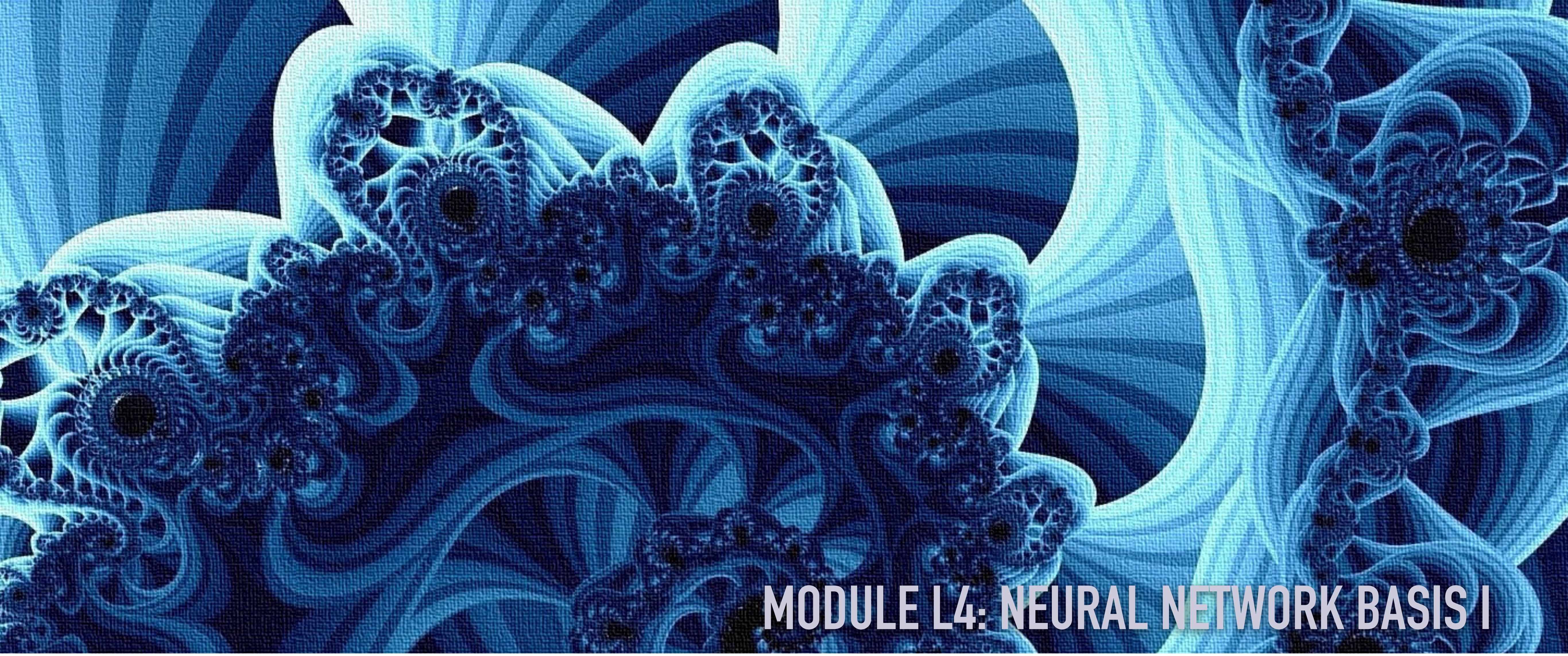
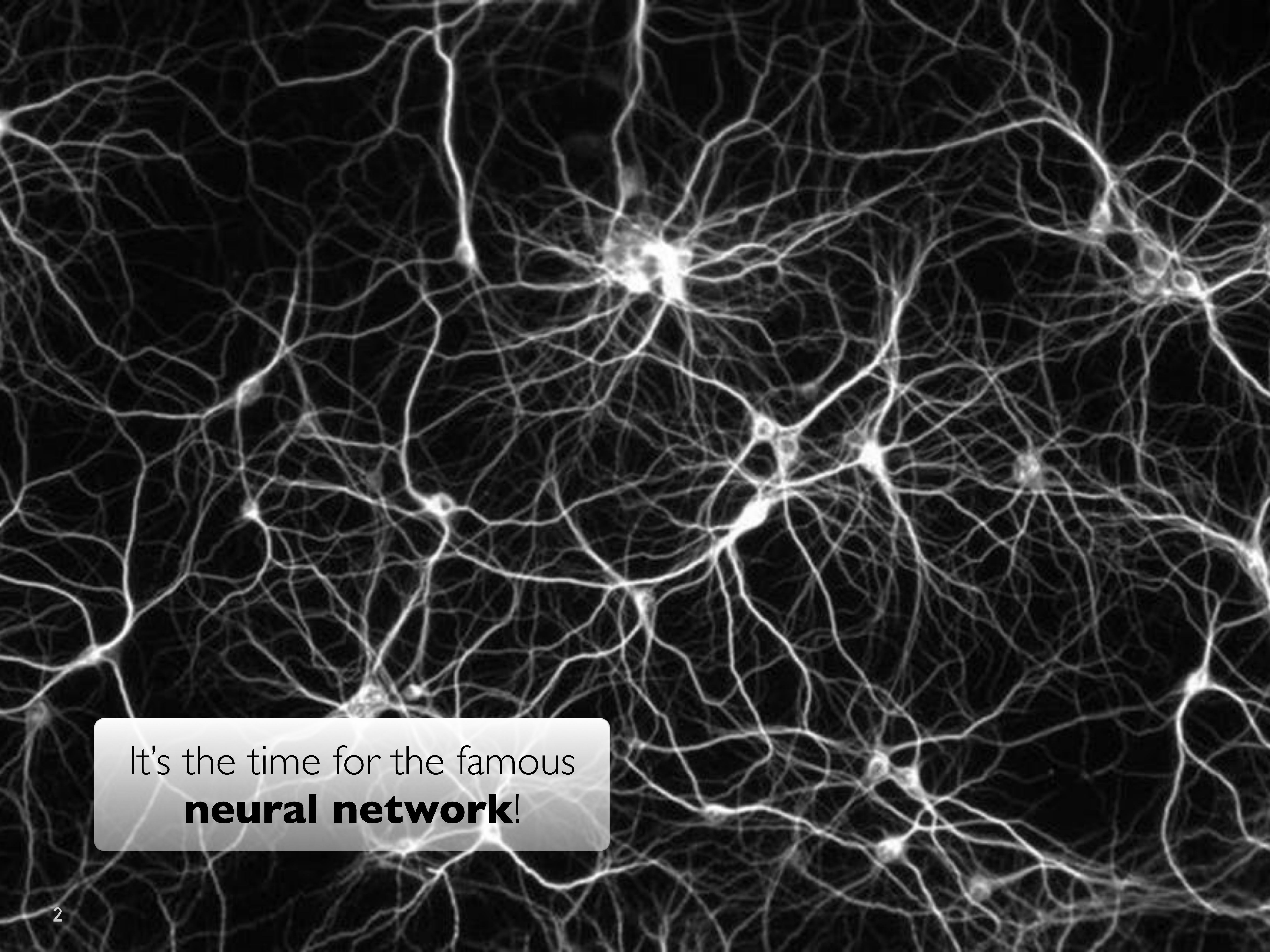


MODULE L4: NEURAL NETWORK BASIS I

INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University



It's the time for the famous **neural network**!

ARTIFICIAL NEURAL NETWORK

- An **artificial neural network (ANN)**, usually just called "neural network" (NN), is a mathematical model or computational model that tries to simulate the structure and/or functional aspects of biological neural networks.
- It consists of an interconnected group of **artificial neurons** and processes information. They are usually used to model complex relationships between inputs and outputs.
- And this is not a new idea at all — was first proposed in 1943 by McCulloch and Pitts. It has been developed for long and used in many applications already. However, with the recent development in the deep neural network, or **deep learning**, it becomes extremely powerful in many of the ML applications.

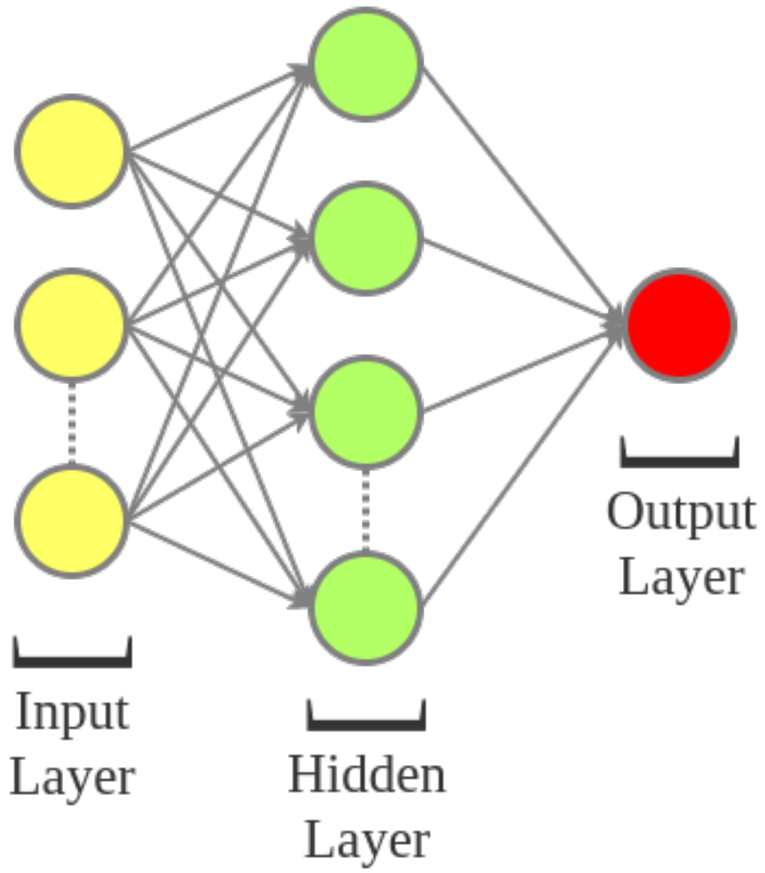
WHAT ARE WE GOING TO DO HERE?

- As we have been doing for many weeks already, we want to talk about the core concepts of the method and algorithm, maybe implement a simple version for helping the understanding. And we will switch to one of the existing packages for further practice.
- So in this module we are going to introduce and **implement a simple NN** and show you how things work. After that we will demonstrate how to do exactly the same thing with the popular packages (**Keras** and **Tensorflow**).
- For a further improvement of the network, we will touch it in the next modules and eventually approach to the idea of deep learning.

Hopefully this will give you a slightly better understanding of **“why it can work”** than just show you the modern fancy tools!

PERCEPTRONS AND NEURON MODELS

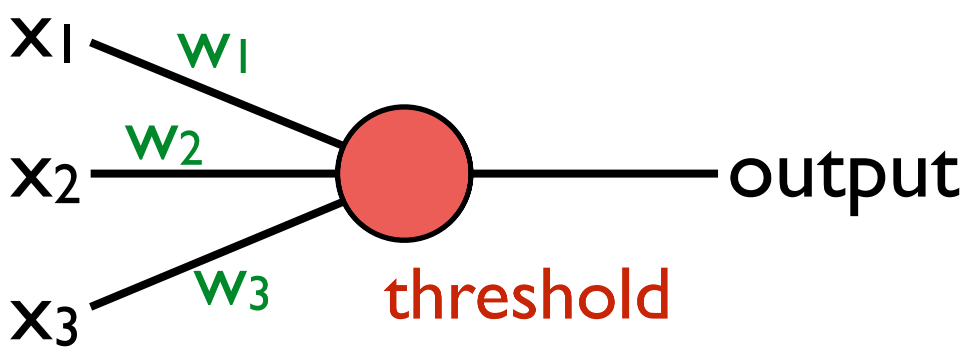
- You may have seen such a neural network schematics from many various sources!
- It shows that a network is usually built by connecting multiple neurons (the “circles” in the diagram). Hence the **neurons** are the most basic building block of the neural network.



- Here the first step is going to implement a very classical neuron model, called the **“sigmoid neurons”**, but however, it would be nice to first understand the **perceptrons** before doing that!

PERCEPTRONS AND NEURON MODELS (II)

- A perceptron usually takes multiple binary inputs, e.g. x_1, x_2, …, and produces a single binary output:

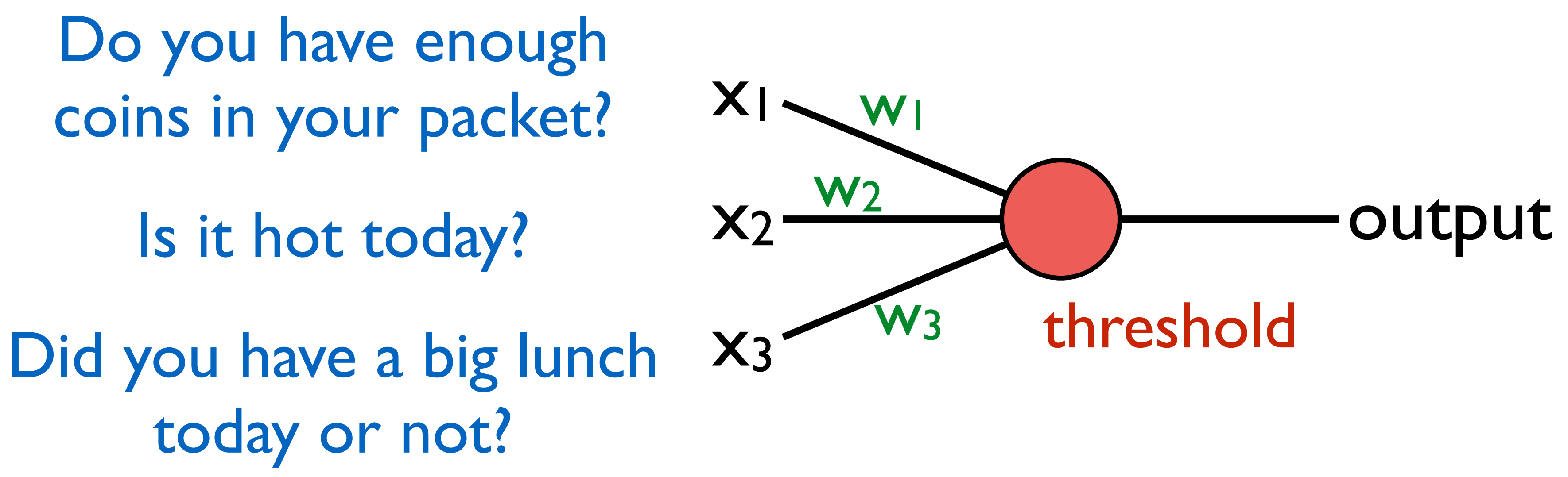


- Given the inputs are either 0 or 1, the idea is to introduce a weight for each input and estimate the **weighted sum of the inputs**, $\sum w_i x_i$. The output can be determined by whether the weighted sum bypass a **threshold** or not, just like a **logic gate**:

$$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \text{threshold} \\ 0 & \text{if } \sum w_i x_i \leq \text{threshold} \end{cases}$$

PERCEPTRONS AND NEURON MODELS (III)

- It's all about the decision making — you can in fact, model a daily life problem with such a simple perceptron model. For example, consider a decision of having an ice cream cone or not:



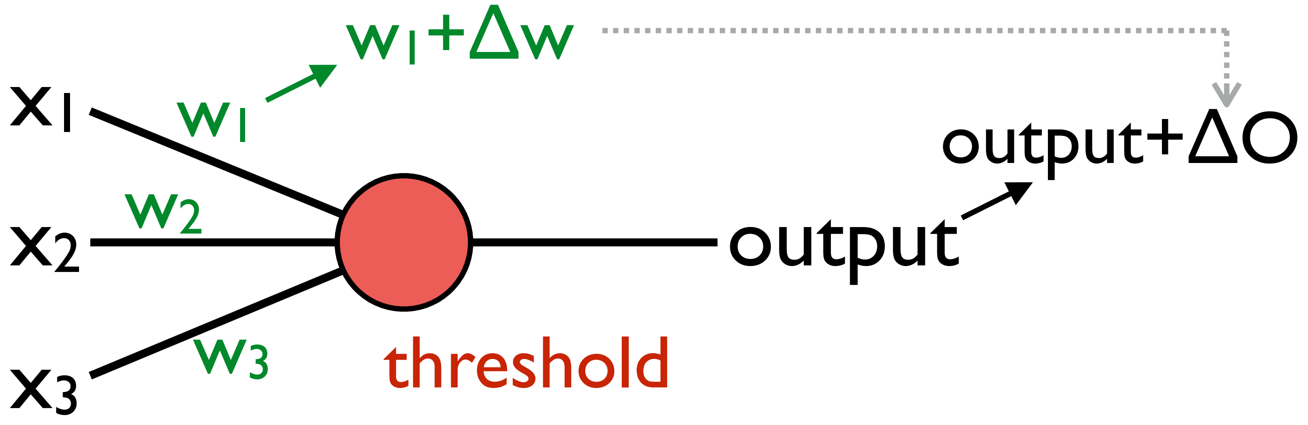
Do you have enough coins in your packet?
Is it hot today?
Did you have a big lunch today or not?
x1
x2
x3
w1
w2
w3
threshold
output

Eat or not to eat, that's the question!

  - **Weights:** decides the importance of each inputs, *e.g. do you care about the weather or how much food in your stomach?*
  - **Threshold:** decides the action taking criterion, *e.g. a value to determine your love of ice cream.*

SIGMOID NEURONS

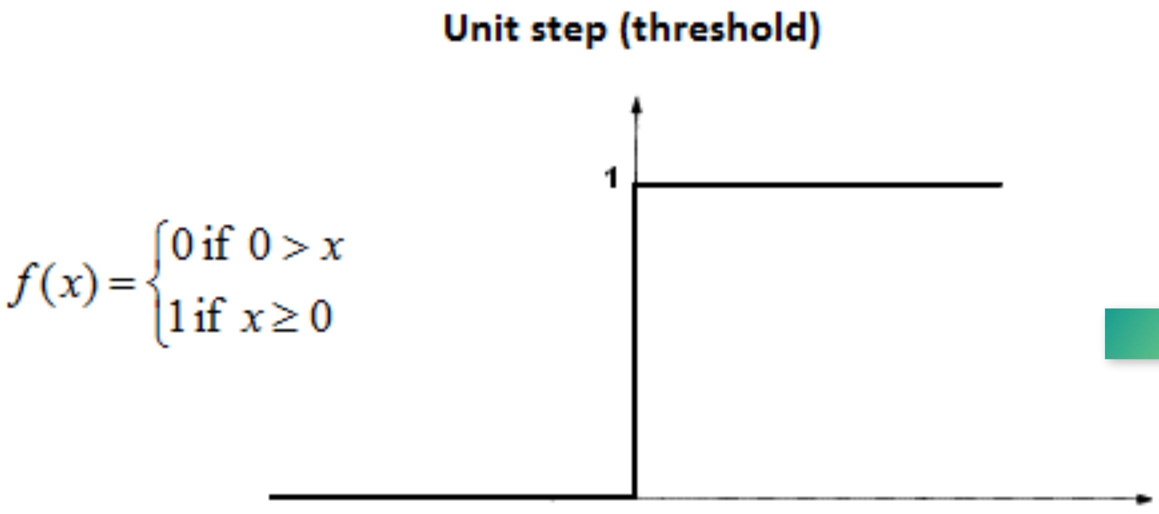
- Although a perceptron design sounds rather reasonable and it can be used to model all kinds of logic gates, but it has a critical issue in the implementation of machine learning.
- The usual learning is carried out by changing the weights or thresholds a little bit and check whether the output is improved or not. e.g.

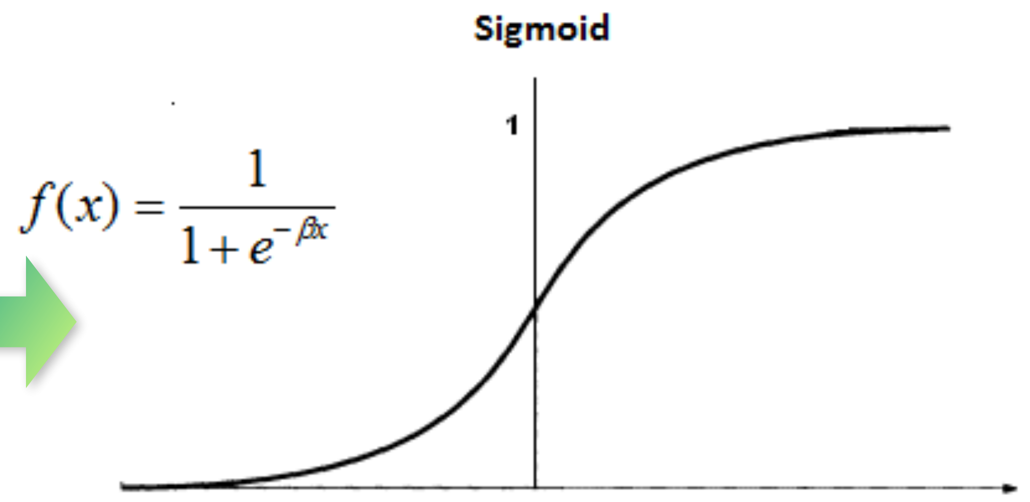


- Given the inputs and outputs are only binary in perceptrons, this will not work.

SIGMOID NEURONS (II)

- The key idea is to introduce an **activation function** at the core of neuron, which can smooth the binary operation to some continuous function and still keep the nice property of perceptrons, for example a **sigmoid function**:





$$z=\sum_i w_i x_i + b, \quad \text{output} = \sigma(z) = \frac{1}{1+\exp(-z)}$$

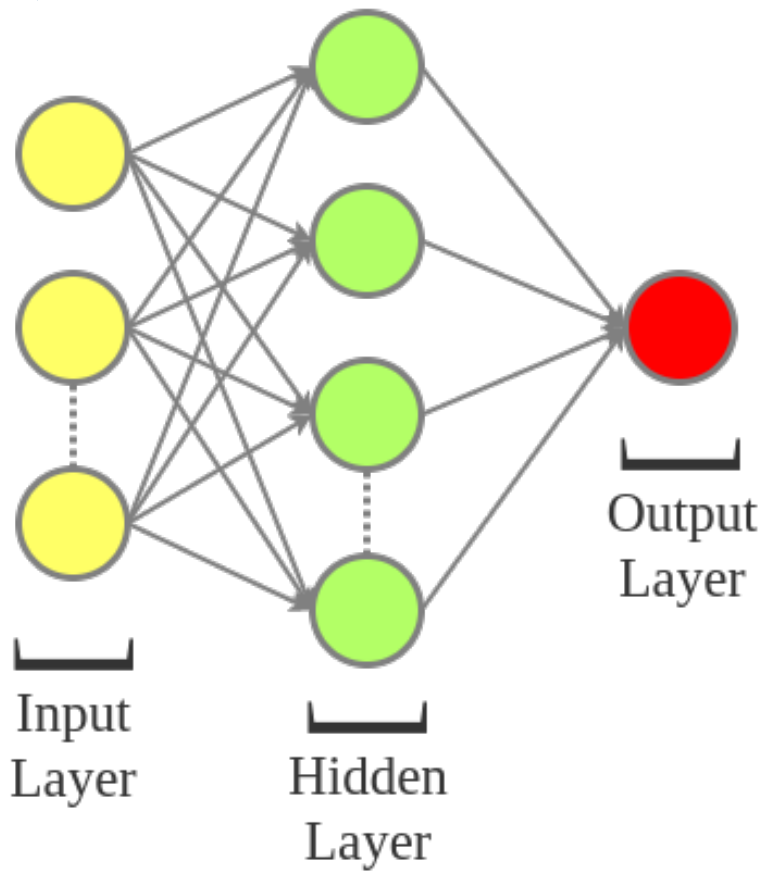
Here we take the **"bias"** to replace **"–threshold"**, which is mathematically the same!

$$\Delta O \approx \sum_i \frac{\partial O}{\partial w_i}\Delta w_i + \frac{\partial O}{\partial b}\Delta b$$

It becomes differentiability!

NETWORK ARCHITECTURE

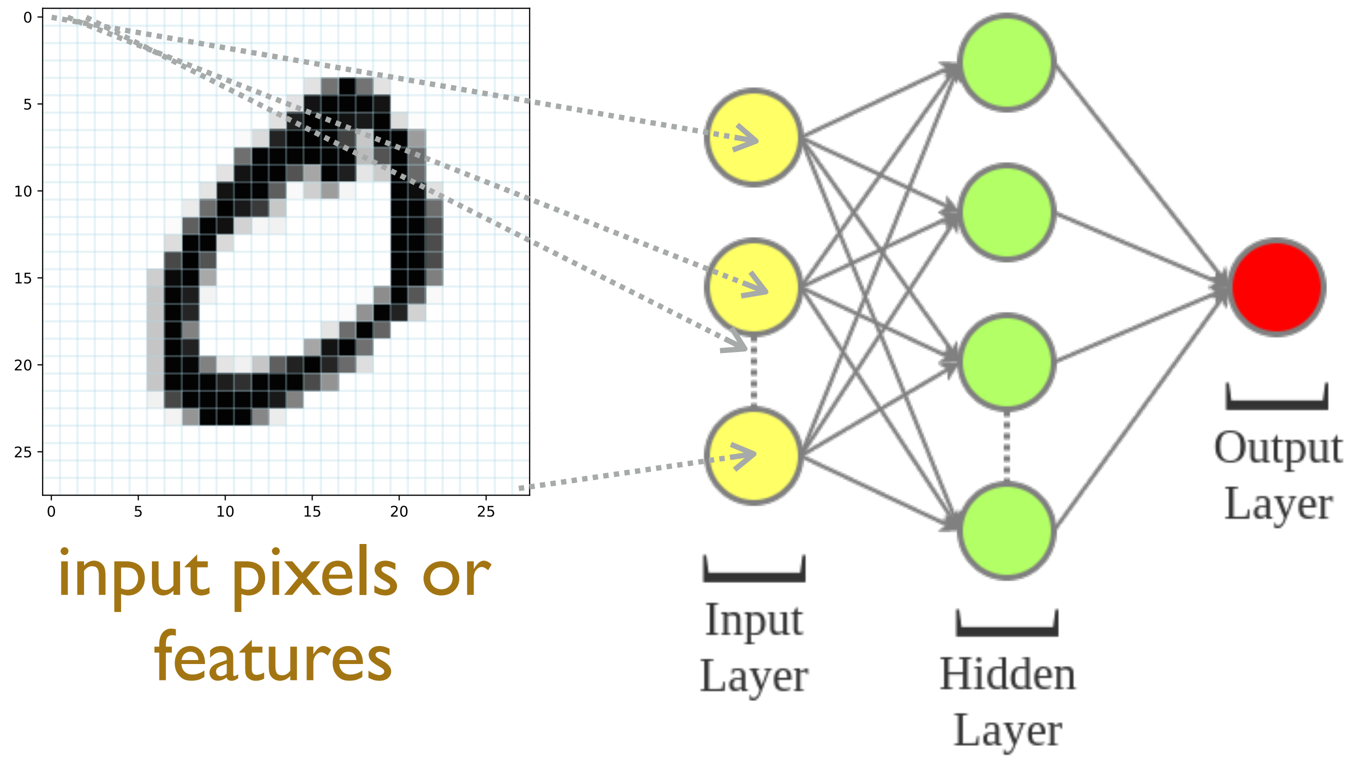
- As we just discussed, a single neuron is like a logic gate. If one want to handle a more complex problem, it is necessary to incorporate multiple neurons and hence the network architecture becomes essential.
- A typical structure is like this, as **multilayer perceptrons (MLP)**:



  - There can be multiple **input neurons**, for handling the actual input features.
  - There can be multiple **hidden layers** of neurons, without connecting to the input nor output directly.
  - There can be multiple **output neurons** as well.

NETWORK ARCHITECTURE (II)

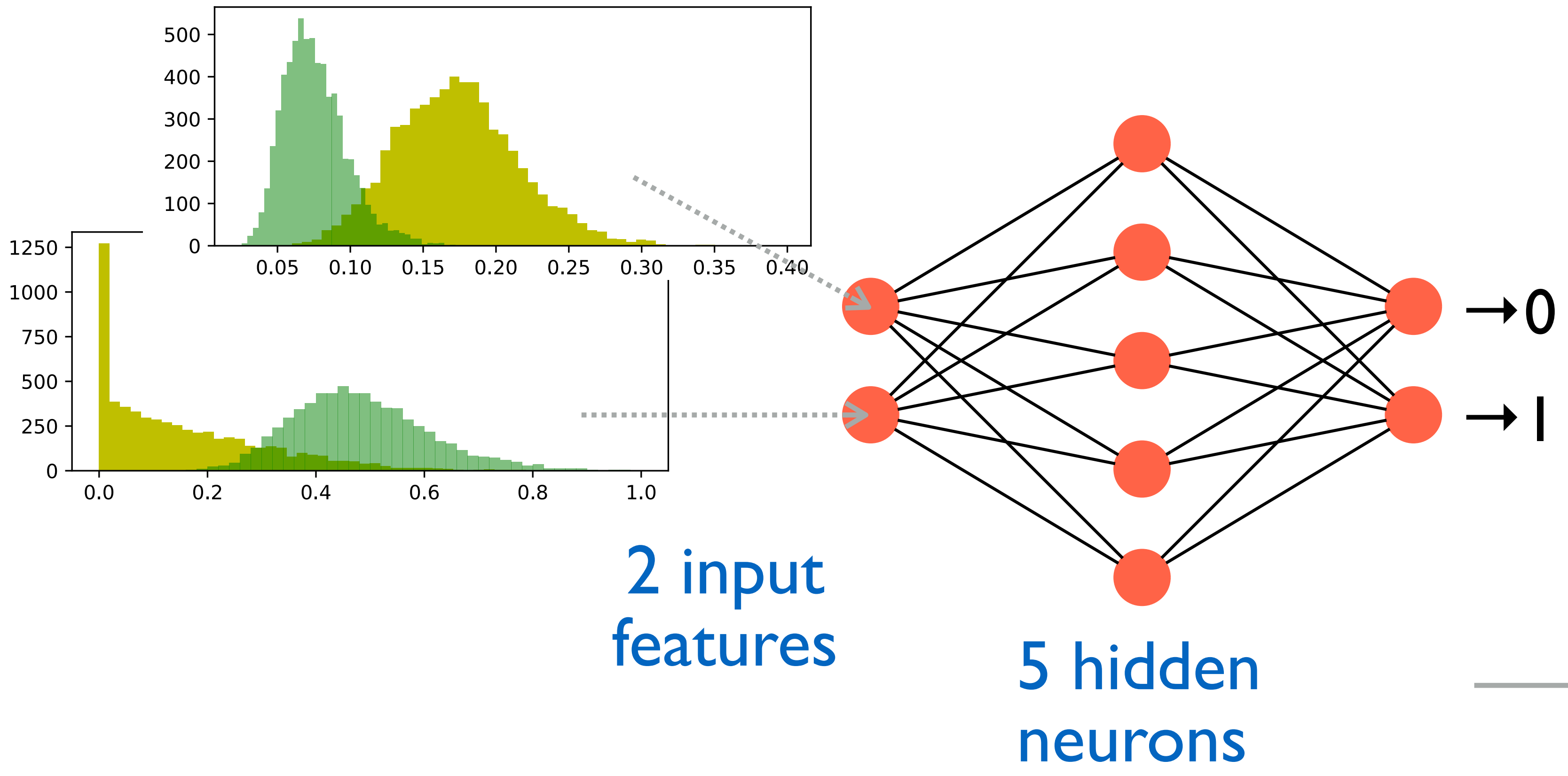
- If the network process the information only in one way (no loops) — e.g. data always feeds forward, never feeds back, this is usually called the **feedforward neural networks**.
- In such a case the whole network can be imaged as a huge and complicated function, and each output is a function of inputs, with all of the *weights* and *biases* as parameters of the function.



$$\begin{cases} \sim 1 & \text{if the input image looks like a “zero”} \\ \sim 0 & \text{if the input image } \textit{doesn't} \text{ look like a “zero”} \end{cases}$$

NETWORK ARCHITECTURE (III)

- Consider the earlier example of separating handwriting digits of zeros and ones, with the two reduced features (*only full pixel average and centered average*) and **fully connected network**.
- If we put in 1 layer of hidden 5 neurons, and two outputs (for 0 and 1 digit), this is the structure one expected to construct:

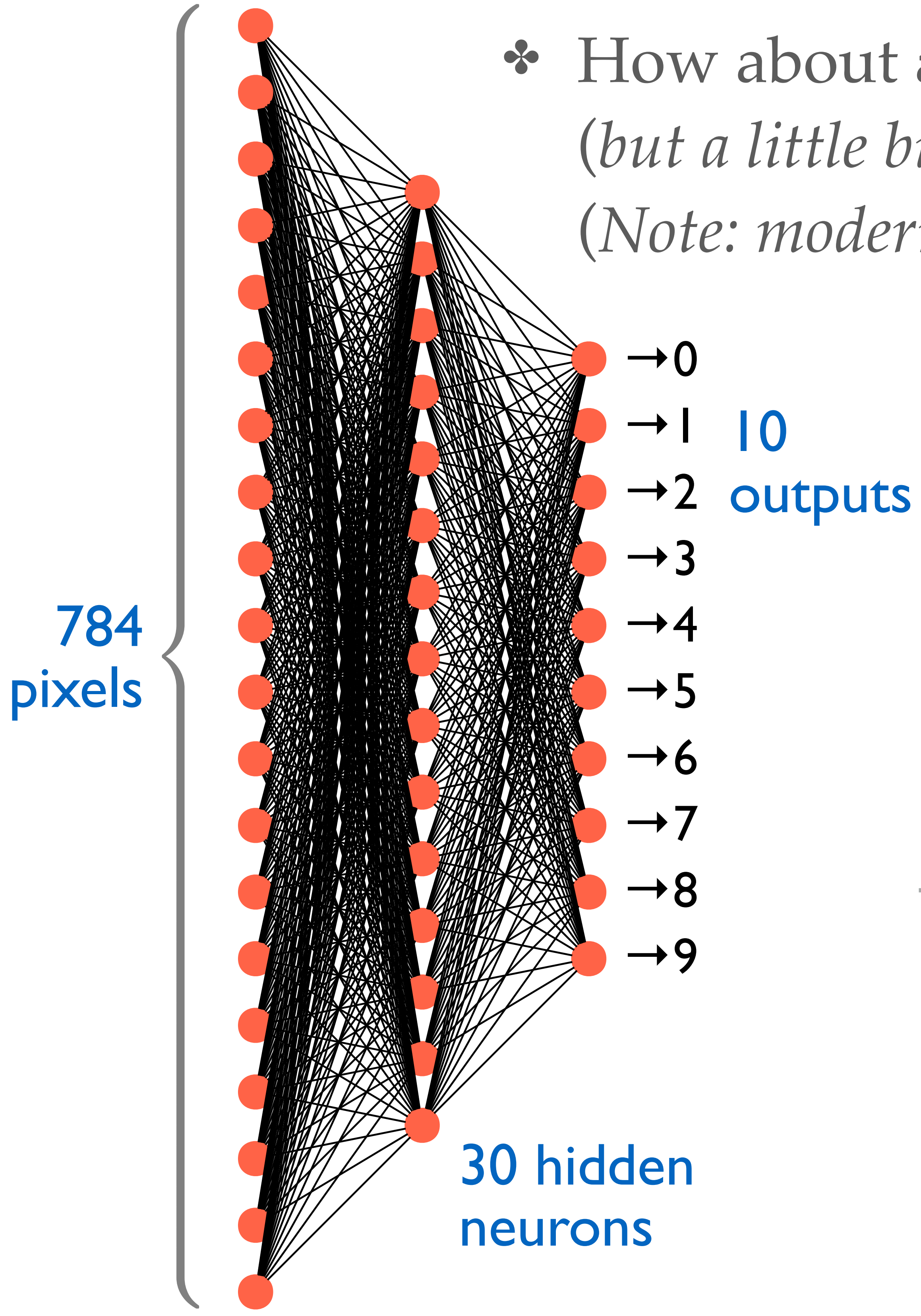


# of biases:
0 (no biases for input)+
5 (hidden neurons)+
2 (output neurons).

# of weights:
2×5 (input to hidden)+
5×2 (hidden to output).

27 parameters in total

NETWORK ARCHITECTURE (IV)



- How about another possible (*but a little bit more complex*) configuration? (*Note: modern NN structure can be very complicated!*)

# of biases:
0 (no biases for input)+
30 (hidden neurons)+
10 (output neurons).

# of weights:
784×30 (input to hidden)+
30×10 (hidden to output).

23860 parameters in total

Lots of parameters to be optimized in the training!

COMMENT: ANN STRUCTURE

- Restrictedly speaking the term multilayer perceptrons MLP are based on multilayers of sigmoid neurons (*not perceptrons*). This name was there due to some historical reason and it is confusion in fact.
- There are more different choice of activation functions. A typical alternative choice is **tanh(z)**, which is exactly the same as sigmoid function but extended to negative. **There are also a couple different choices and to be discussed later.**
- A typical example of non-feedforward network is the recurrent neural networks (RNN). The main idea in these models is to have neurons which fire for a limited duration of time, and hence it allows loops in the network.

IMPLEMENTATION START!

- Let's start our implementation of a simple feedforward network.
- Remark: my implementation is definitely not prepared for a high performance computation!

partial neurons.py

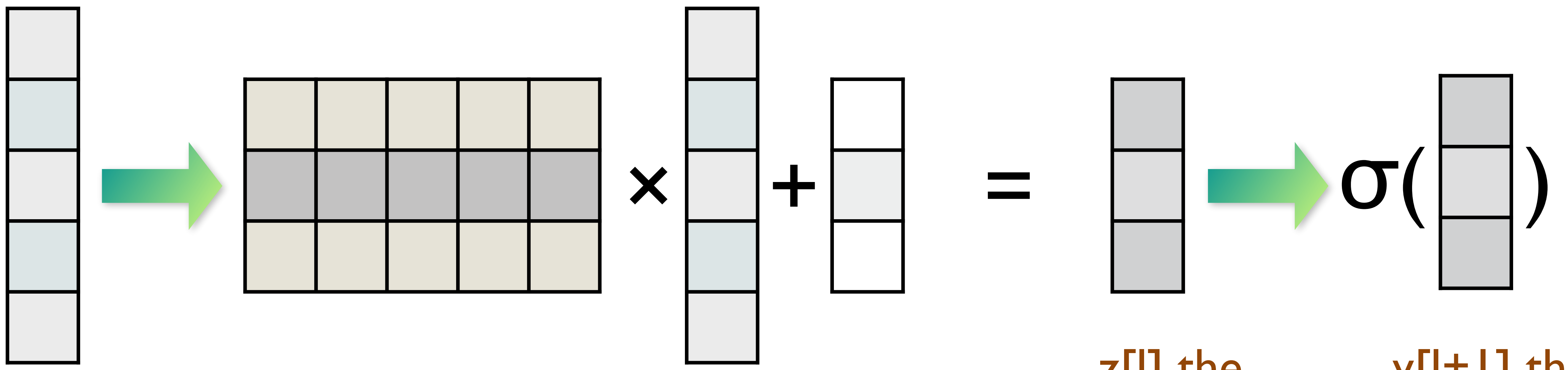
```
import numpy as np

def sigma(z):
    return 0.5*(np.tanh(0.5*z)+1.) ← activation function

class neurons(object):             constructor: expected to get a tuple or
    def __init__(self, shape): ← list of network structure, e.g. [2,5,2]
        self.shape = shape
        self.v = [np.zeros((n,1)) for n in shape]
        self.z = [np.zeros((n,1)) for n in shape[1:]]
        self.w = [np.random.randn(n,m) for n,m in zip(shape[1:],shape[:-1])]
        self.b = [np.random.randn(n,1) for n in shape[1:]]
```

↑ no bias nor weights for the first layer!

- **v** stores the values of σ(z)
- **z** stores the values of $z = \sum w_i x_i + b$
- **w** stores the weights
- **b** stores the biases

w, b are initialized as Gaussian random numbers for now!

FEEDFORWARD CALCULATION



v[l] the values at the previous layer

w[l] the weights between previous and the next layer

b[l] the biases at the next layer

z[l] the weighted sum at the next layer

v[l+1] the values at the next layer

partial neurons.py

```
def predict(self, x):
    self.v[0] = x.reshape(self.v[0].shape)
    for l in range(len(self.shape)-1):
        self.z[l] = np.dot(self.w[l],self.v[l])+self.b[l]
        self.v[l+1] = sigma(self.z[l])
    return self.v[-1]
```

THE NN OUTPUT BEFORE TRAINING…

- Let's take a look what is the output of NN before performing any training! Note the NN can be considered as a function with plenty of tunable parameters.

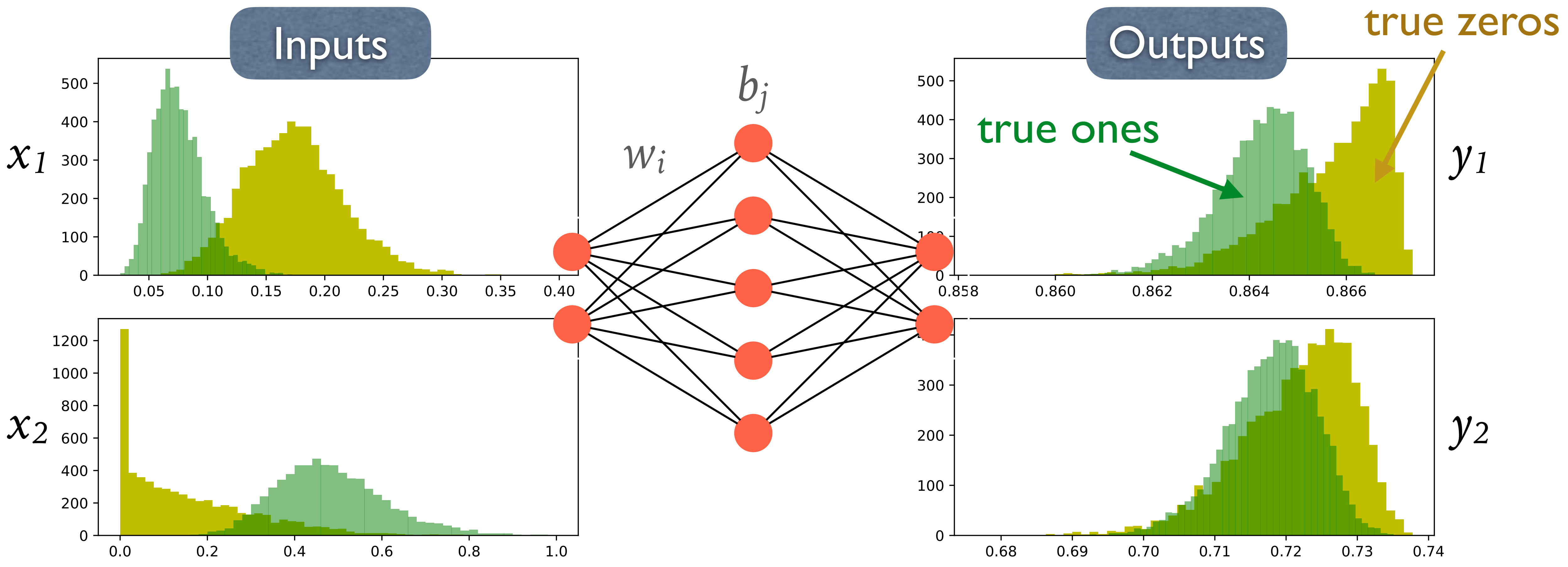
partial ex_ml4_1.py

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][mnist['y_train']<=1]/255.
y_train = mnist['y_train'][mnist['y_train']<=1]

x_train = np.array([[img.mean(),img[10:18,11:17].mean()] for img in x_train])
y_train = np.array([[[1,0],[0,1]][n] for n in y_train])

from neurons import neurons
model = neurons([2,5,2])
out = np.array([model.predict(x) for x in x_train])

fig = plt.figure(figsize=(6,6), dpi=80)
plt.subplot(2,1,1)
plt.hist(out[:,0][y_train[:,0]==1], bins=50, color='y')
plt.hist(out[:,0][y_train[:,1]==1], bins=50, color='g', alpha=0.5)
plt.subplot(2,1,2)
plt.hist(out[:,1][y_train[:,0]==1], bins=50, color='y')
plt.hist(out[:,1][y_train[:,1]==1], bins=50, color='g', alpha=0.5)
plt.show()
```

← calling the "neurons" class we just prepared

THE NN OUTPUT BEFORE TRAINING…(II)

- Before the training the network is acting like a random smearing function of the inputs.



$$(y_1, y_2) = f(x_1, x_2; w_i, b_j)$$

Now we should start to **“train”** the network to reach its maximum separation power at the outputs!

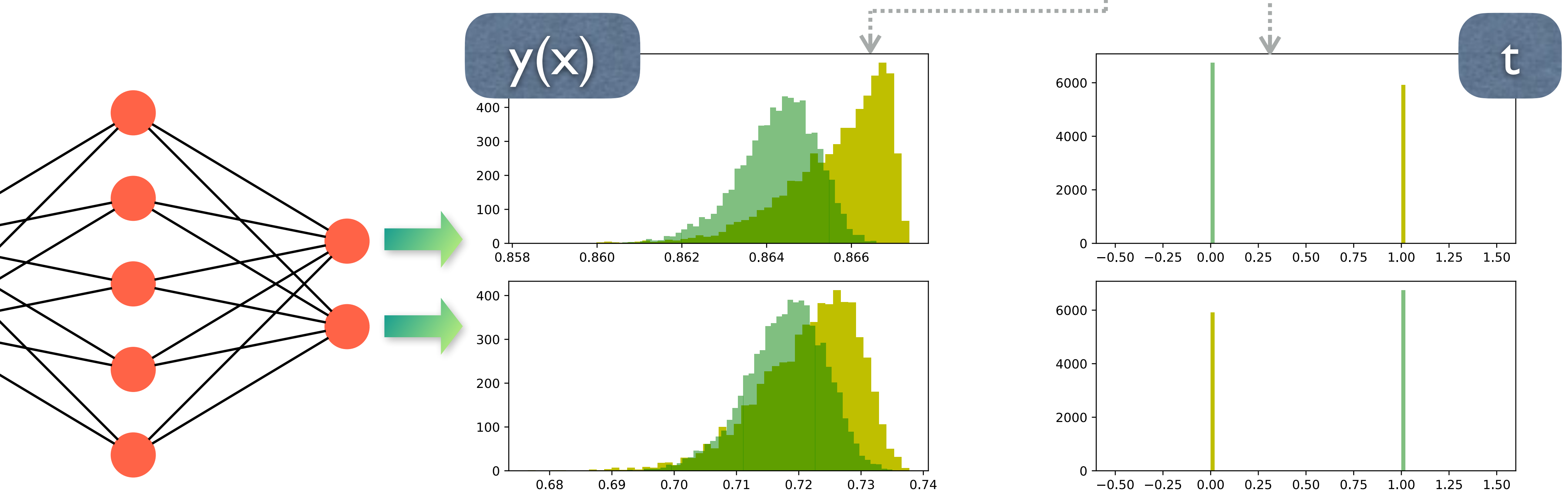
THE TRAINING GOAL

- In order to train our network, it is required to define a **loss function**, which indicates the distance between the current output and their target values.

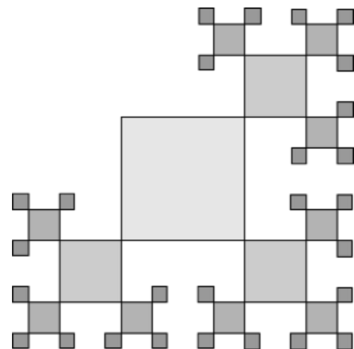
There are other choices, too!

- A typical choice can be this **mean squared error (MSE)**, it should be minimized in the training process:

$$\mathrm{Loss}(w_i, b_j) = \frac{1}{2n}\sum_{x}^{n} |y(x) - t|^2$$



True zeros	True ones
1	0
0	1



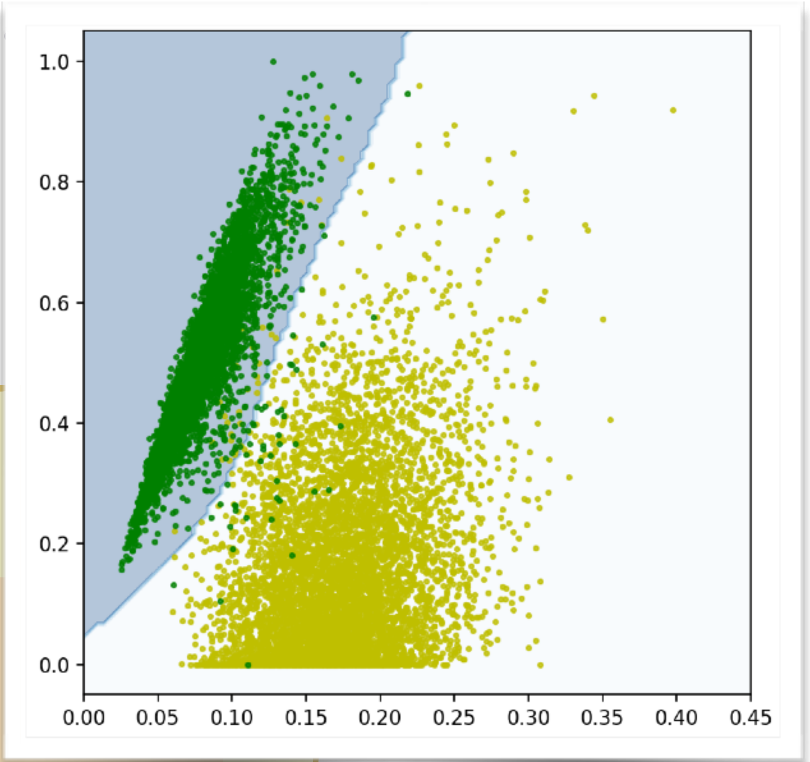
COMMENT: MULTI-DIMENSIONAL MINIMIZATION

- A feed-forward network of a structure like 2-5-2 (*as implemented in the previous example*) is nothing more than a function which has 2 inputs, 2 output, and 27 tunable parameters.
- All we need to do is varying those tunable parameters until the loss function reaches its minimal ⇒ **a multi-dimensional minimization problem**.
- Can we use the existing minimisers to do the job? It is indeed possible to do so with limited tunable parameters. But it will become totally unpractical when the # of parameters is large.
- Also the training of NN does not require to reach the global minimal — in most of the cases we just the a "good enough" solution!

How to optimize those parameters to be discussed in the next module!

COMMENT: MULTI-DIMENSIONAL MINIMIZATION (II)

- As a fun demonstration, let's just minimize the loss function by SciPy tools and see how it goes:



partial ex_ml4_2.py

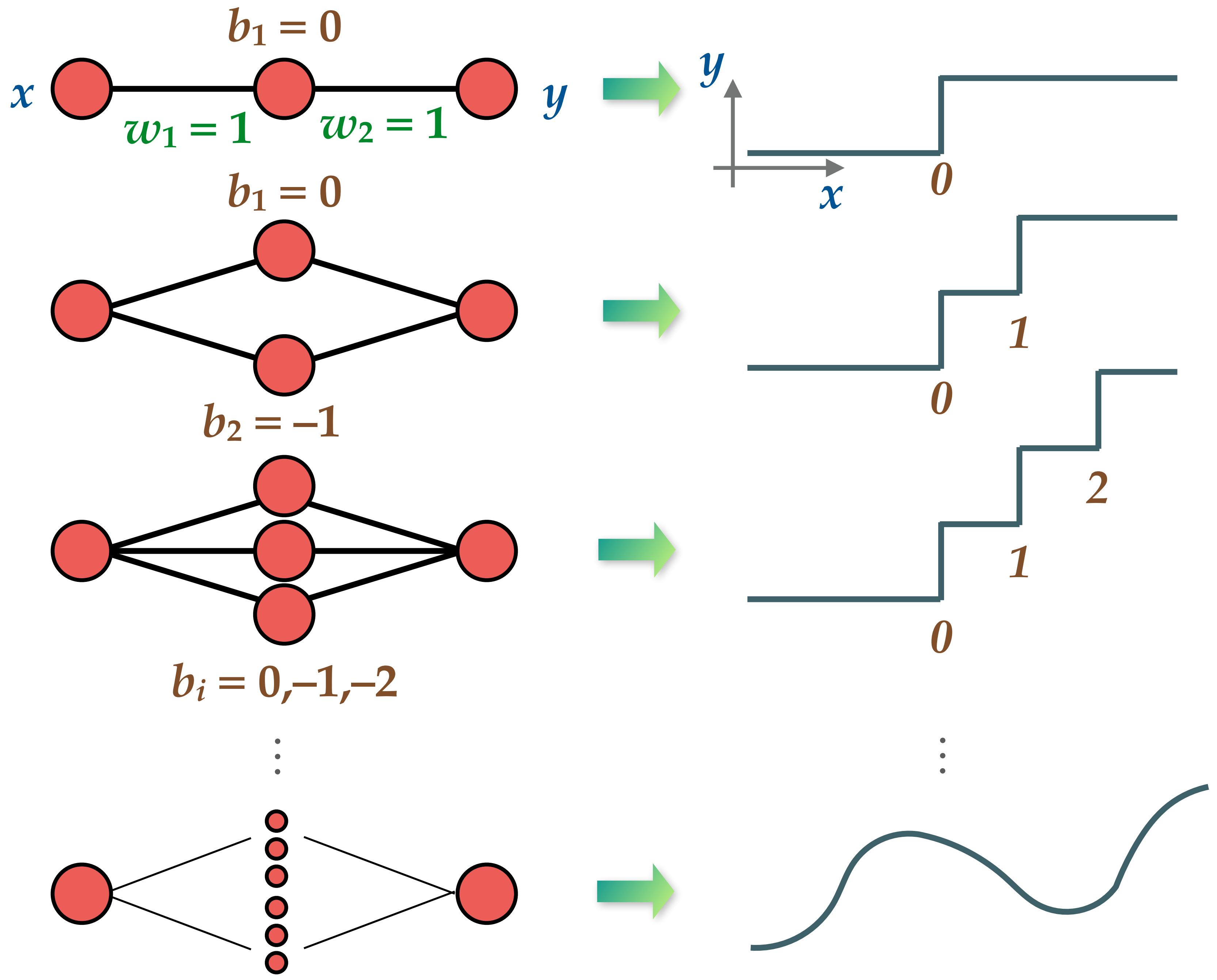
```
from neurons import neurons
model = neurons([2,5,2])

import scipy.optimize as opt
def f(x):
    model.w[0] = x[:10].reshape(model.w[0].shape)
    model.w[1] = x[10:20].reshape(model.w[1].shape)
    model.b[0] = x[20:25].reshape(model.b[0].shape)
    model.b[1] = x[25:].reshape(model.b[1].shape)
    y = np.array([model.predict(x) for x in x_train])
    return ((y[:,:,0]-y_train)**2).sum()/len(x_train)/ 2.
x_init = np.random.randn(27)
res = opt.minimize(f,x_init,options={'disp': True})

out = np.array([model.predict(x) for x in x_train])
s_train = (np.argmax(out[:,:,0],axis=1)==\
           np.argmax(y_train,axis=1)).sum()/len(y_train)
print('Performance (training):', s_train)
```

← *tuning the 27 parameters with scipy tool (slow!)*

```
Performance (training): 0.9936833793920252
```

COMMENT: NEURAL NETWORK AS A UNIVERSAL FUNCTION

- One of the arguments why the NN is so powerful — a network with enough "bandwidth" (=# *of hidden nodes*) can be used to approximate an arbitrary function.



By adding more and more hidden nodes, the network is in fact start to construct s function as sum of many step functions (or sigmoid functions). Eventually it can approximate nearly anything.

COMMENT: NEURAL NETWORK AS A UNIVERSAL FUNCTION (II)

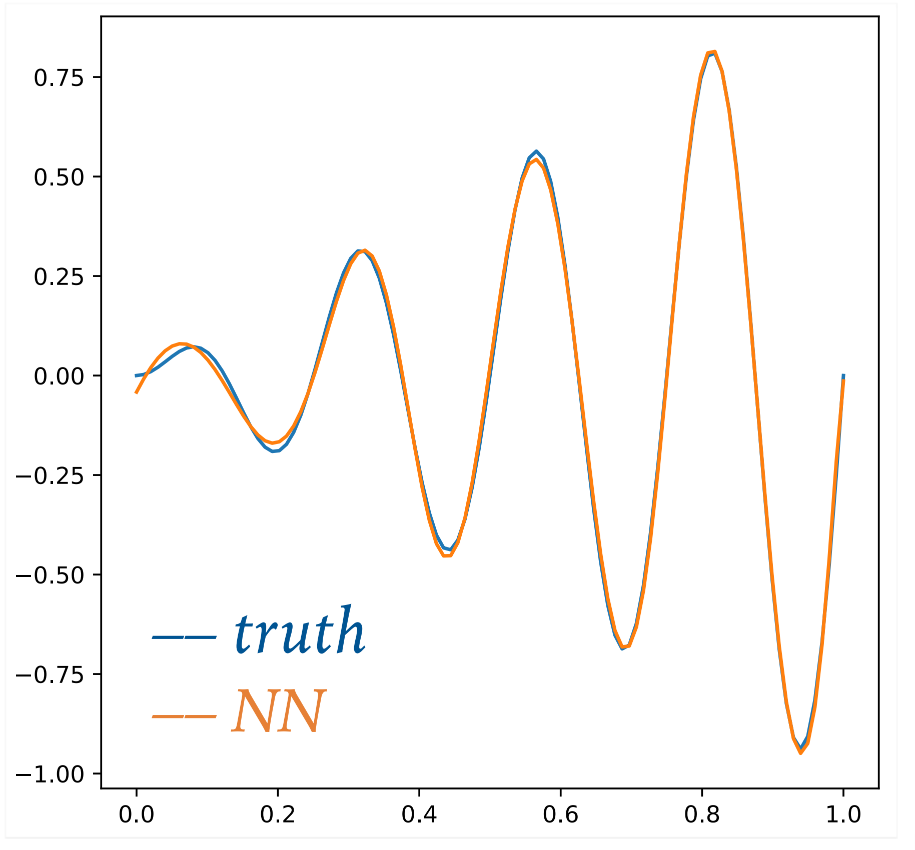
- Let's ask our simple network (with a not-yet-large 1-10-1 structure) to learn a complicated function, e.g.:

$$y = f(x) = x\sin(8\pi x)$$

partial ex_ml4_3.py

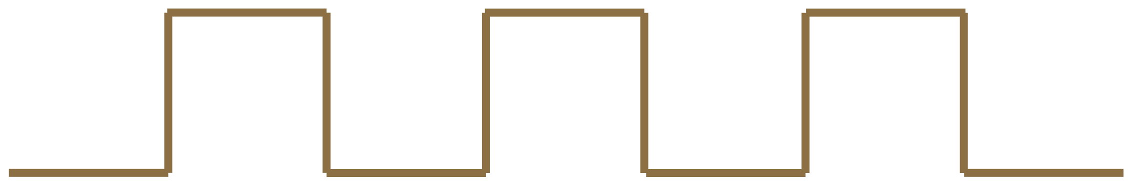
```
x_train = np.linspace(0.,1.,100)
y_train = [x*np.sin(x*np.pi*8.) \
           for x in x_train]

from neurons import neurons
model = neurons([1,10,1])

import scipy.optimize as opt
def f(x):
. . . . . . .
    y = []
    for x in x_train:
        model.predict(x)
        y.append(model.z[-1][0,0])
    y = np.array(y)
    return ((y-y_train)**2).sum()/len(x_train)/2.
x_init = np.random.randn(31)
res = opt.minimize(f,x_init,options={'disp': True})
```

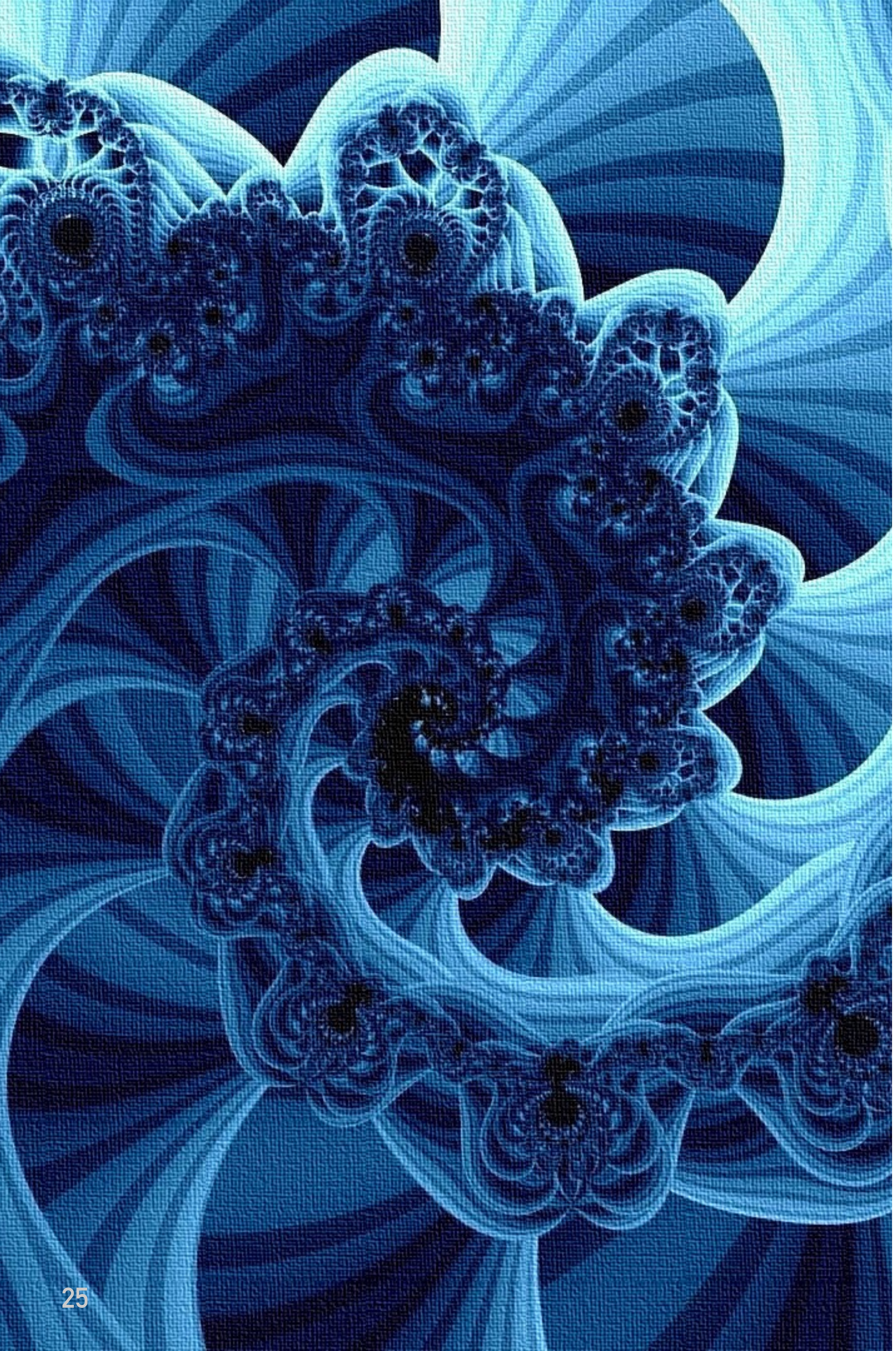
← *since this is kind of "regression" topic, it is recommended not to apply the sigmoid activation at the ending layer.*



JUST TRY IT OUT!

- We have shown that a fully connected network in the structure of 2-5-2 (784-30-10) has 27 (23860) tunable parameters. Could you construct a simple code to calculate the expected number of tunable parameters for any shape?
  - ➡ You may consider to add a method to our neurons class which can return this estimate!
- As a fun practice, are you able to construct a NN directly (assign the parameters by hand) and represent a square-wave, e.g.





MODULE SUMMARY

- In this module we have introduced the most famous model in machine learning, the artificial neural network, which is stimulated by the biological neurons. The structure of network could easily include a lot of tunable parameters.
- Next module we will continue to discuss how to effectively "train" the neural network.