Deep Learning

Computational Data Science,
Addis Ababa University

www.aau.edu.et

mesfin.diro@aau.eud.et

+251-912-086156

Data Manipulation

- Generally, there are two important things we need to do with data:
  - acquire them; and
  - process them once they are inside the computer.
- To start, we introduce the n-dimensional array, which is also called the tensor
- No matter which framework you use, its tensor class (`ndarray` `Tensor` in both PyTorch and TensorFlow) is similar to NumPy's `ndarray` with a few killer features
- First, GPU is well-supported to accelerate the computation whereas NumPy only supports CPU computation.
- Second, the tensor class supports automatic differentiation.
- These properties make the tensor class suitable for deep learning.

Data Manipulation

- We will use pytorch in this class and note that though it's called PyTorch, we should import `torch` instead of `pytorch`.

```
import torch
x = torch.arange(12)
x
```

- We can access a tensor's shape (the length along each axis) by inspecting its `shape` property.

```
x.shape
```

- If we just want to know the total number of elements in a tensor:

```
x.numel()
```

Data Manipulation

- To change the shape of a tensor without altering either the number of elements or their values, we can invoke the `reshape` function

```
X = x.reshape(3, 4)
X
```

- We can create a tensor representing a tensor with all elements set to 0 and a shape of (2, 3, 4) as follows:

```
torch.zeros((2, 3, 4))
```

- Similarly, we can create tensors with each element set to 1 as follows:

```
torch.ones((2, 3, 4))
```

Tensor Operations

- The common standard arithmetic operators (+, −, ∗, /, and ∗∗) have all been lifted to element-wise operations:

```
x = torch.tensor([1.0,2.0,4.5])
y = torch.tensor([3,3,3])
y + x, y-x, y*x,x/y , x**y
```

- Many more operations can be applied elementwise, including unary operators like exponentiation.:

```
 torch.exp(x)
```

- We can also concatenate multiple tensors together, stacking them end-to-end to form a larger tensor:

```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

Tensor Broadcasting Mechanism

- Under certain conditions, even when shapes differ, we can still perform element-wise operations by invoking the broadcasting mechanism.

```
x = torch.arange(3).reshape((3, 1))
y = torch.arange(2).reshape((1, 2))
x, y
```

- Since x and y matrices do no match up if we want to add them. However, broadcast the entires of the matrices into a lager matrix a follows:

```
x+y
```

Conversion to the Tensor Format

- Use can convert any numerical values can be converted to the equivalent tensor format

```
from sklearn.datasets import load_iris
iris = load_iris()
X, y = torch.tensor(iris.data), torch.tensor(iris.target)
X, y
```

- Like many other extension packages in the vast ecosystem of Python, pandas and numpy can work together with tensors
- Imputation and deletion can be used to handle missing data.

- A scalar is represented by a tensor with just one element

```
import torch
x = torch.tensor(3.0)
y = torch.tensor(2.0)
x + y, x * y, x / y, x**y
```

- In general vectors are one-dimensional tensors that can have arbitrary lengths

```
import torch
x = torch.arange(3)
x
```

- column vectors to be the default orientation of vectors which can be written as:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \text{ where } x_1, \cdots, x_n \text{ are elements of the vectors}$$

Linear Algebra

- We can create a $m \times n$ matrix by specifying a shape with two components m and n when calling any of our favorite functions for instantiating a tensor
- When a matrix has the same number of rows and columns, its shape becomes a square called a square matrix.

```
import torch
A = torch.arange(25).reshape(5,5)
```

- Sometimes we want to flip the axes and if $B = A^T$, then $b_{ij} = a_{ji}$ for any i and j

```
import torch
A.T
```

Linear Algebra

- We can create a $m \times n$ matrix by specifying a shape with two components m and n when calling any of our favorite functions for instantiating a tensor
- When a matrix has the same number of rows and columns, its shape becomes a square called a square matrix.

```
import torch
A = torch.arange(25).reshape(5,5)
```

- Sometimes we want to flip the axes and if $B = A^T$, then $b_{ij} = a_{ji}$ for any i and j

```
import torch
A.T
```

- Scalars, vectors, matrices, and tensors of an arbitrary number of axes have some nice properties
- When a matrix has the same number of rows and columns, its shape becomes a square called a square matrix.

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

- Specifically, element-wise multiplication of two matrices is called their Hadamard product (math notation $\odot$)
- The Hadamard product of matrices $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}.$$

Linear Algebra

- One useful operation that we can perform with arbitrary tensors is to calculate the sum of their elements..

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
A.shape, A.sum()
```

- However, sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
sum_A =A.sum(axis=1, kepdims=True)
sum_A
```

- If we want to calculate the cumulative sum of elements of A along some axis, say axis =1, the function will not reduce the input tensor along any axis:

```
A.cumsum(axis=0)
```

Linear Algebra

- Some of the most useful operators in linear algebra are norms. Informally, the norm of a vector tells us how big a vector is.
- In linear algebra, a vector norm is a function f that maps a vector to a scalar, satisfying a handful of properties.
- The L_2 norm of $\mathbf{x}$ is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2},$$

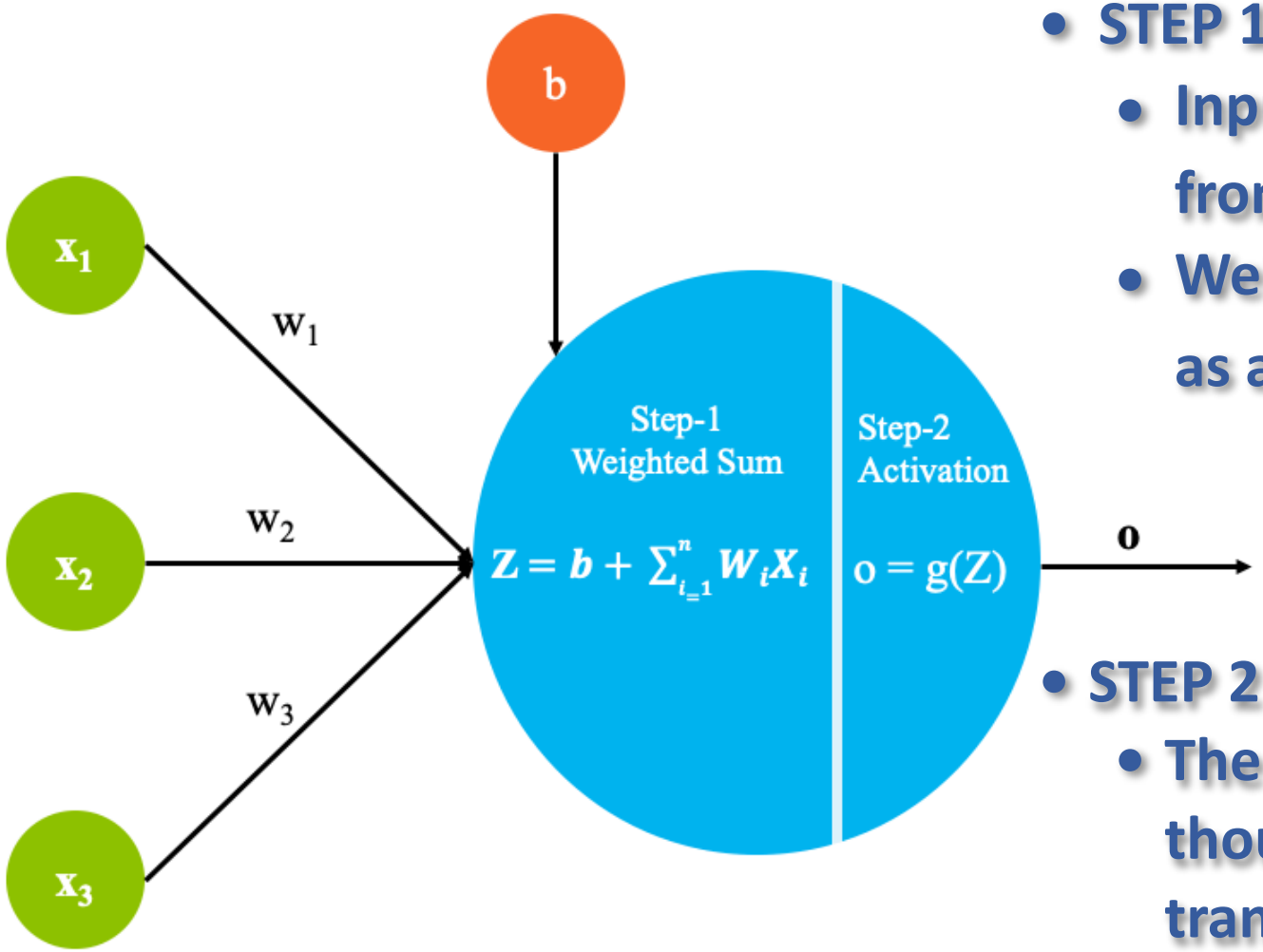
- Hwhere the subscript 2 is often omitted in L_2

```
A = torch.tensor([3.0, -4.0])
torch.norm(A)
```

Deep Learning

- In an effort to create systems that learn similar to how humans learn, the underlying architecture for deep learning was inspired by the structure of a human brain.
- For this reason, quite a few fundamental terminologies within deep learning can be mapped back to neurology.
- The fundamental building blocks of deep learning architecture contains a computational unit that allows modeling of nonlinear functions called *perceptron*.
- Similar to how a “neuron” in a human brain transmits electrical pulses throughout our nervous system, the perceptron receives a list of input signals and transforms them into output signals.

Artificial neural network

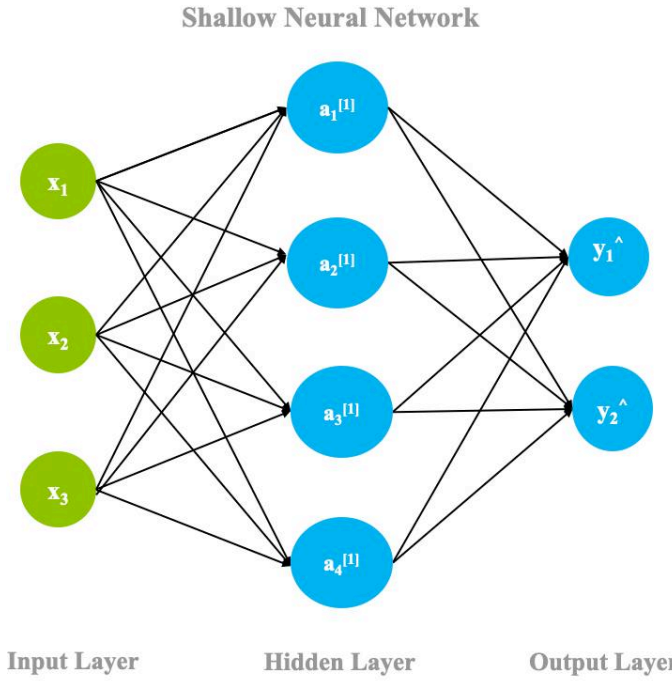


- **STEP 1 calculate the weighted sum:**
  - Input x_1 through x_n represents entry from the dataset
  - Weights w_1 through x_n represented as a matrix W

- **STEP 2 activate function:**
  - The output of step 1 is now passed though a an activation function to transform to a desired non-linear format it is sent to the next layer.

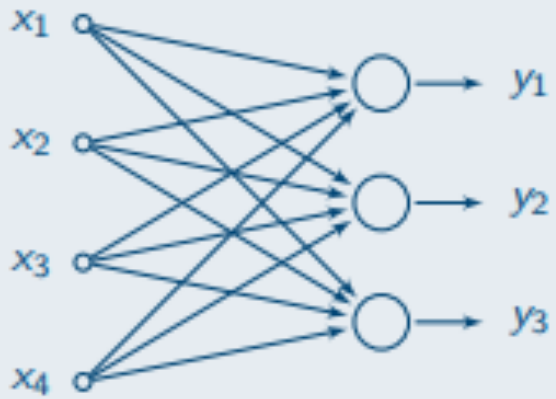
Shallow neural network

- In its most basic form, a neural network contains three layers: *input layer*, *hidden layer*, and *output layer*.
- As shown in the following figure, a network with just one hidden layer is termed a *shallow neural network*.



Fully Connected

Layer



$$\mathbf{y} = f(\mathbf{W}^T \cdot \mathbf{x} + \mathbf{b})$$

CLASS `torch.nn.Linear(in_features, out_features, bias=True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters:

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: `True`

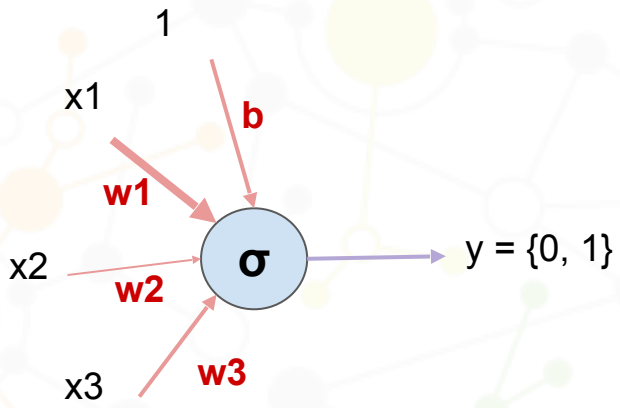
Shape:

- Input: $(N, *, \text{in\_features})$ where $*$ means any number of additional dimensions
- Output: $(N, *, \text{out\_features})$ where all but the last dimension are the same shape as the input.

Variables:

- **weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Fully Connected: A perceptron

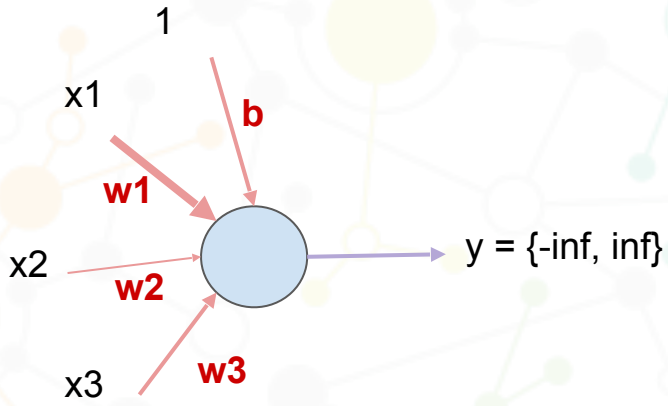


Fully connected layer with one unit. A sigmoid activation makes it a **logistic regression** (binary linear classifier):

```
lor = nn.Sequential(
    nn.Linear(NUM_INPUTS, 1),
    nn.Sigmoid()
)
```

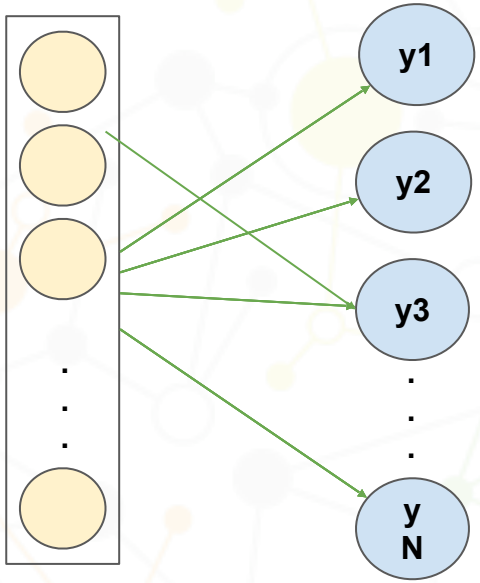
Fully Connected: A perceptron

Fully connected layer with one unit. No activation makes it a **linear regression**:




```
lir = nn.Sequential(
    nn.Linear(NUM_INPUTS, 1)
)
```

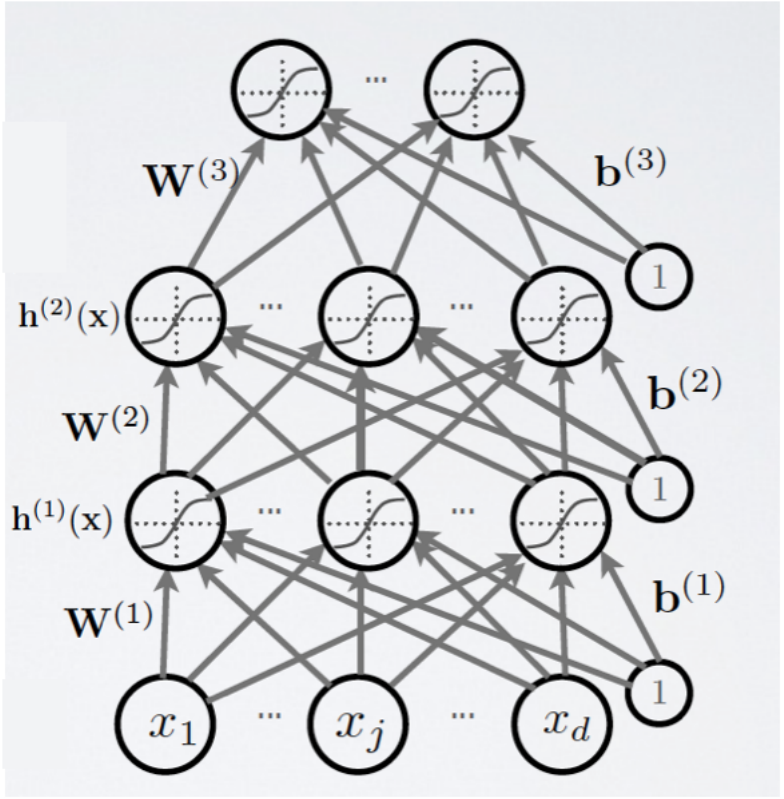
Fully Connected: Multiclass Classifier



Fully connected layer with many units. Softmax activation makes it a “softmax classifier”.

```
smx = nn.Sequential(
    nn.Linear(NUM_INPUTS, NUM_OUTPUTS),
    nn.LogSoftmax(dim=1)
)
```

Fully Connected: Multiclass Perceptron



Slide Credit: Hugo Laroche NN course

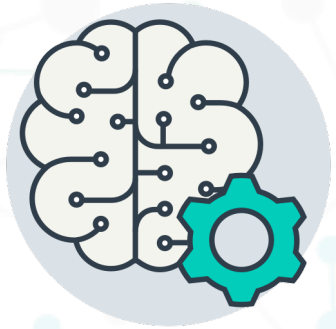
This is also a deep neural network of course.

Many fully connected layers with many units.

```
NUM_INPUTS=100
HIDDEN_SIZE=1024
NUM_OUTPUTS=20

mlp = nn.Sequential(
    nn.Linear(NUM_INPUTS, HIDDEN_SIZE),
    nn.Tanh(),
    nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE),
    nn.Tanh(),
    nn.Linear(HIDDEN_SIZE, NUM_OUTPUTS),
    nn.LogSoftmax(dim=1)
)
```

Linear Perceptron Model

What would be the grade if I study 4 hours?

4 hours →  → *? points*

Prediction

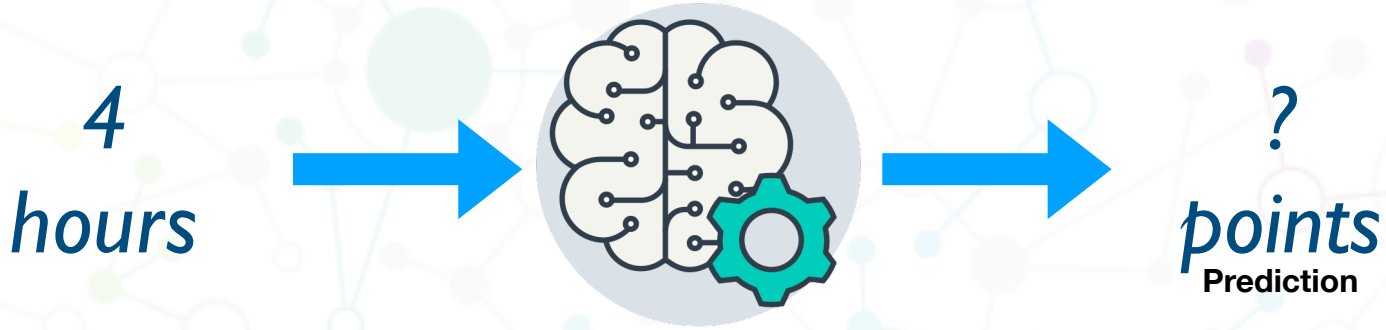
Hours (x)	Points (y)
1	2
2	4
3	6
4	?

Training dataset

Test dataset

Linear Perceptron Model

What would be the grade if I study 4 hours?



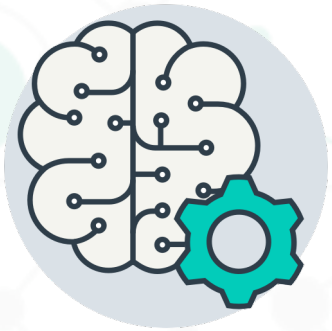
Hours (x)	Points (y)
1	2
2	4
3	6
4	?

Training dataset

Test dataset

Linear Perceptron Model

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

} Supervised learning

Linear Perceptron Model

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 Perceptron Model

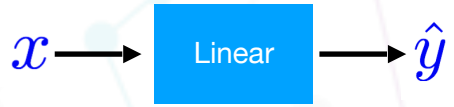
Model design

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

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

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

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



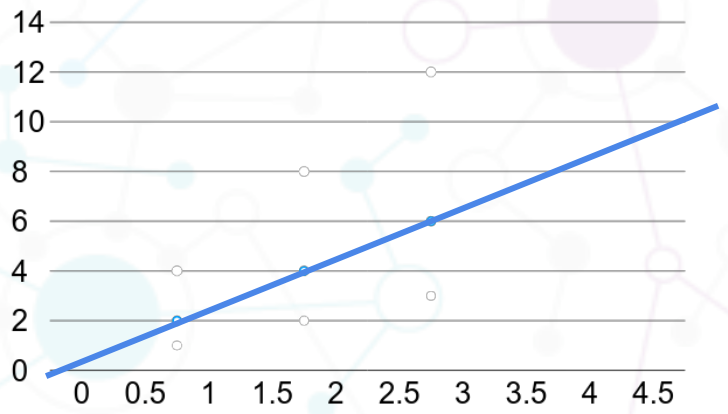
Linear Perceptron Model

Linear Regression

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

* The machine starts with **a random guess**, w=random value

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



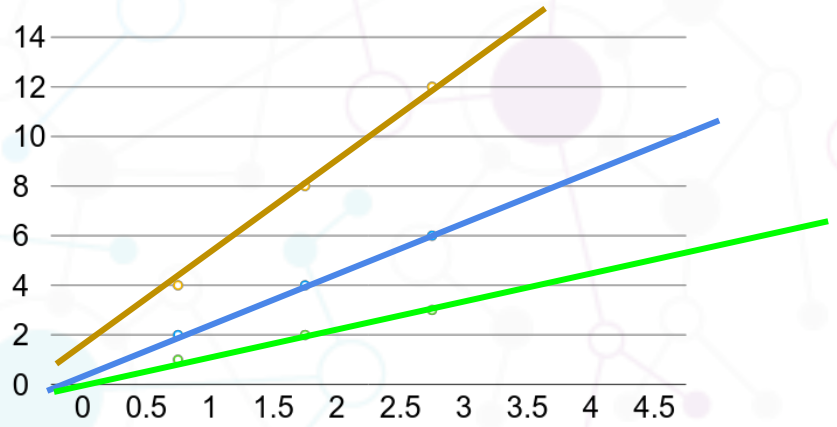
Linear Perceptron Model

Linear Regression error?

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

* The machine starts with **a random guess**, w=random value

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



Linear Perceptron Model

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

Linear Perceptron Model

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

Linear Perceptron Model

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

Linear Perceptron Model

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	2	0
2	4	4	0
3	6	6	0
			mean=0

Linear Perceptron Model

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

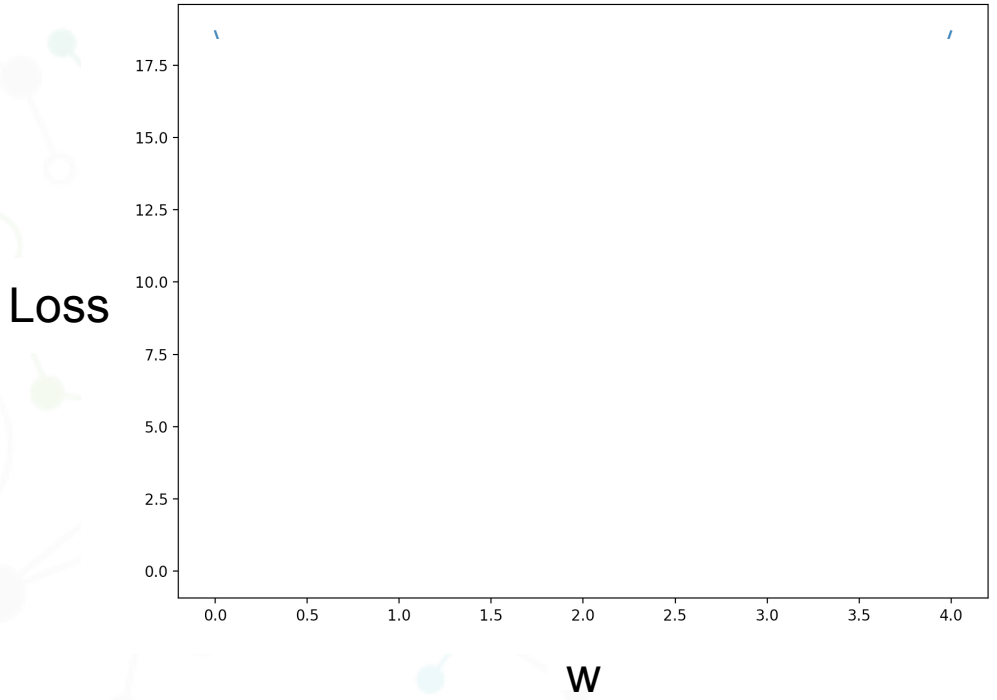
Hours, x	Loss (w=0)	Loss (w=1)	Loss (w=2)	Loss (w=3)	Loss (w=4)
1	4	1	0	1	4
2	16	4	0	4	16
3	36	9	0	9	36
	MSE=56/3=18.7	MSE=14/3=4.7	MSE=0	MSE=14/3=4.7	MSE=56/3=18.7

Linear Perceptron Model

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

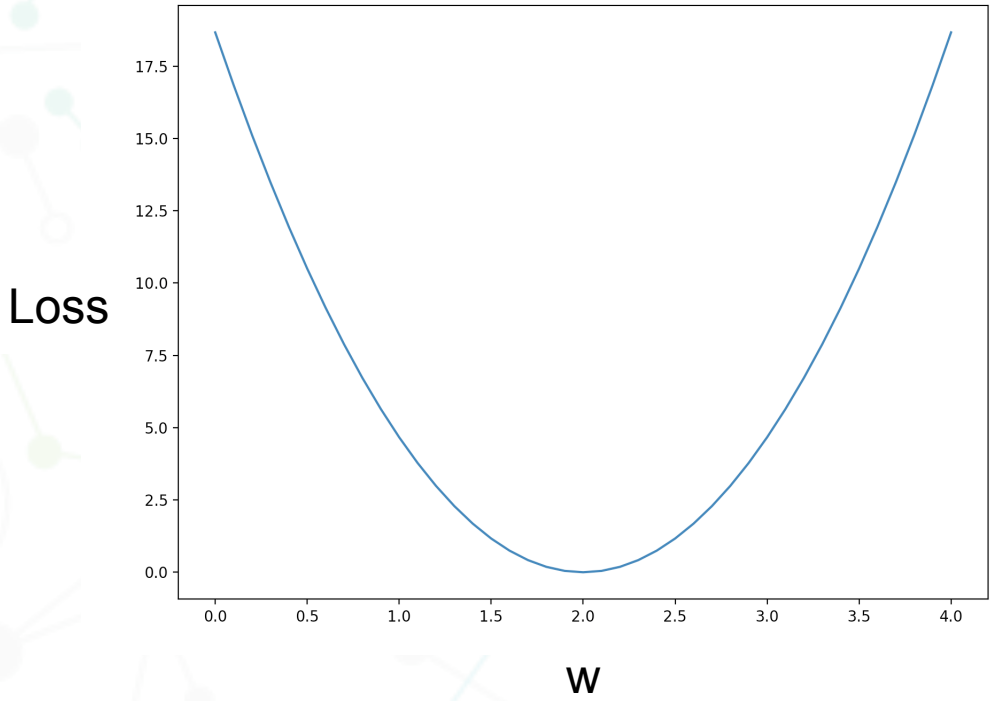


Linear Perceptron Model

Loss graph

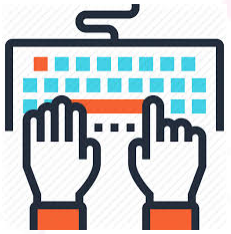
$$loss = \frac{1}{N}\sum_{n=1}^{N}(\hat{y_n} - y_n)^2$$

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



Linear Perceptron Model

Model & Loss



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

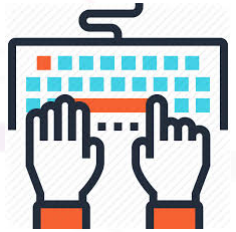
```
w = 1.0  # a random guess: random value

# our model for the 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)
```

Linear Perceptron Model

Compute loss for w

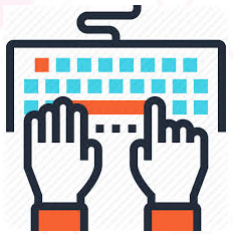


```
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val, l)

    print("MSE=", l_sum / 3)
```

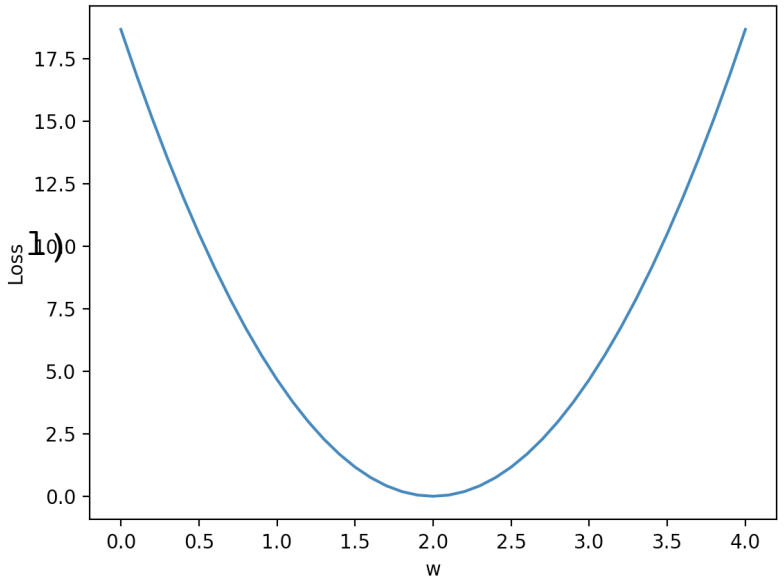
```
w= 0.0
	 1.0 2.0 0.0 4.0
	 2.0 4.0 0.0 16.0
	 3.0 6.0 0.0 36.0
NSE= 18.6666666667
w= 0.1
	 1.0 2.0 0.1 3.61
	 2.0 4.0 0.2 14.44
	 3.0 6.0 0.3 32.49
NSE= 16.8466666667
w= 0.2
	 1.0 2.0 0.2 3.24
	 2.0 4.0 0.4 12.96
	 3.0 6.0 0.6 29.16
NSE= 15.12
w= 0.3
	 1.0 2.0 0.3 2.89
	 2.0 4.0 0.6 11.56
	 3.0 6.0 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
```

Linear Perceptron Model

Plot graph

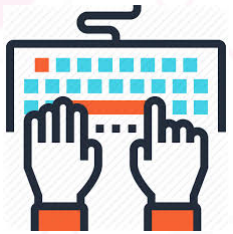


```
w_list = []
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val, l)

    print("MSE=", 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()
```

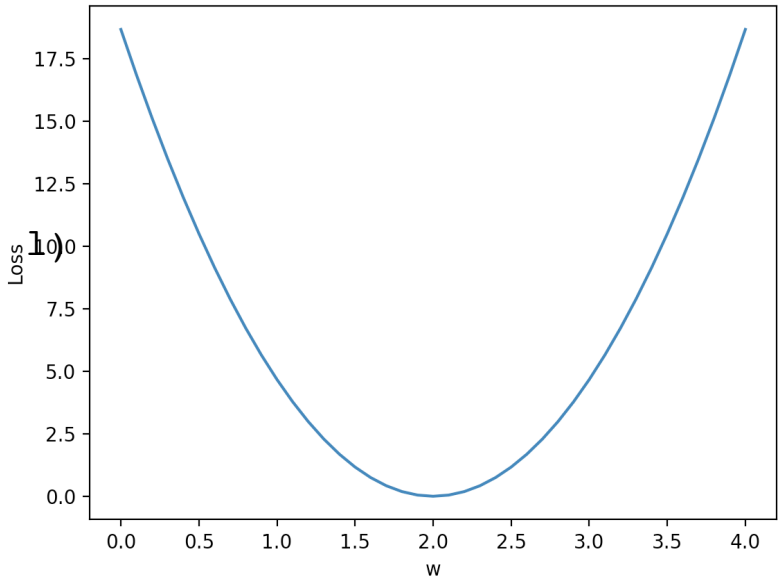


Linear Perceptron Model

Plot graph

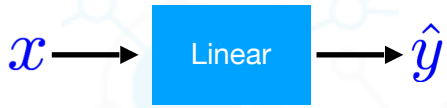


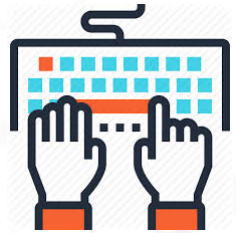
```
w_list = []
mse_list = []
for w in np.arange(0.0, 4.1, 0.1):
    print("w=", w)
    l_sum = 0
    for x_val, y_val in zip(x_data, y_data):
        y_pred_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val, l)

    print("MSE=", 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()
```



Linear Perceptron Model

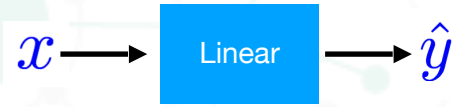

```python
import numpy as np
import matplotlib.pyplot as plt
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0  # a random guess: random value, 1.0

# our model for the 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_val = forward(x_val)
        l = loss(x_val, y_val)
        l_sum += l
        print("\t", x_val, y_val, y_pred_val, l)
    print("MSE=", 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()
```





Linear Perceptron Model

Computing gradient in simple network



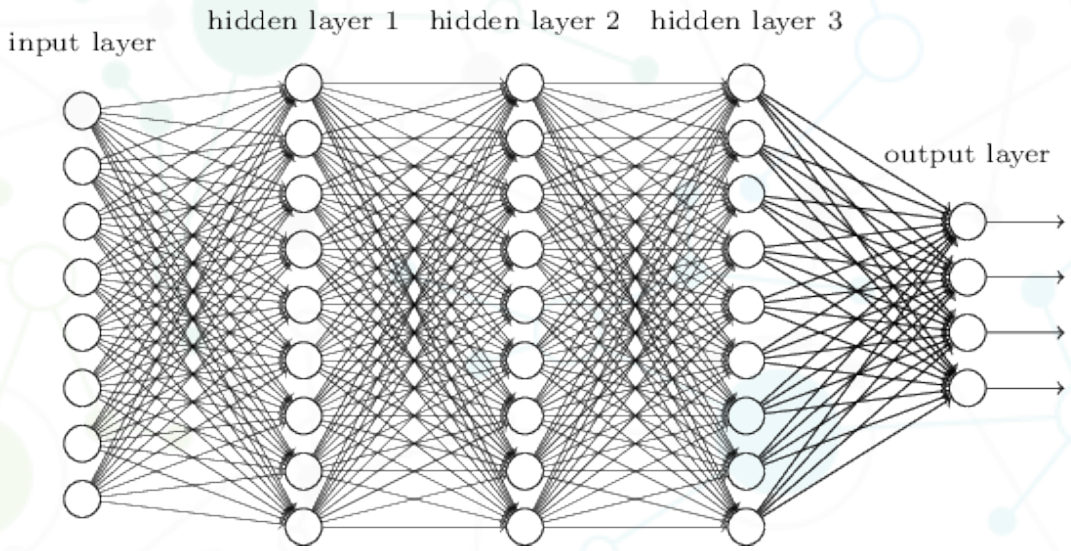
Gradient of **loss** with respect to ***w***

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

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

Linear Perceptron Model

Complicated network?



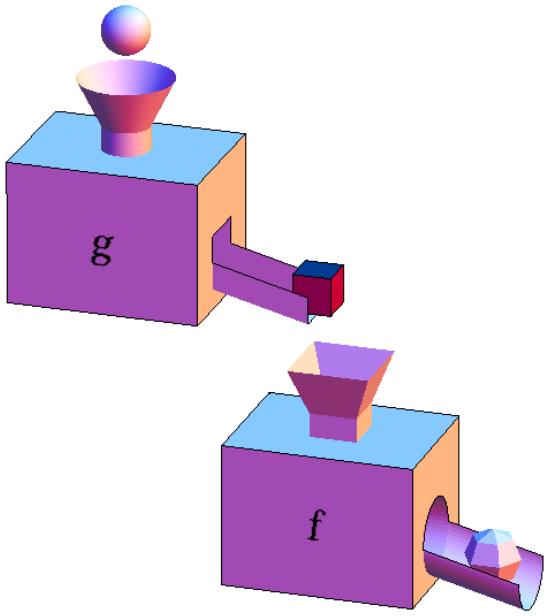
Gradient of **loss** with respect to ***w*** $\frac{\partial loss}{\partial w} = ?$

Linear Perceptron Model

Better way? Computational graph + chain rule

Linear Perceptron Model

Chain Rule



The Chain Rule

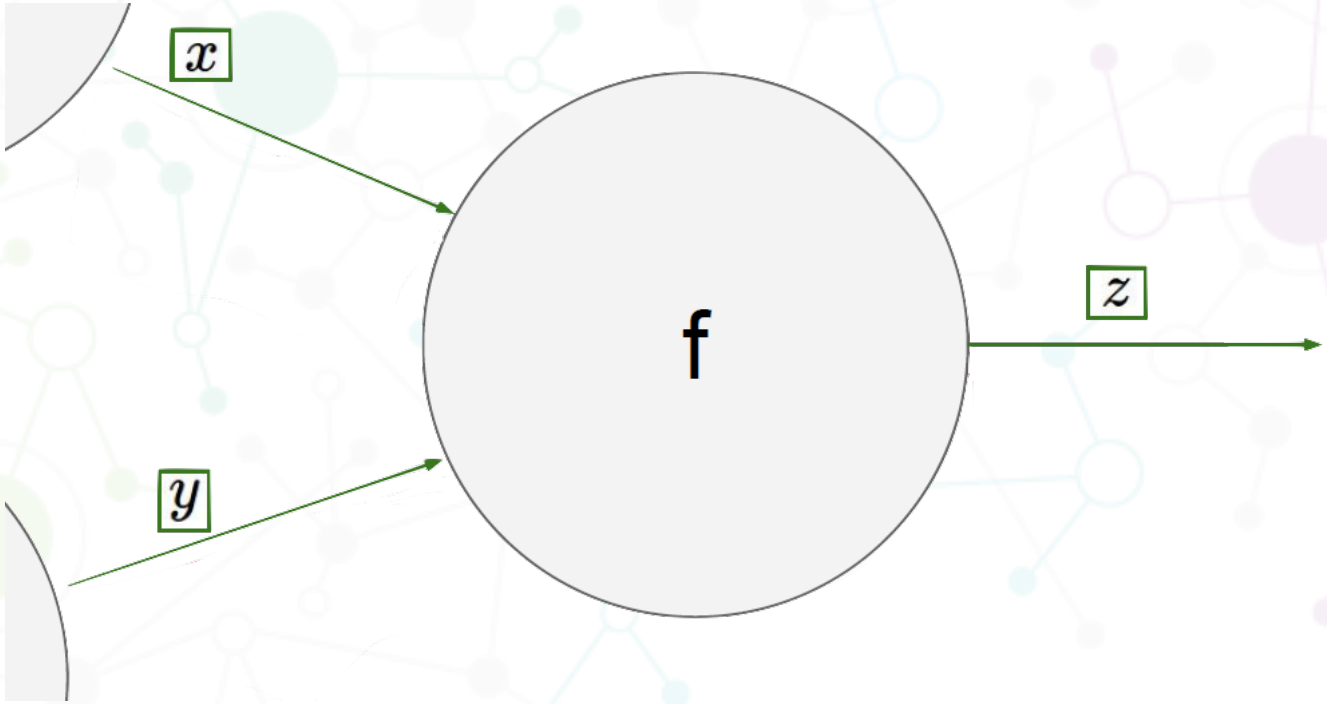
$f = f(g)\,;\ \ g = g(x)$

$$\frac{df}{dx} = \frac{df}{dg}\,\frac{dg}{dx}$$

http://mathinsight.org/image/function_machines_composed

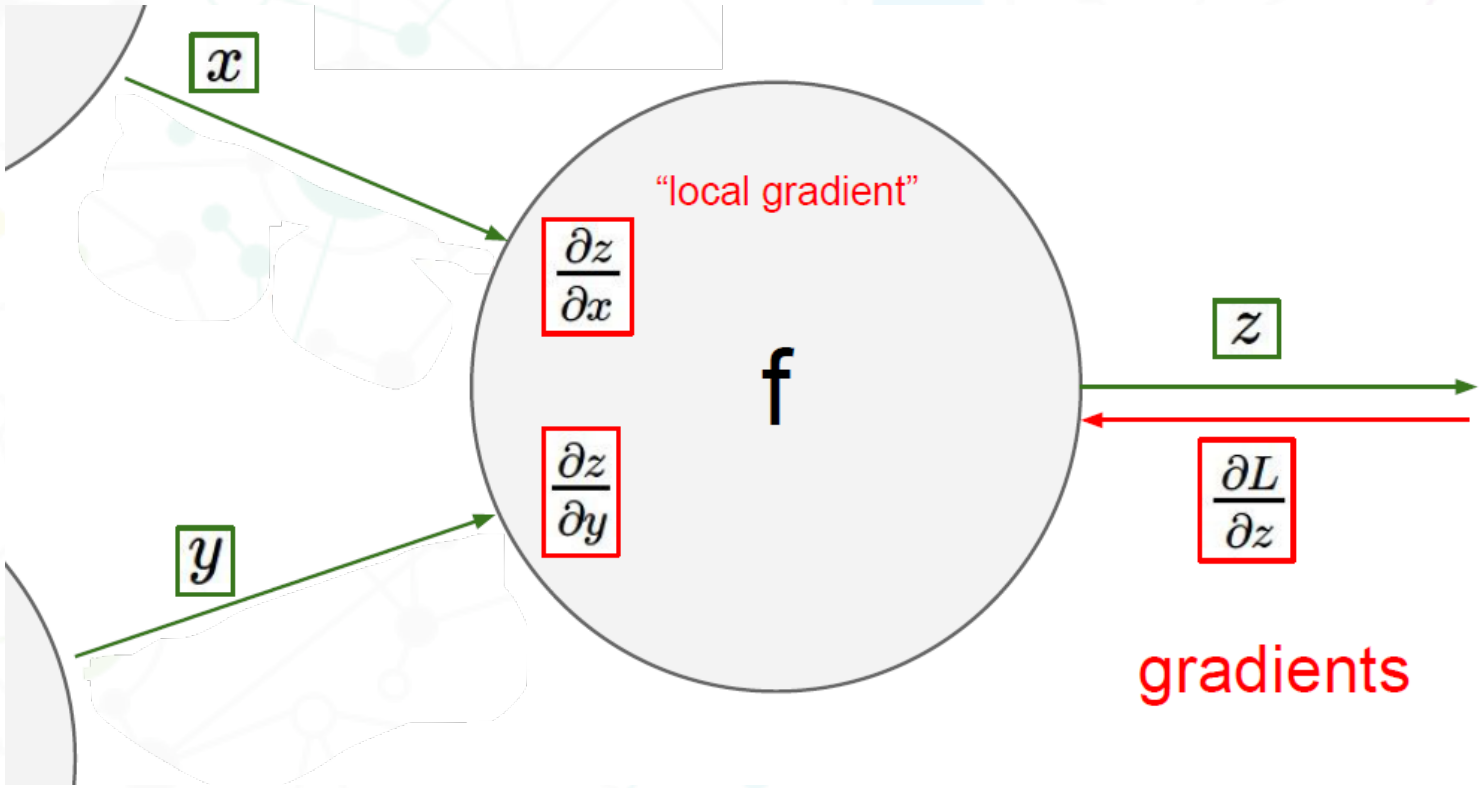
Linear Perceptron Model

Chain Rule



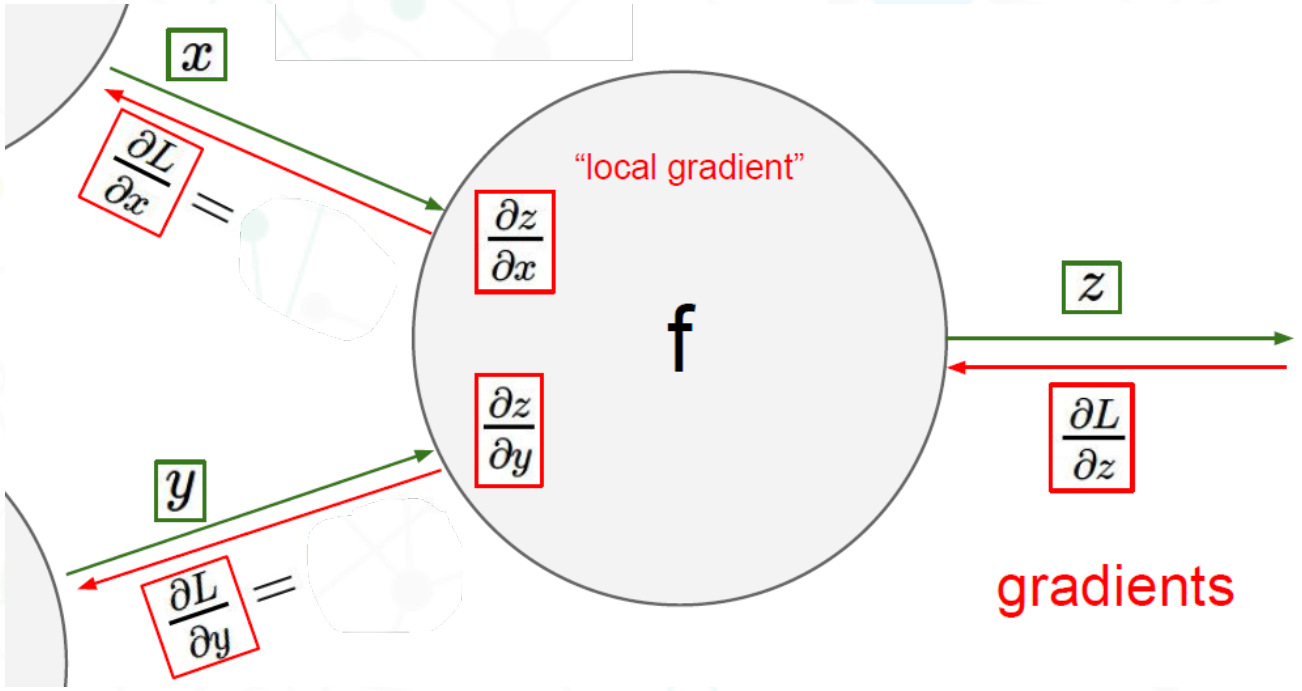
Linear Perceptron Model

Chain Rule



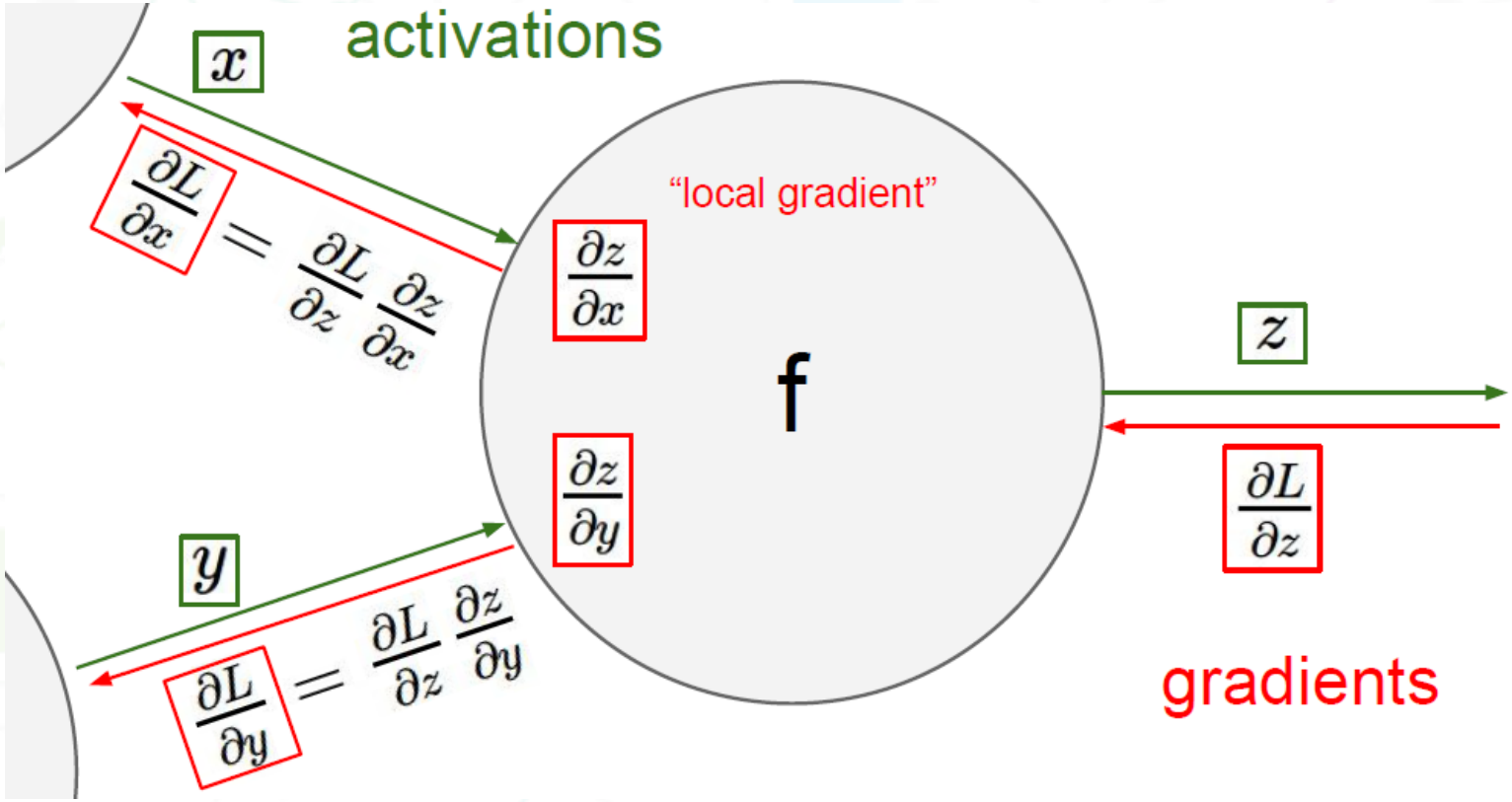
Linear Perceptron Model

Chain Rule



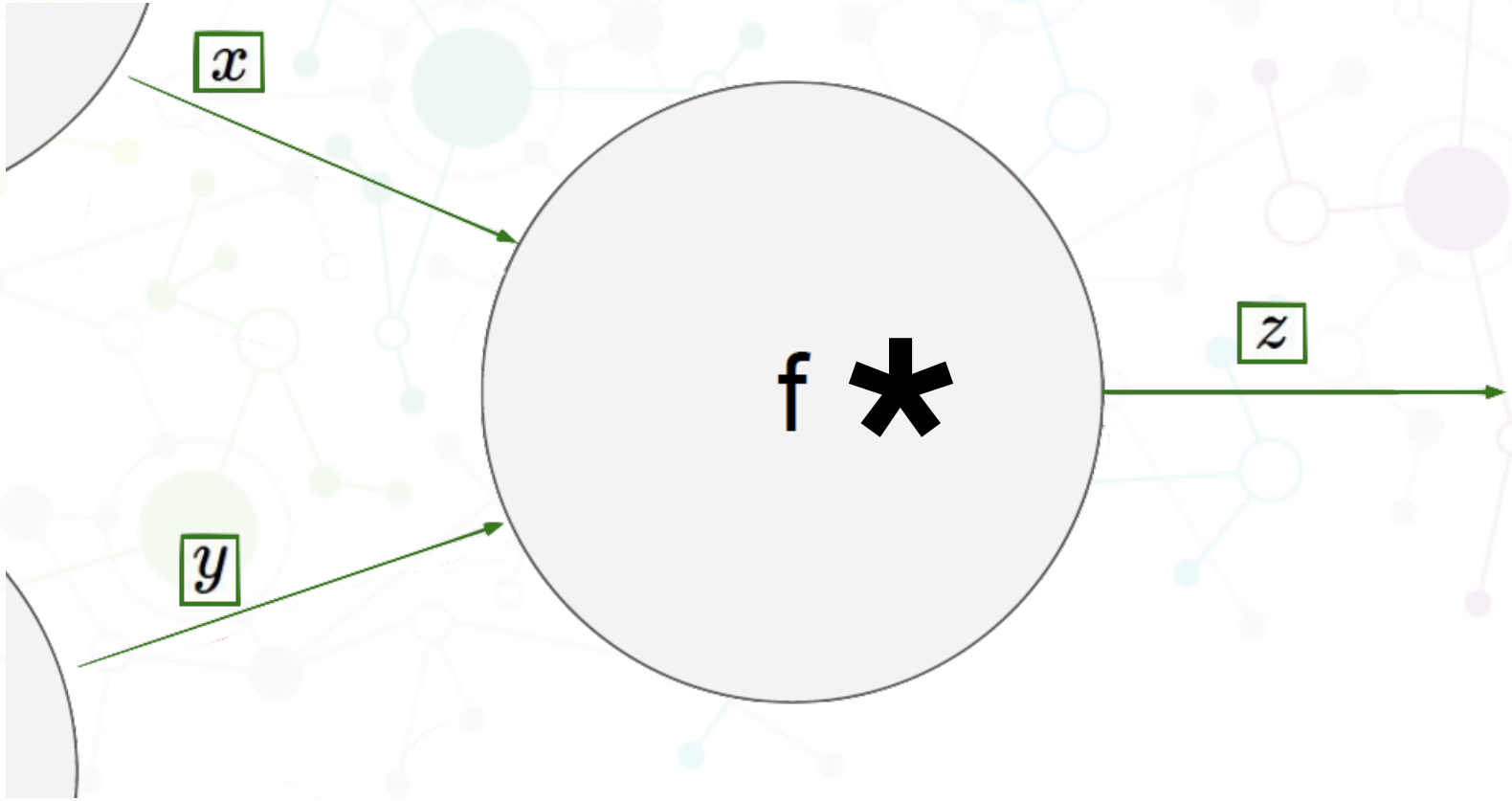
Linear Perceptron Model

Chain Rule



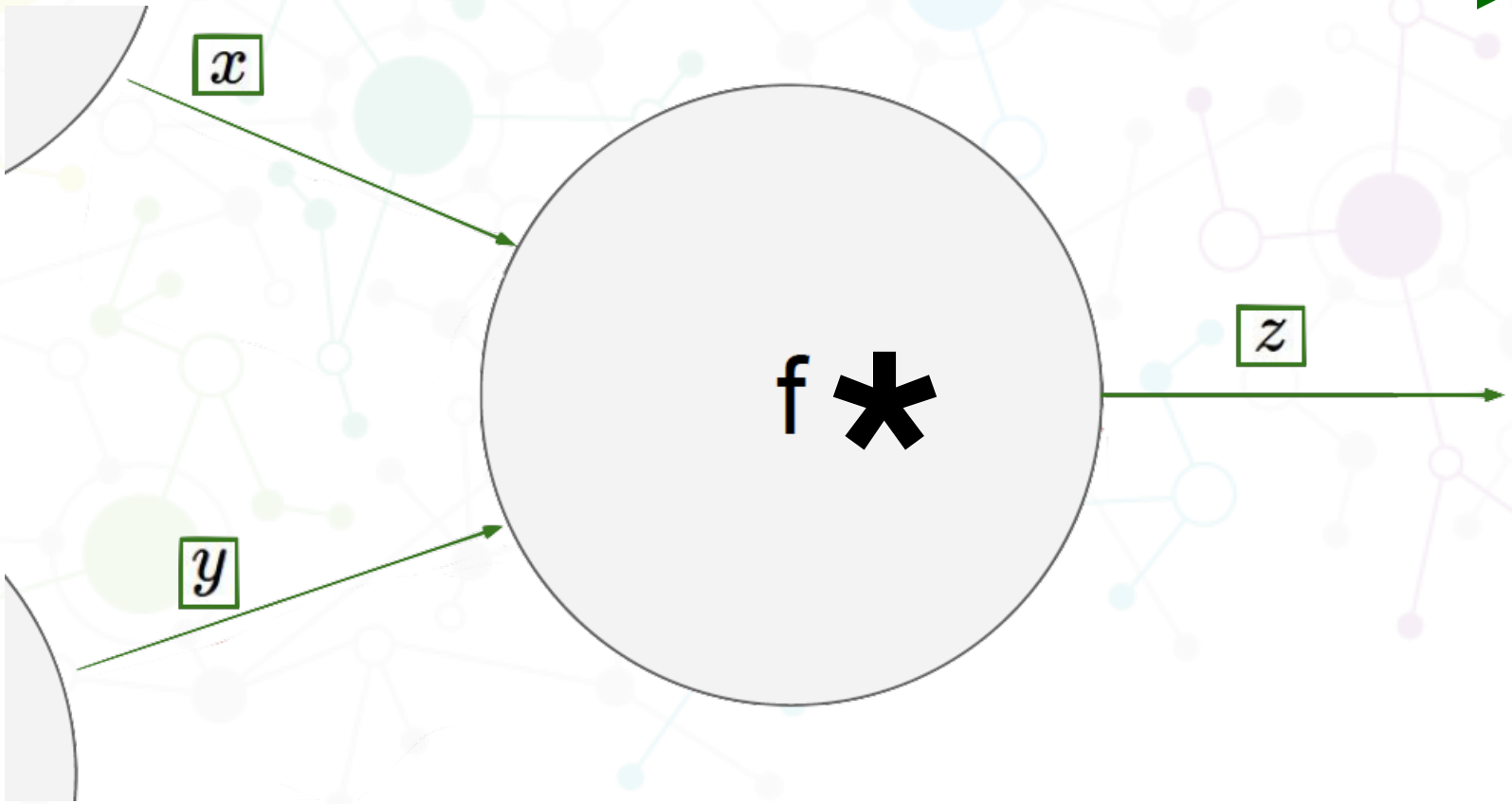
Linear Perceptron Model

Chain Rule

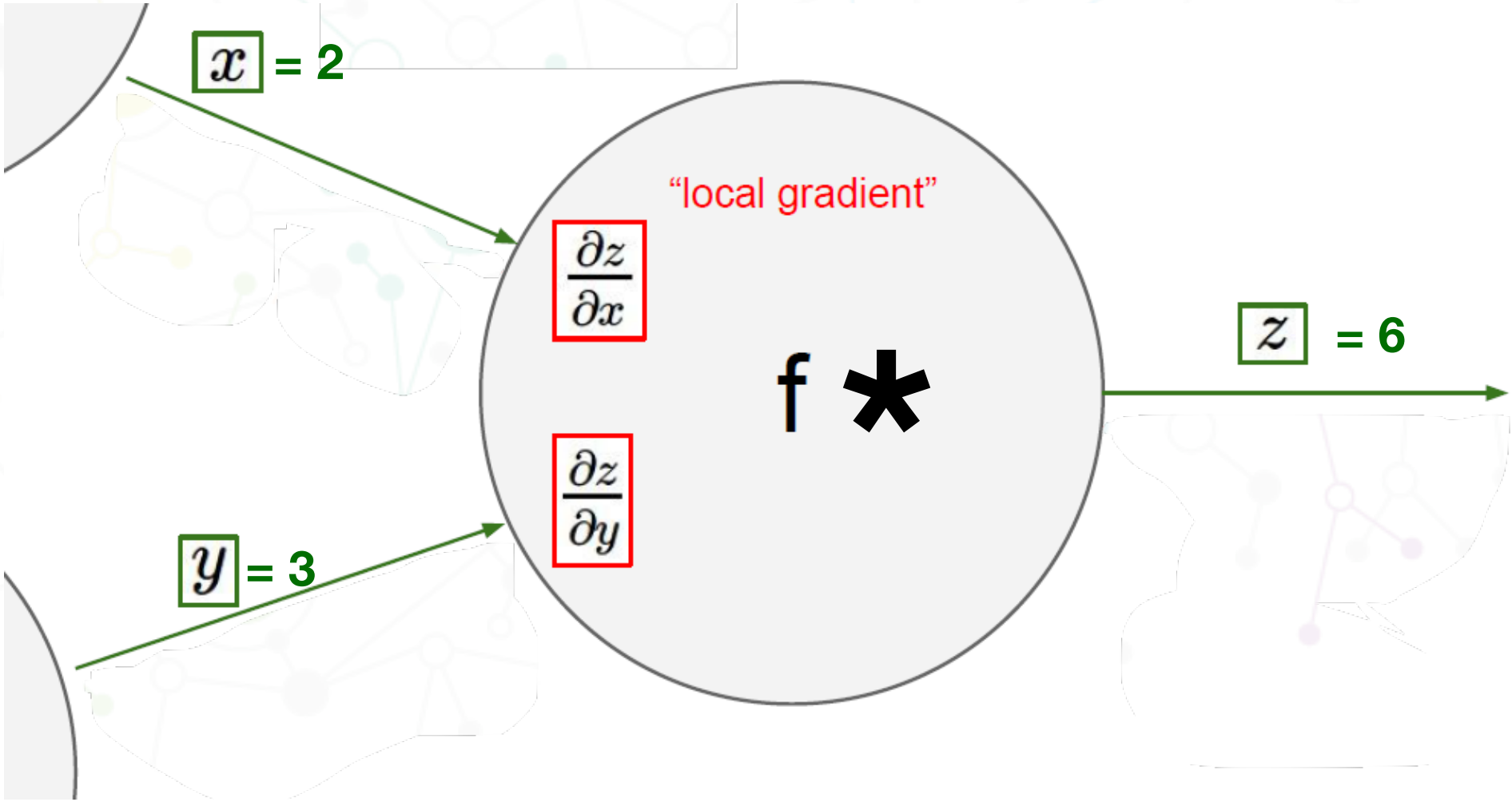


Linear Perceptron Model

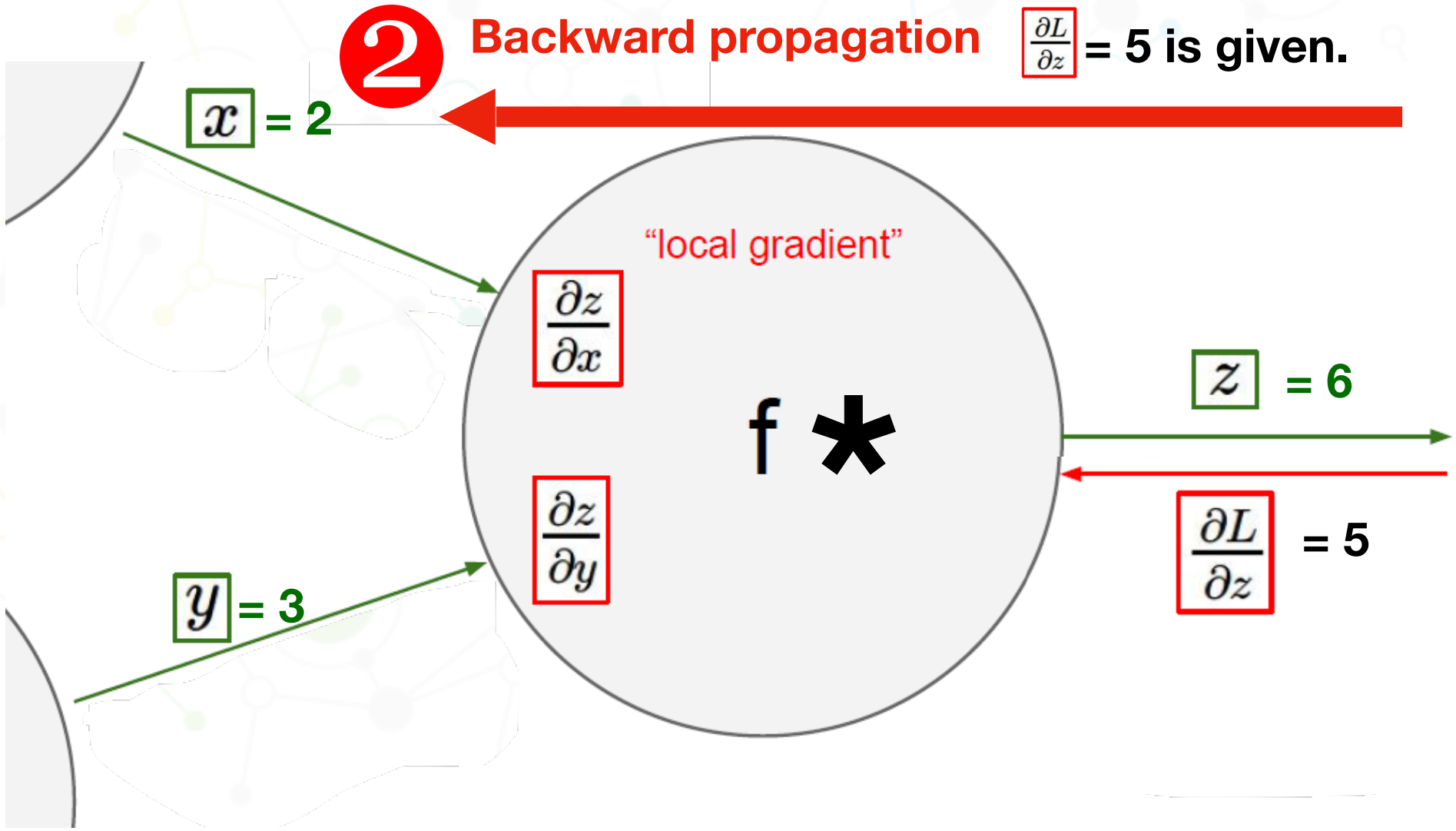
1 Forward pass x= 2, y = 3



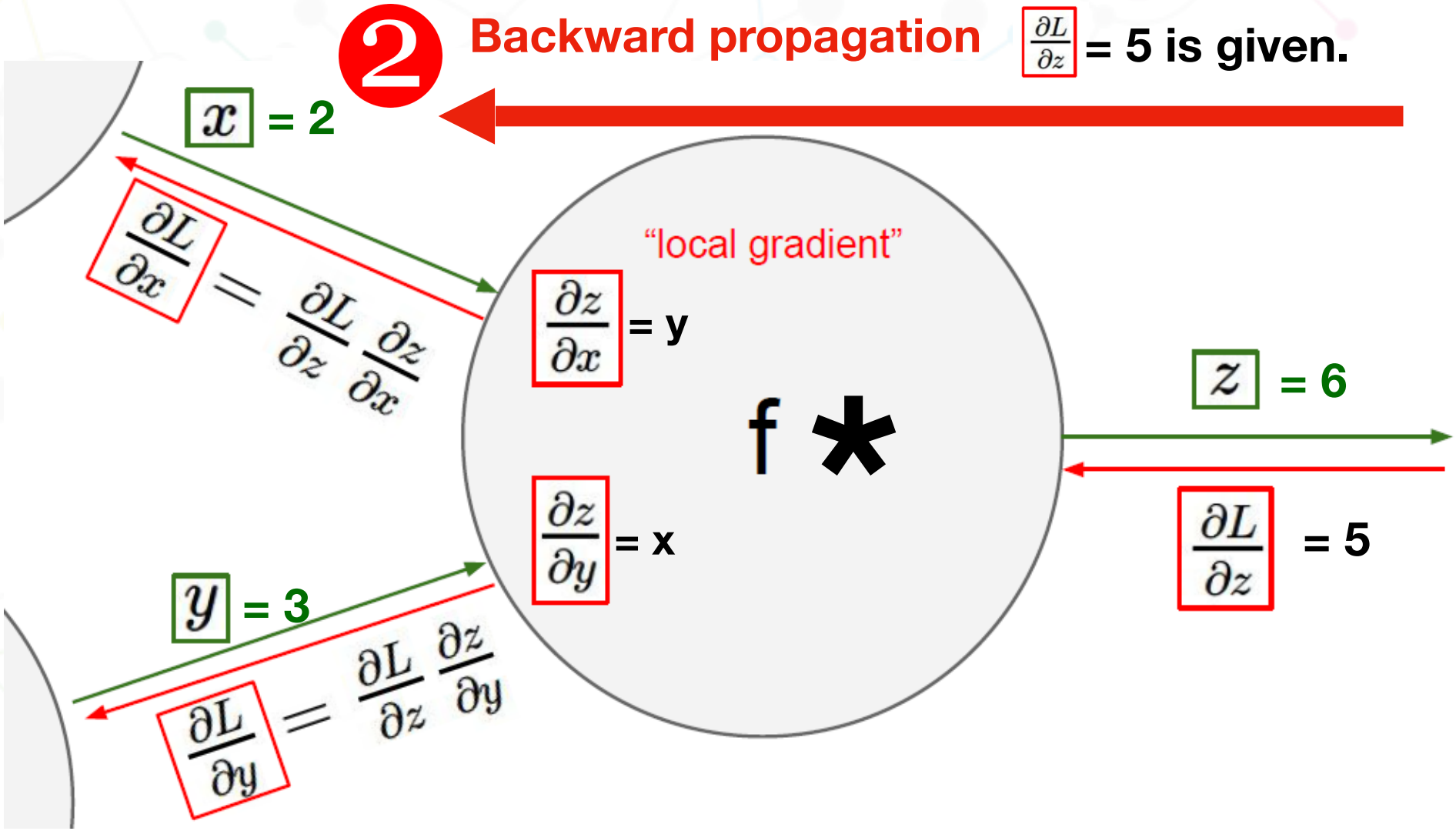
Linear Perceptron Model



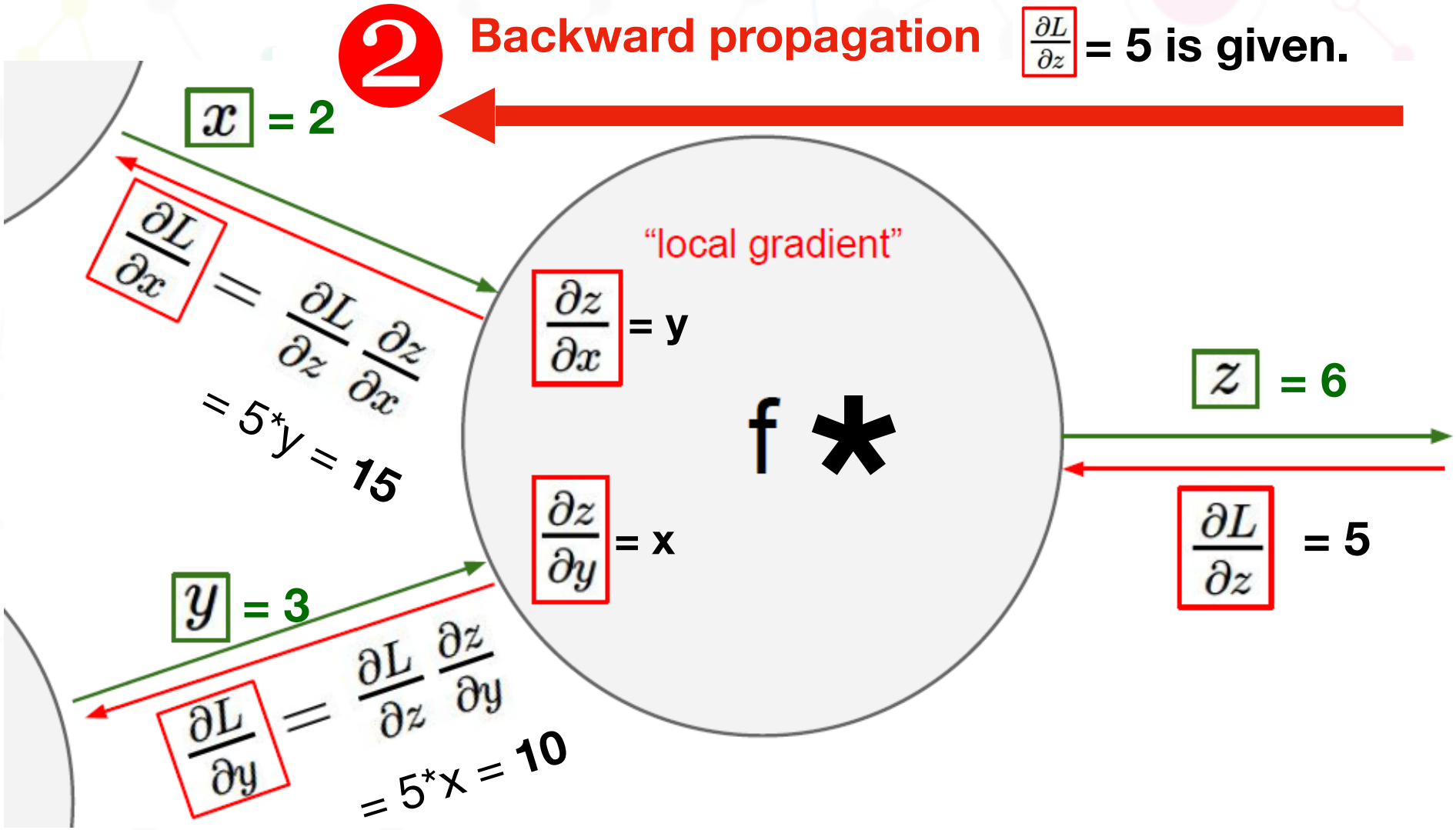
Linear Perceptron Model



Linear Perceptron Model



Linear Perceptron Model



Linear Perceptron Model

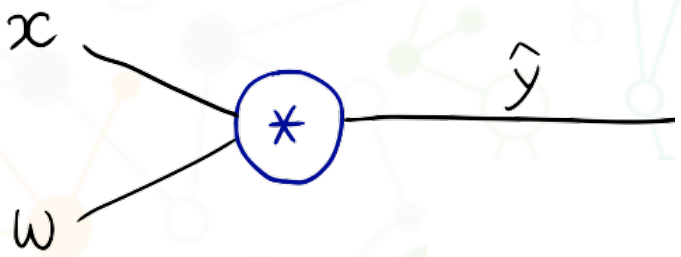
Computational graph

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

Linear Perceptron Model

Computational graph

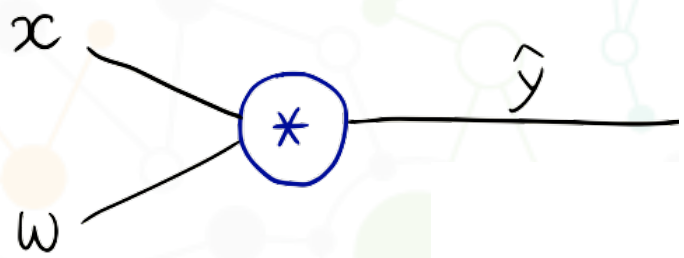
$$\hat{y} = x * w$$



Linear Perceptron Model

Computational graph

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



Linear Perceptron Model

Computational graph

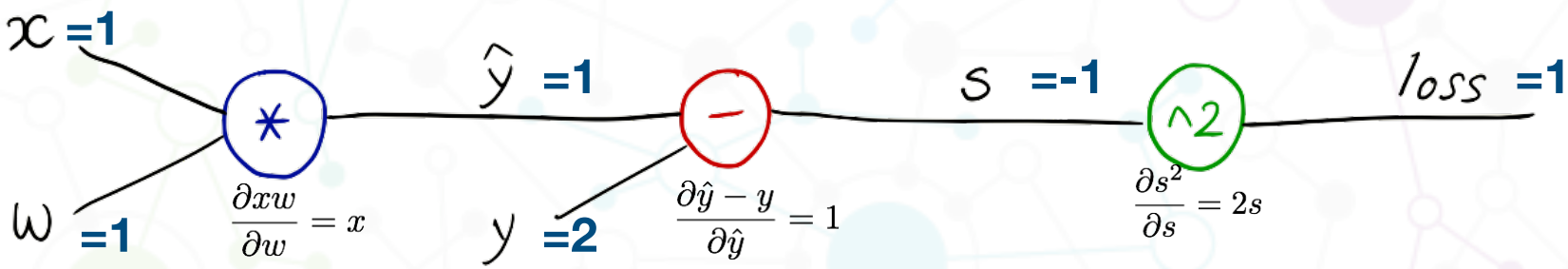
$$\hat{y} = x * w \qquad loss = (\hat{y} - y)^2 = (x * w - y)^2$$

Linear Perceptron Model

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

Linear Perceptron Model

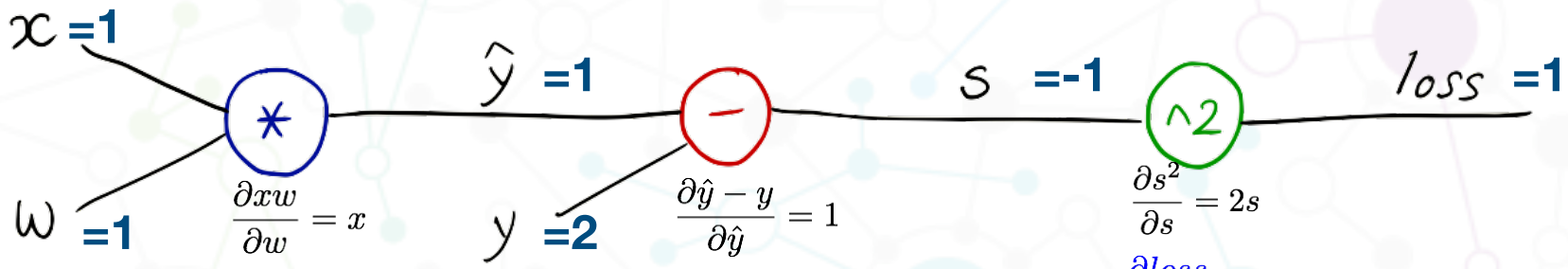
❷ Backward propagation



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

Linear Perceptron Model

2 Backward propagation



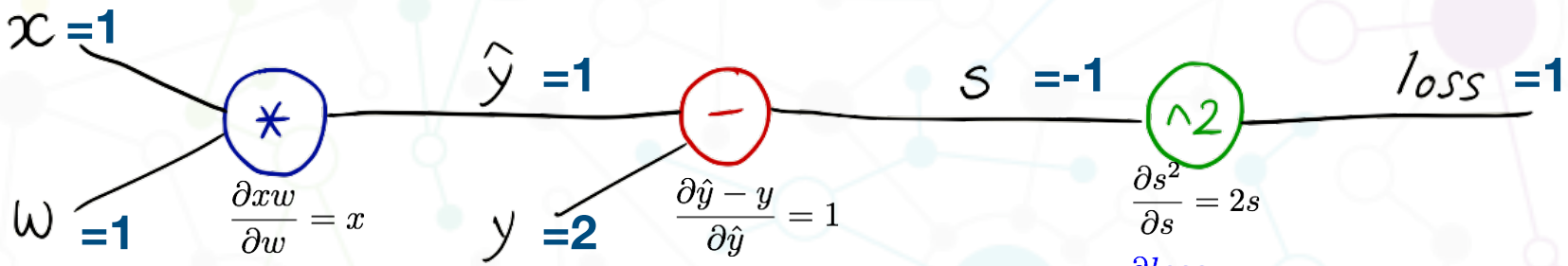
$$\frac{\partial loss}{\partial s} = 2s = -2$$

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

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

Linear Perceptron Model

2 Backward propagation



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

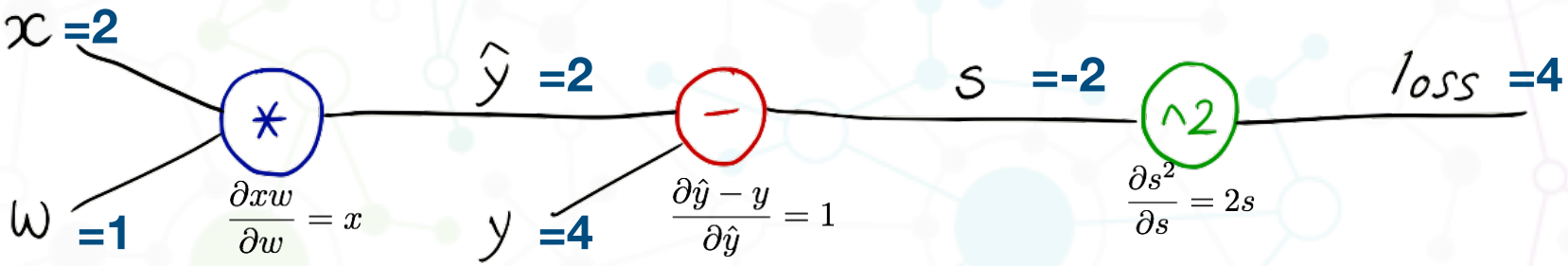
$$\frac{\partial loss}{\partial \hat{y}} = \frac{\partial loss}{\partial s}\frac{\partial s}{\partial \hat{y}} = -2 * 1 = -2$$

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

```
    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
```

Linear Perceptron Model

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

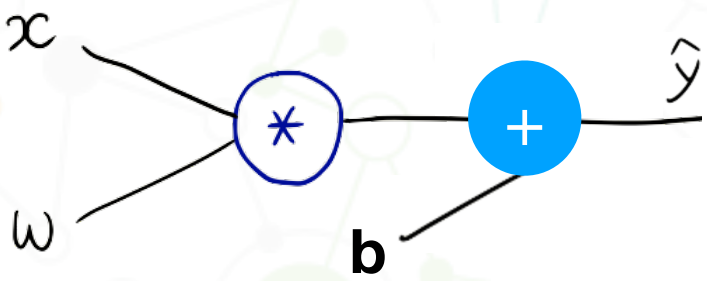


$$\frac{\partial loss}{\partial w} =$$

Linear Perceptron Model

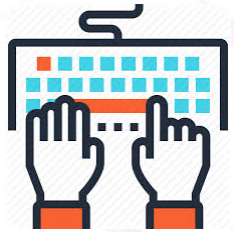
Exercise 2: x = 1, y=2, *w*=*1*, *b*=2

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



Linear Perceptron Model

Data and **Variable**

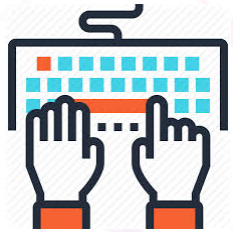


```
import torch
from torch.autograd import Variable

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

Linear Perceptron Model

Data and **Variable**

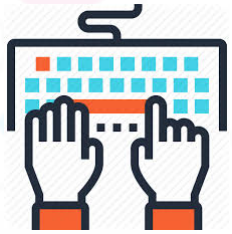


```
import torch
from torch.autograd import Variable

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

Linear Perceptron Model

Data and **Variable**



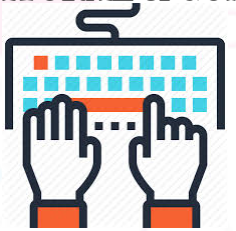
A graph is created on the fly


```
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

Linear Perceptron Model

Model and Loss



```
import torch
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)

# Before training
print("predict (before training)",  4, forward(4).data[0])
```

Training: forward, backward, and update weight

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

        # Manually zero the gradients after updating weights
        w.grad.data.zero_()

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

# After training
print("predict (after training)",  4, forward(4).data[0])
```

Linear Perceptron Model

Output

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

        # Manually zero the gradients after updating weights
        w.grad.data.zero_()

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

# After training
print("predict (after training)",  4, forward(4).data[0])
```

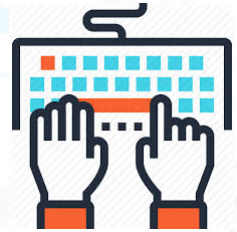
```
predict (before training) 4 4.0
    grad:  1.0 2.0 -2.0
    grad:  2.0 4.0 -7.840000152587891
    grad:  3.0 6.0 -16.228801727294922
progress: 0 7.315943717956543
    grad:  1.0 2.0 -1.478623867034912
    grad:  2.0 4.0 -5.796205520629883
    grad:  3.0 6.0 -11.998146057128906
progress: 1 3.9987640380859375
    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 -0.597504138946533 2
    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
```

Linear Perceptron Model


```
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 -0.326585221398038
    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))
```

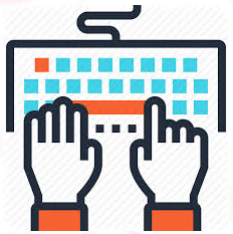
Linear Perceptron Model

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
    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
(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
    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
```



Linear Perceptron Model

PyTorch forward/backward

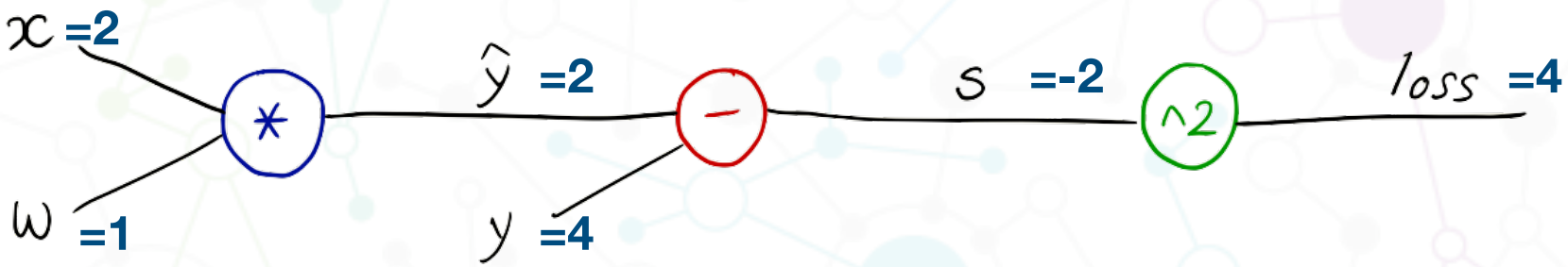
Linear Perceptron Model

Forward pass

```
# Any random value
w = Variable(torch.Tensor([1.0]),  requires_grad=True)
l = loss(x=1, y=2)
```

Linear Perceptron Model

Back propagation: l.backward()



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

Linear Perceptron Model

Weight update (step)

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

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

Linear Perceptron Model

Exercise 3: implement computational graph and backprop using NumPy

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

Linear Perceptron Model

Exercise 4: Compute gradients using computational graph (manually)

$$\hat{y} = x^2 w_2 + x w_1 + b$$
$$loss = (\hat{y} - y)^2$$

$$\frac{\partial loss}{\partial w_1} = ?$$

$$\frac{\partial loss}{\partial w_2} = ?$$

Linear Perceptron Model

Exercise 5: compute gradients using PyTorch

$$\hat{y} = x^2 w_2 + x w_1 + b$$
$$loss = (\hat{y} - y)^2$$

$$\frac{\partial loss}{\partial w_1} = ?$$

$$\frac{\partial loss}{\partial w_2} = ?$$

THANK YOU!