_true = 5

w = np.random.rand(3, 1).squeeze(1)
b = np.random.rand(1)

X = np.random.rand(100, 3)
y = np.matmul(X, w_true) + b_true

gamma = 0.1
losses = []

for i in range(100):

  errors = y - (np.dot(X, w) + b)
  dEdw = np.dot(X.T, errors)
  dEdb = errors.sum()
  loss = (errors**2).sum()/2.0

  w += gamma * dEdw
  b += gamma * dEdb

  losses.append(loss)

print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)

from matplotlib import pyplot as plt
plt.plot(losses)
```

Numpy does not support backward()!

‘’By ourselves, we have to find the closed form for the gradient.’’

Implementing linear regression using Numpy

```
obtained w:  [1. 2. 3.] true w: [1 2 3]
obtained b:  [5.] true b: 5
[<matplotlib.lines.Line2D at 0x7f5b25169950>]
```



Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Learning rate and the variable we will accumulate our loss.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Learning rate and the variable we will accumulate our loss.

Loop until 100 iteration to change (W, b) solution using gradient descent.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Learning rate and the variable we will accumulate our loss.

Loop until 100 iteration to change (W, b) solution using gradient descent.

Regression loss.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

Ground-truth linear regression parameter (W, b) we decided.

We will find the solution (W, b) in this random initialized variable.

Data (X, y) are generated using ground-truth (W, b).

Learning rate and the variable we will accumulate our loss.

Loop until 100 iteration to change (W, b) solution using gradient descent.

Regression loss.

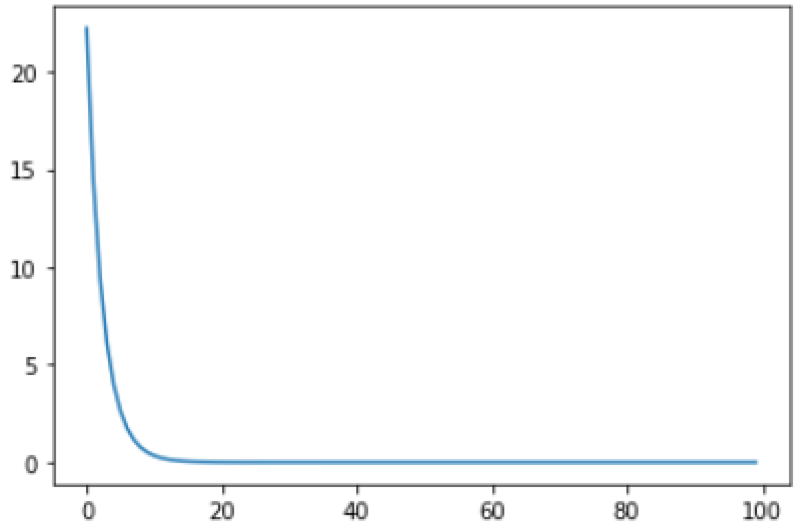
Gradient descent formula.

Implementing linear regression using PyTorch

```python
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)

from matplotlib import pyplot as plt
plt.plot(losses)
```

Compare obtained (W, b) with their ground-truth.
It's same!

```
obtained w:  tensor([1.0000, 2.0000, 3.0000], requires_grad=True) true w: tensor([1., 2., 3.])
obtained b:  tensor([5.0000], requires_grad=True) true b: 5
[<matplotlib.lines.Line2D at 0x7f5db4a31f28>]
```



Implementing linear regression using PyTorch

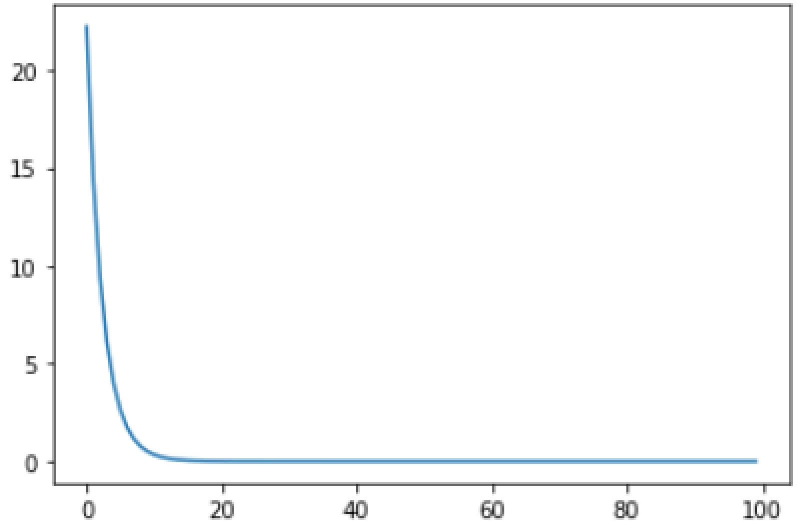
```python
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
```

Compare obtained (W, b) with their ground-truth.
It's same!

```python
from matplotlib import pyplot as plt
plt.plot(losses)
```

The loss plotted for each iteration. It is gradually reduced!

```
obtained w:  tensor([1.0000, 2.0000, 3.0000], requires_grad=True) true w: tensor([1., 2., 3.])
obtained b:  tensor([5.0000], requires_grad=True) true b: 5
[<matplotlib.lines.Line2D at 0x7f5db4a31f28>]
```



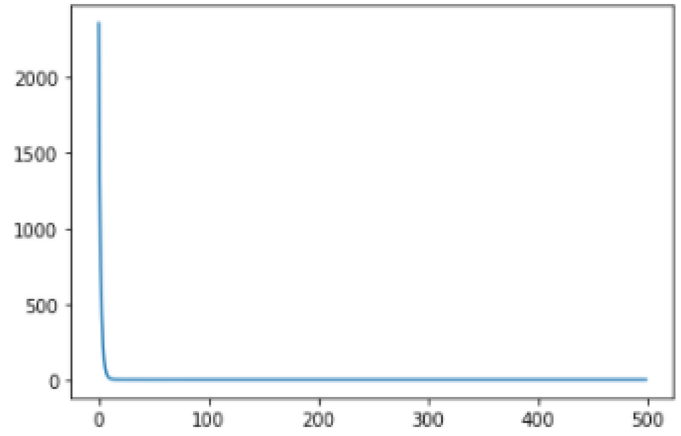
Implementing linear regression using PyTorch

```python
import numpy as np

w_true = np.array([1, 2, 3])
b_true = 5

w = np.random.rand(3, 1).squeeze(1)
b = np.random.rand(1)

X = np.random.rand(100, 3)
y = np.dot(X, w_true) + b_true

gamma = 0.01
losses = []

for i in range(500):
  errors = y - (np.dot(X, w) + b)
  dEdw = np.dot(X.T, errors)
  dEdb = errors.sum()
  loss = (errors**2).sum() / 2.0

  w += gamma * dEdw
  b += gamma * dEdb

  losses.append(loss)
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
from matplotlib import pyplot as plt
plt.plot(losses)
```

```python
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

w = torch.randn(3, requires_grad=True)
b = torch.randn(1, requires_grad=True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true

gamma = 0.01
losses = []

for i in range(500):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y-y_pred)**2)
  loss.backward()

  w.data -= gamma * w.grad.data
  b.data -= gamma * b.grad.data

  losses.append(loss.item())

print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
from matplotlib import pyplot as plt
plt.plot(losses)
```

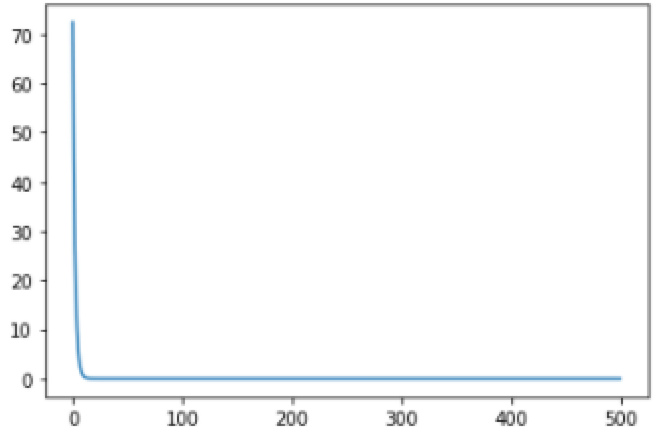
Implementing linear regression using PyTorch

```
obtained w:  [1. 2. 3.] true w: [1 2 3]
obtained b:  [5.] true b: 5
[<matplotlib.lines.Line2D at 0x7fb776c7b650>]
```



[Numpy result]

```
obtained w:  tensor([1.0000, 2.0000, 3.0000], requires_grad=True) true w: tensor([1., 2., 3.])
obtained b:  tensor([5.0000], requires_grad=True) true b: 5
[<matplotlib.lines.Line2D at 0x7f271f38b190>]
```

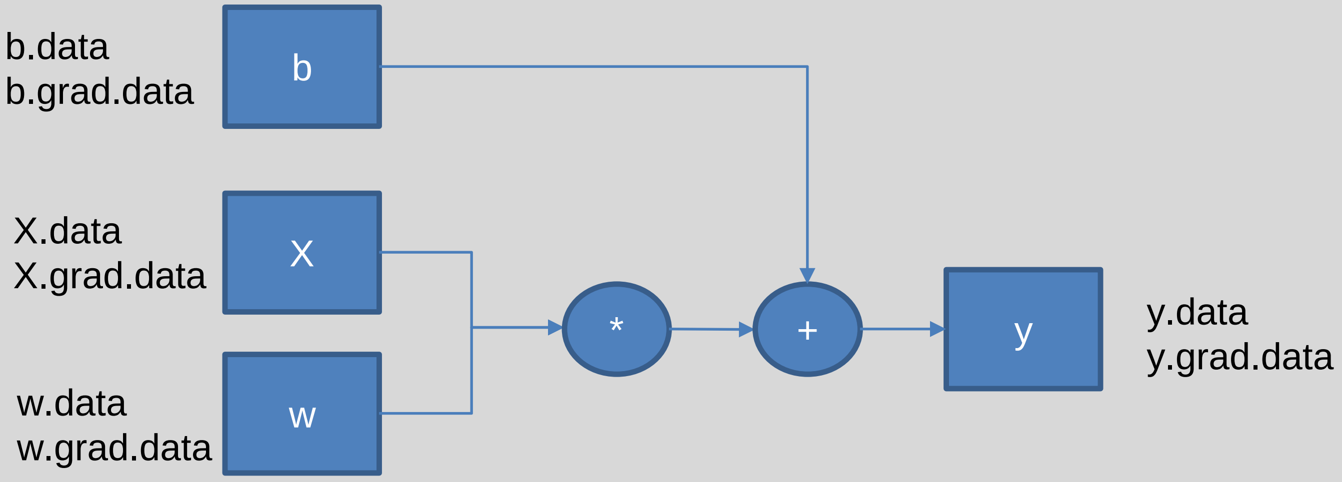


[PyTorch result]

Result is same; however we don’t need to explicitly specify the differentiation form.

Implementing linear regression using PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

w = torch.randn(3, requires_grad=True)
b = torch.randn(1, requires_grad=True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true

gamma = 0.01
losses = []

for i in range(500):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y-y_pred)**2)
  loss.backward()

  w.data -= gamma * w.grad.data
  b.data -= gamma * b.grad.data

  losses.append(loss.item())

print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)
from matplotlib import pyplot as plt
plt.plot(losses)
```



After calling loss.backward(), .grad values are calculated.

PyTorch

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

w = torch.randn(3, requires_grad = True)
b = torch.randn(1, requires_grad = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = torch.mv(X, w) + b

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

PyTorch

```python
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

for i in range(100):
  w.grad = None
  b.grad = None

  y_pred = net(X)

  loss = torch.mean((y- y_pred)**2)
  loss.backward()

  w.data = w.data - gamma * w.grad.data
  b.data = b.data - gamma * b.grad.data

  losses.append(loss.item())
```

```python
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

PyTorch

```python
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = torch.mean((y- y_pred.squeeze(1))**2)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

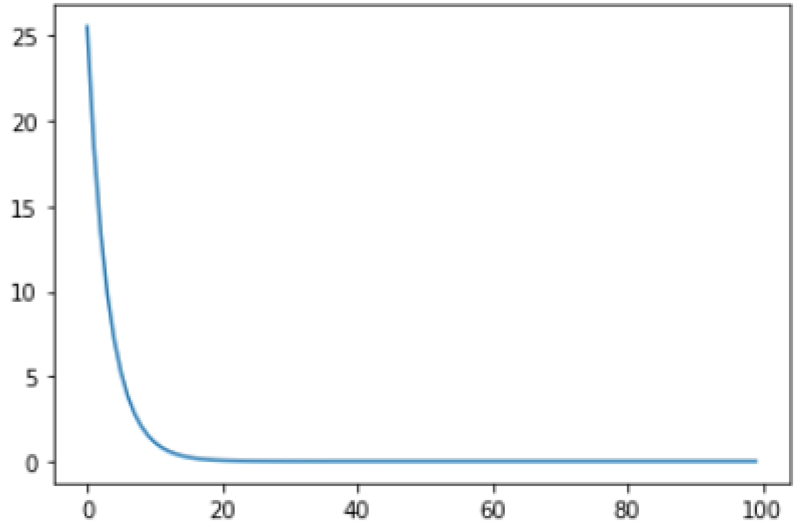
```python
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)

X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

optimizer = torch.optim.SGD(net.parameters(), lr=gamma)
loss_fn = torch.nn.MSELoss()

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = loss_fn(y_pred.squeeze(1),y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

PyTorch

```
print('obtained w: ', w, 'true w:', w_true)
print('obtained b: ', b, 'true b:', b_true)

from matplotlib import pyplot as plt
plt.plot(losses)
```

```
print(list(net.parameters()))

from matplotlib import pyplot as plt
plt.plot(losses)
```

```
[Parameter containing:
tensor([[1.0000, 2.0000, 3.0000]], requires_grad=True), Parameter containing:
tensor([5.0000], requires_grad=True)]
[<matplotlib.lines.Line2D at 0x7f32a3f21710>]
```



PyTorch

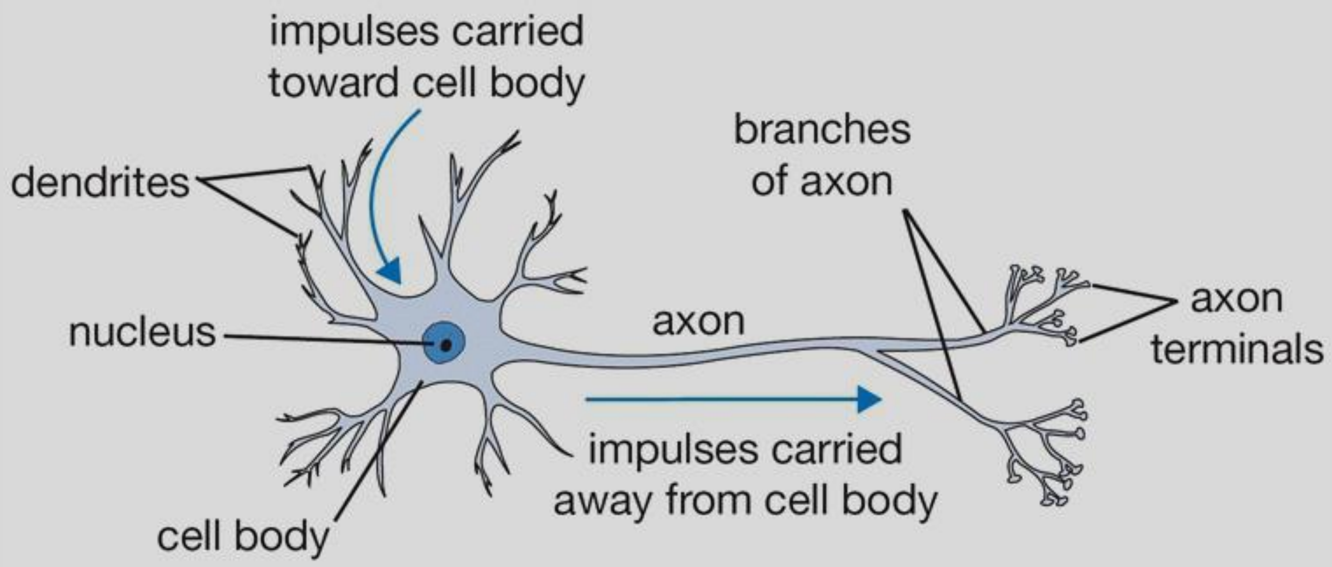
```
import torch

w_true = torch.Tensor([1, 2, 3])
b_true = 5
X = torch.randn(100, 3)
y = torch.mv(X, w_true) + b_true
gamma = 0.1
losses = []

net = torch.nn.Linear(in_features = 3, out_features = 1, bias = True)
optimizer = torch.optim.SGD(net.parameters(), lr=gamma)
loss_fn = torch.nn.MSELoss()

for i in range(100):
  optimizer.zero_grad()

  y_pred = net(X)

  loss = loss_fn(y_pred.squeeze(1),y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())

print(list(net.parameters()))

from matplotlib import pyplot as plt
plt.plot(losses)
```

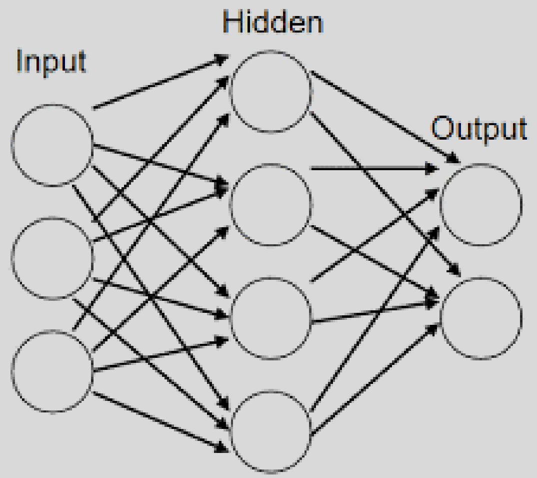
Data Preparation.

Network structure
Optimizer
Loss

Iterate for updating network parameters.
Initialize. gradients.
Forward pass.
Calculate loss.
Backward pass (Calc. gradients.).
Update network parameter.

Multi-layer perceptron





$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1x+b_1))+b_2) +b_3))$$

Multi-layer perceptron

Why we need σ ?

$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1x+b_1))+b_2)) +b_3))$$

$$\begin{aligned} y &= w_3(w_2(w_1x+b_1)+b_2) +b_3 \\ &= (w_3w_2w_1)x +(w_3w_2b_1+w_3b_2+b_3) \\ &= wx+b \end{aligned}$$

➡ Huge network converges to a simple linear regression task.

Multi-layer perceptron

$$y = \sigma(w_3(\sigma(w_2(\sigma(w_1x+b_1))+b_2)) + b_3))$$

σ : Sigmoid, Tanh, ReLu and so on…

These functions are also differentiable.

Multi-layer perceptron

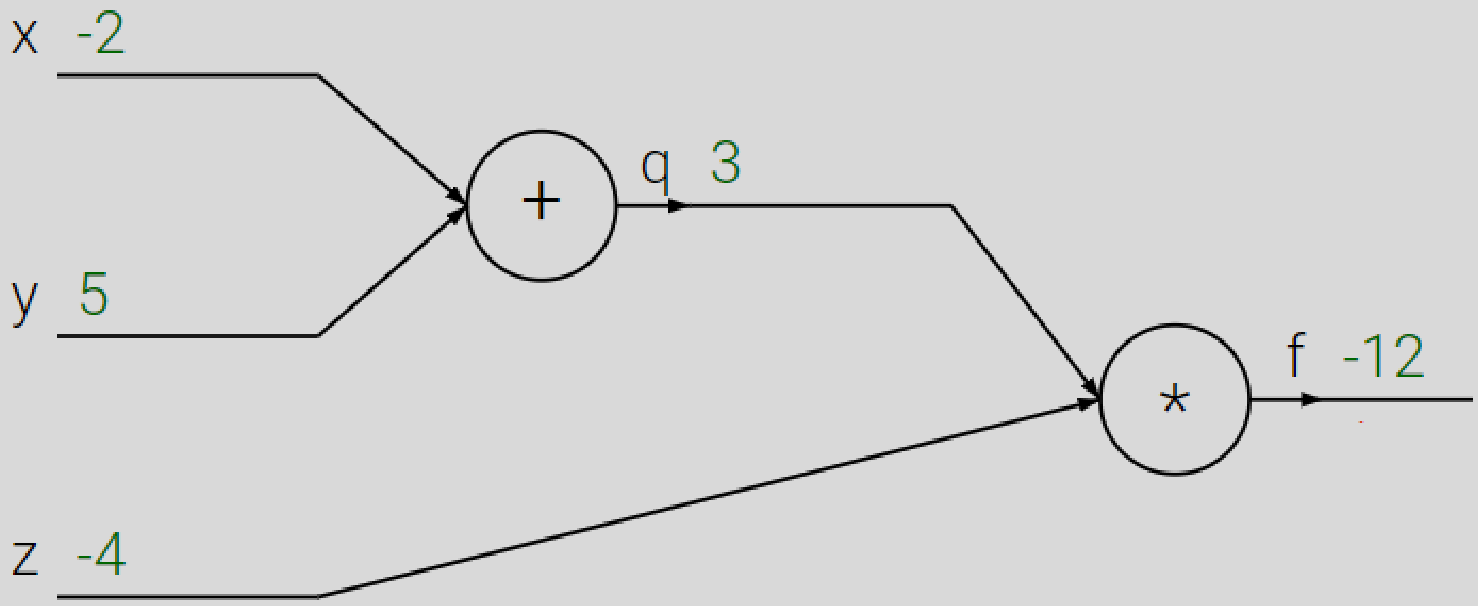
$$\text{Error}(\mathbf{w}, \mathbf{b}) = \frac{1}{\mathrm{N}}\sum_{\mathrm{i}=1}^{\mathrm{N}}(\sigma(\boldsymbol{w}_3(\sigma(\boldsymbol{w}_2(\sigma(\boldsymbol{w}_1 \mathrm{x}+\mathrm{b}_1)) + \mathrm{b}_2)) + \mathrm{b}_3)) - \mathrm{y}_\mathrm{i})^2,$$

$$\boldsymbol{w}_\mathrm{i}^{(t+1)} = \boldsymbol{w}_\mathrm{i}^{(t)} - \epsilon \frac{\partial}{\partial \boldsymbol{w}_\mathrm{i}} \text{Error}(\mathbf{w}, \mathbf{b})$$

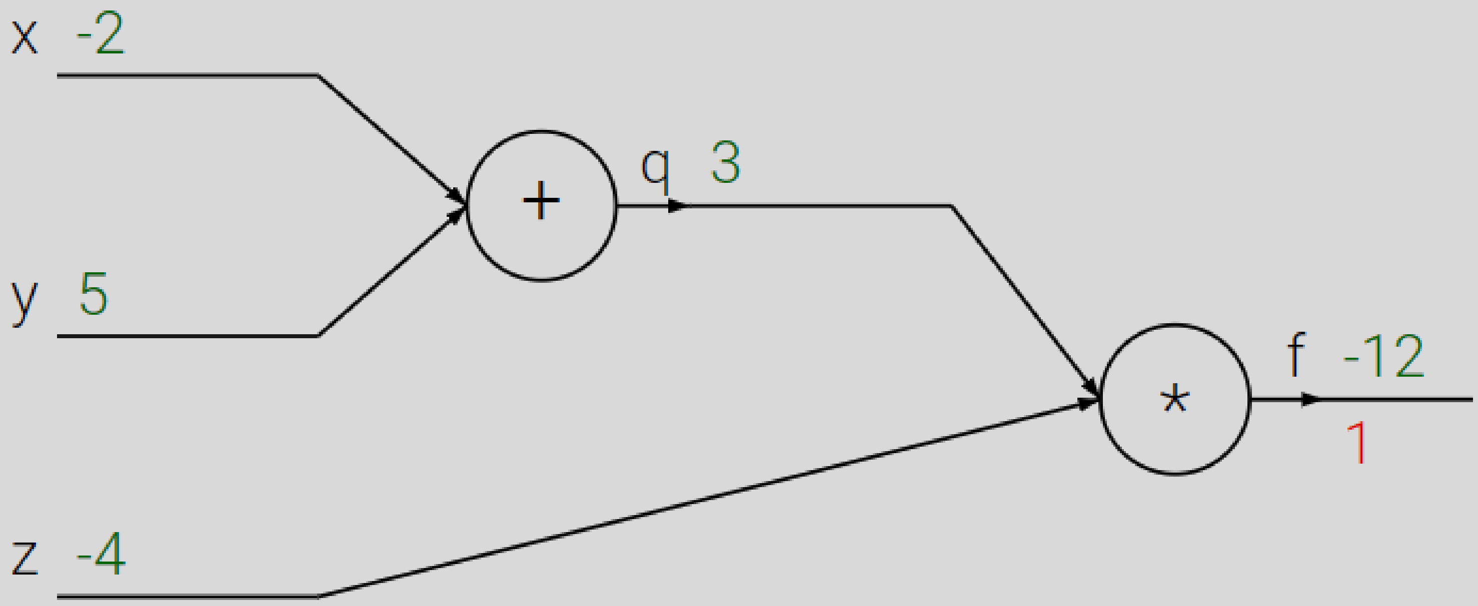
$$\mathrm{b}_\mathrm{i}^{(t+1)} = \mathrm{b}_\mathrm{i}^{(t)} - \epsilon \frac{\partial}{\partial \mathrm{b}_\mathrm{i}} \text{Error}(\mathbf{w}, \mathbf{b})$$

ϵ : Learning rate (small value e.g. 0.001)

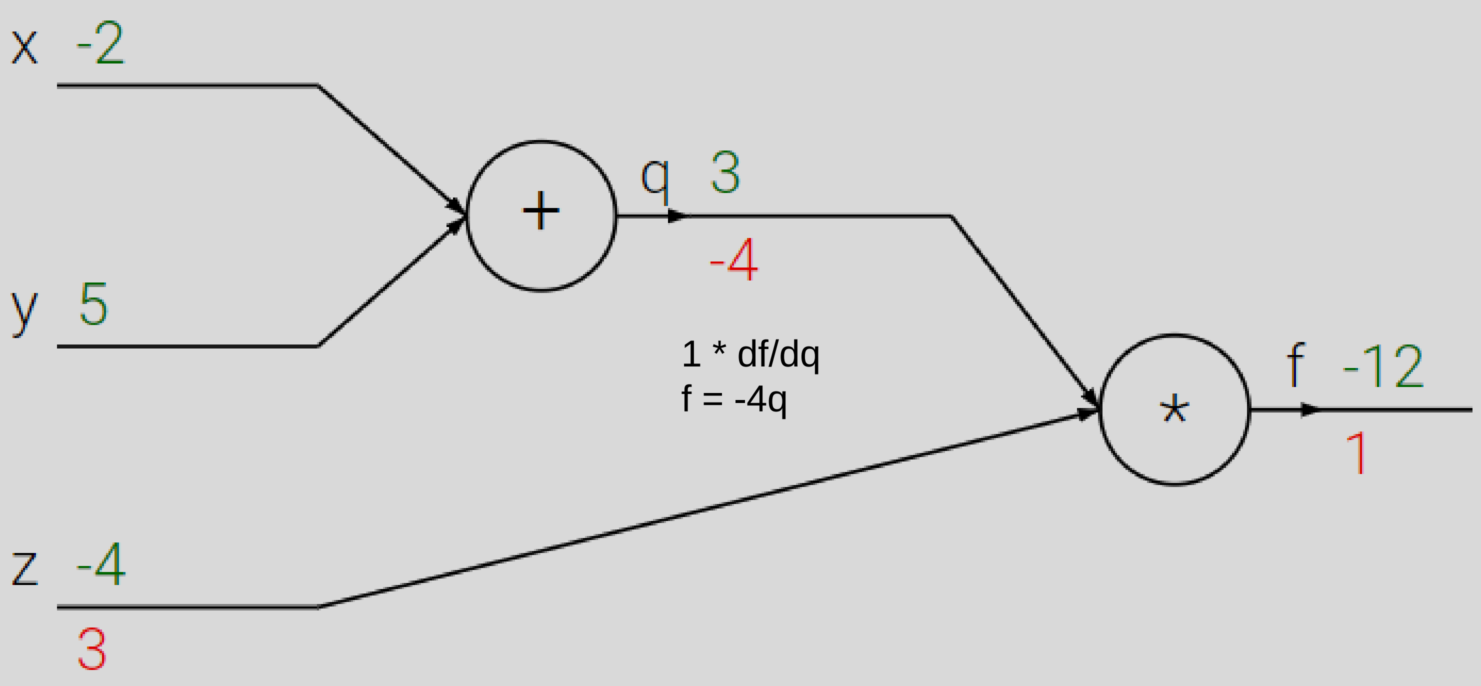
Multi-layer perceptron



Multi-layer perceptron



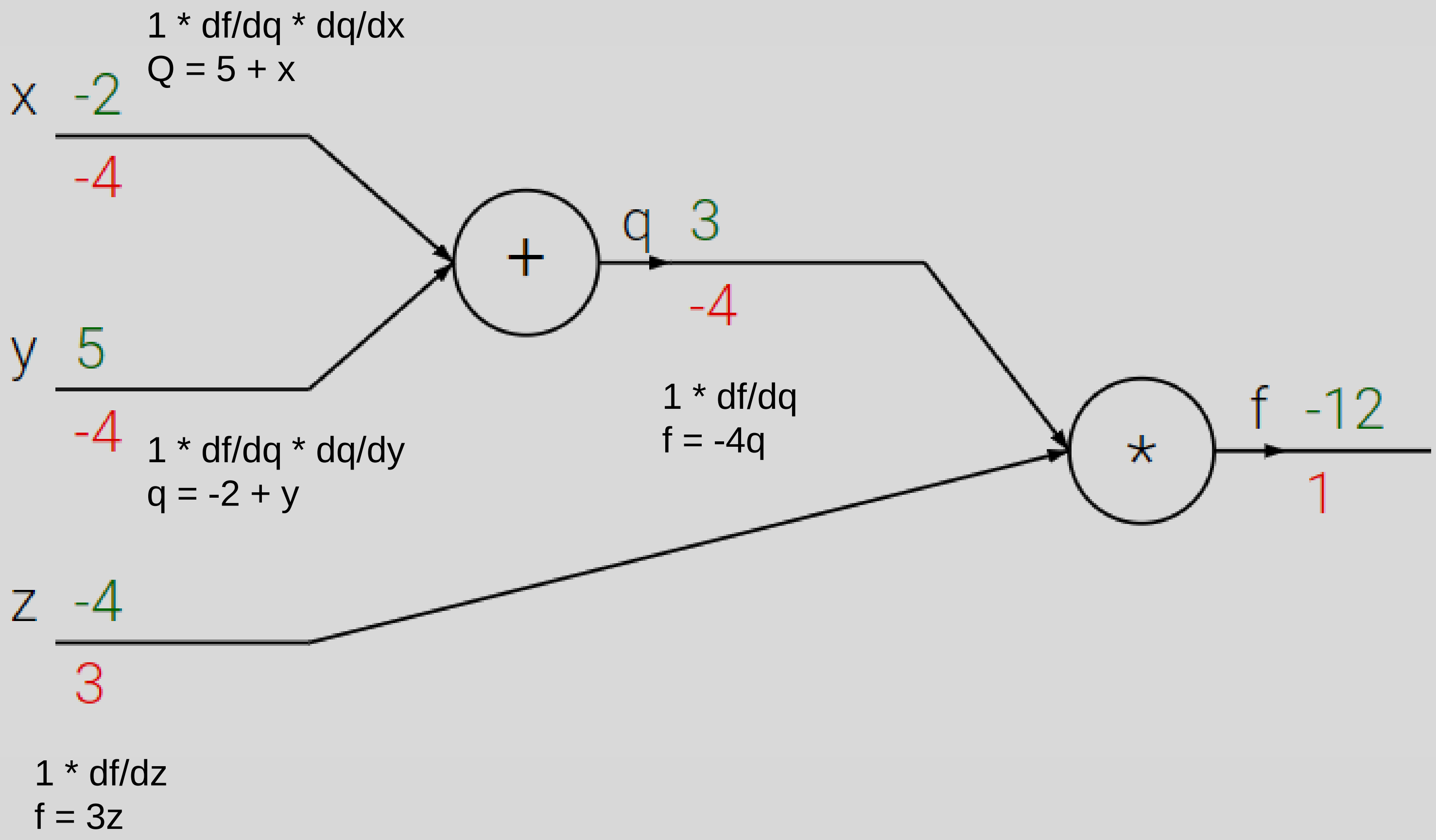
Multi-layer perceptron



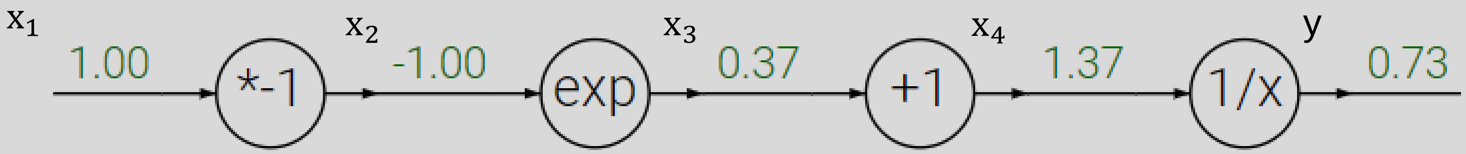
1 * df/dq
f = -4q

1 * df/dz
f = 3z

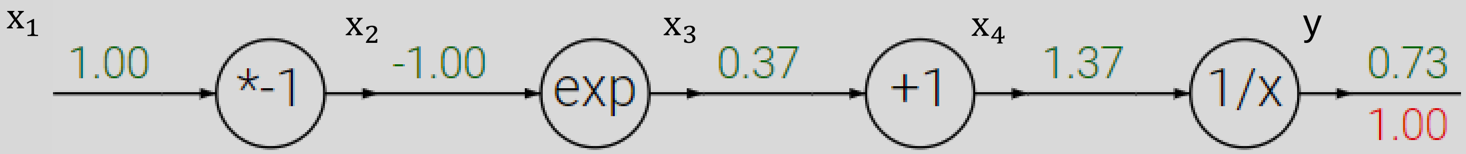
Multi-layer perceptron



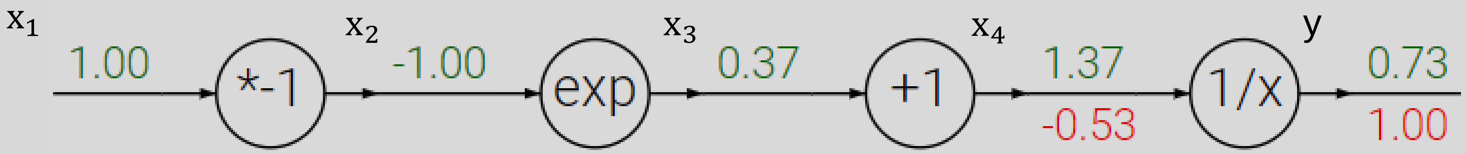
Multi-layer perceptron



Multi-layer perceptron



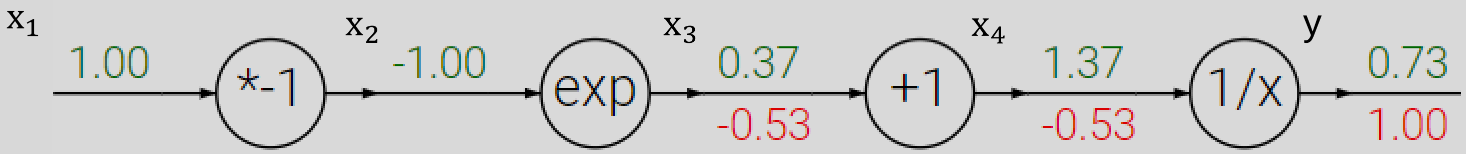
Multi-layer perceptron



$y=1/x_4$

$\frac{dy}{dx_4}=1.00 * \frac{-1}{{x_4}^2}=\frac{-1}{(1.37)^2}=-$
0.53

Multi-layer perceptron

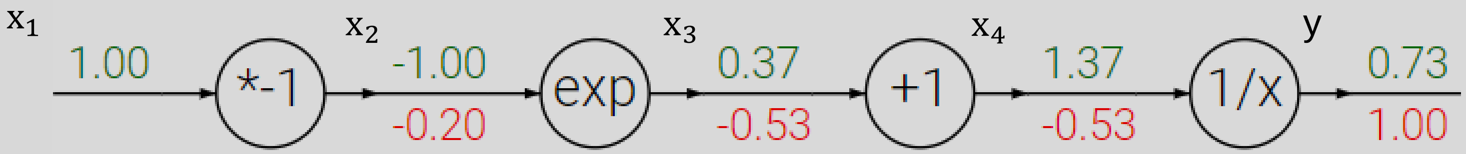


$x_4 = x_3$

$\frac{dx_4}{dx_3} = 1.00$

$\frac{dy}{dx_3} = \frac{dy}{dx_4} * \frac{dx_4}{dx_3} = -0.53$

Multi-layer perceptron

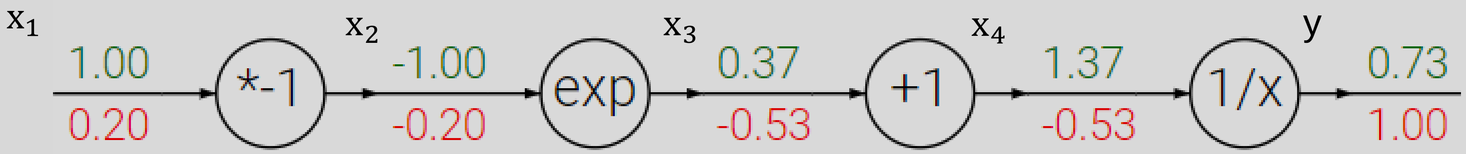


$x_3 = \exp(x_2)$

$\frac{dx_3}{dx_2} = \exp(x_2)$

$\frac{dy}{dx_2} = \frac{dy}{dx_3} * \frac{dx_3}{dx_2}$=-0.53*0.3678…=-0.1949…

Multi-layer perceptron



$x_2 = -x_1$

$\frac{dx_2}{dx_1} = -1$

$\frac{dy}{dx_1} = \frac{dy}{dx_2} * \frac{dx_2}{dx_1} = -0.2 * -0.1 = 0.2$

Implementing Multi-layer perceptron using PyTorch

```
import torch

num_data = 1000
num_epoch = 10000
x = torch.randn(num_data, 1)
y = (x**2) + 3

net = torch.nn.Sequential(
    torch.nn.Linear(1,6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 1),
)

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Make data.

Multi-layer perceptron structure.

Define loss function and optimizer.

Optimize network through iterations.

Implementing Multi-layer perceptron using PyTorch

```
import torch

num_data = 1000
num_epoch = 10000
x = torch.randn(num_data, 1)
y = (x**2) + 3

net = torch.nn.Sequential(
    torch.nn.Linear(1,6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 1),
)

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Make data.

Multi-layer perceptron structure.

Define loss function and optimizer.

Optimize network through iterations.

Implementing Multi-layer perceptron using PyTorch

```
import torch

num_data = 1000
num_epoch = 10000
x = torch.randn(num_data, 1)
y = (x**2) + 3

net = torch.nn.Sequential(
    torch.nn.Linear(1,6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 1),
)

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Make data.

Multi-layer perceptron structure.

Define loss function and optimizer.

Optimize network through iterations.

Implementing Multi-layer perceptron using PyTorch

```
import torch

num_data = 1000
num_epoch = 10000
x = torch.randn(num_data, 1)
y = (x**2) + 3

net = torch.nn.Sequential(
    torch.nn.Linear(1,6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 6),
    torch.nn.ReLU(),
    torch.nn.Linear(6, 1),
)

loss_func = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):
  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)
  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Make data.

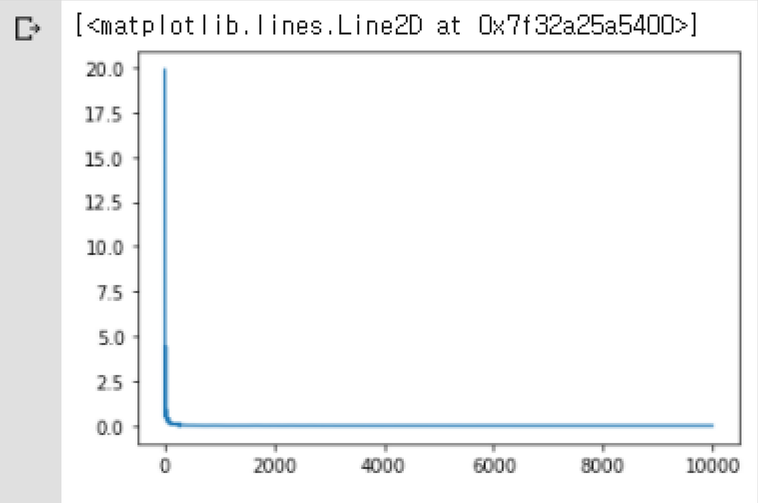
Multi-layer perceptron structure.

Define loss function and optimizer.

Optimize network through iterations.

Implementing Multi-layer perceptron using PyTorch

```
from matplotlib import pyplot as plt
plt.plot(losses)
```



Implementing Multi-layer perceptron using PyTorch

```
x = torch.randn(5, 1)
y = (x**2) + 3
y_pred = net(x)

print(y)
print(y_pred)
```

```
tensor([[5.5024],
        [3.3881],
        [3.0404],
        [3.0684],
        [3.2888]])
tensor([[5.5617],
        [3.3964],
        [3.0547],
        [3.0892],
        [3.2588]], grad_fn=<AddmmBackward>)
```

Thank you!

UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY
2007