

# Neural Network Architectures and Backpropagation

Fundamentals of Artificial Intelligence

Instructor: Chenhui Chu

Email: [chu@i.kyoto-u.ac.jp](mailto:chu@i.kyoto-u.ac.jp)

Teaching Assistant: Youyuan Lin

E-mail: [youyuan@nlp.ist.i.kyoto-u.ac.jp](mailto:youyuan@nlp.ist.i.kyoto-u.ac.jp)

# Schedule

- 1. Overview of AI and this Course (4/14)
- 2. Introduction to Python (4/21)
- 3, 4. Mathematics Concepts I, II (4/28, 5/12)
- 5, 6. Regression I, II (5/19, 5/26)
- 7. Classification (6/2)
- 8. Introduction to Neural Networks (6/9)
- **9. Neural Networks Architecture and Backpropagation (6/16)**
- 10. Fully Connected Layers (6/23)
- 11, 12, 13. Computer Vision I, II, III (6/30, 7/7, 7/14)
- 14. Natural Language Processing (7/17)

# Overview of This Course

11, 12, 13. Computer Vision  
I, II, III

14. Natural language  
processing

## Deep Learning Applications

8. Neural network  
Introduction

9. Architecture and  
Backpropagation

10. Feedforward  
neural networks

## Deep Learning

5. Regression I

6. Regression II

7. Classification

## Basic Supervised Machine Learning

2. Python

3, 4. Mathematics Concepts I, II

## Fundamental of Machine Learning

# Neural Network Architectures and Backpropagation

- What we are going to discuss today:
  - An overview of Neural Network Architectures
  - The backpropagation algorithm that allows us to compute the gradient in neural network and apply Gradient Descent to learn parameters

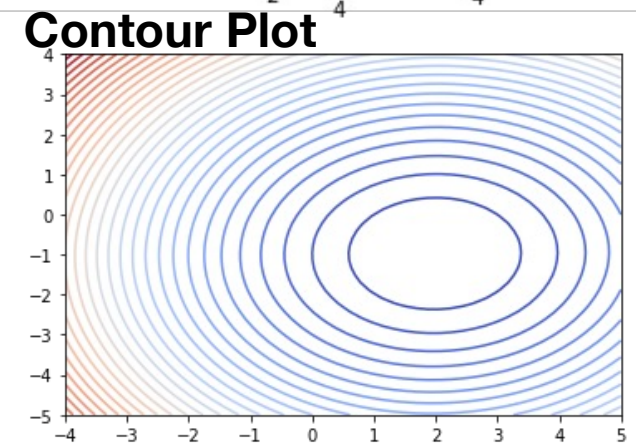
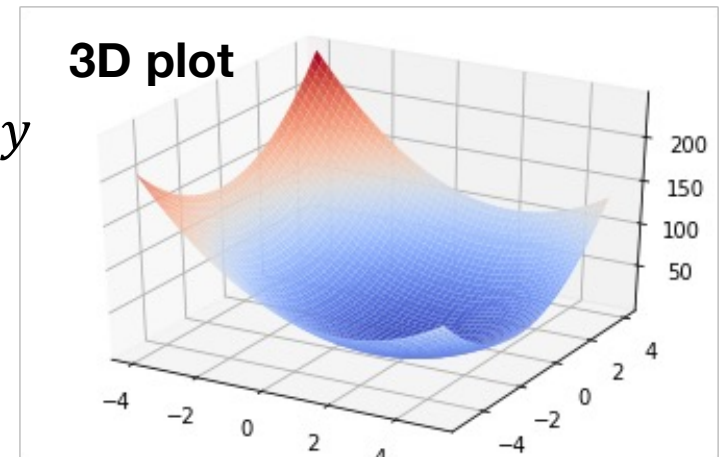
# Previously, in This Class

- Let us recap what we have seen so far

# Minimizing a Function of Several Variables

$$f(x, y) = 4(x - 2)^2 + 4(y + 1)^2 - 0.1xy$$

- We have seen that, given a function of several variables, we could find its minimum by gradient descent

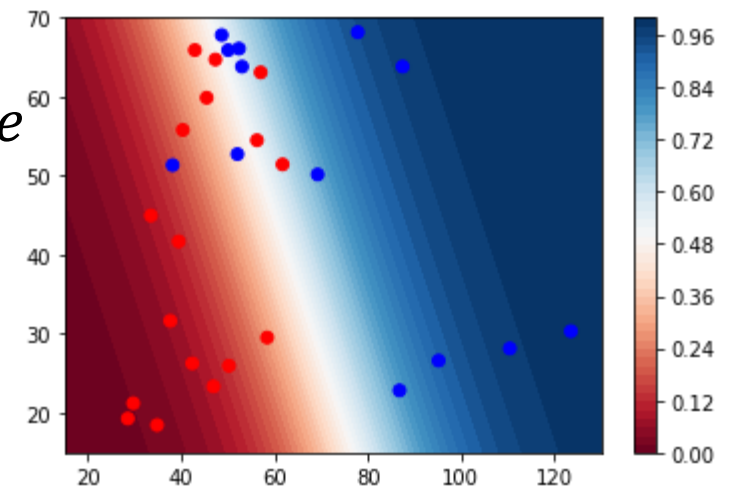


# We Have Seen

- We have seen that we can learn to predict classes with a simple parameterized function called a **logistic classifier**

$$\text{score}(\text{income}, \text{age}) = \theta_0 + \theta_1 \times \text{income} + \theta_2 \times \text{age}$$

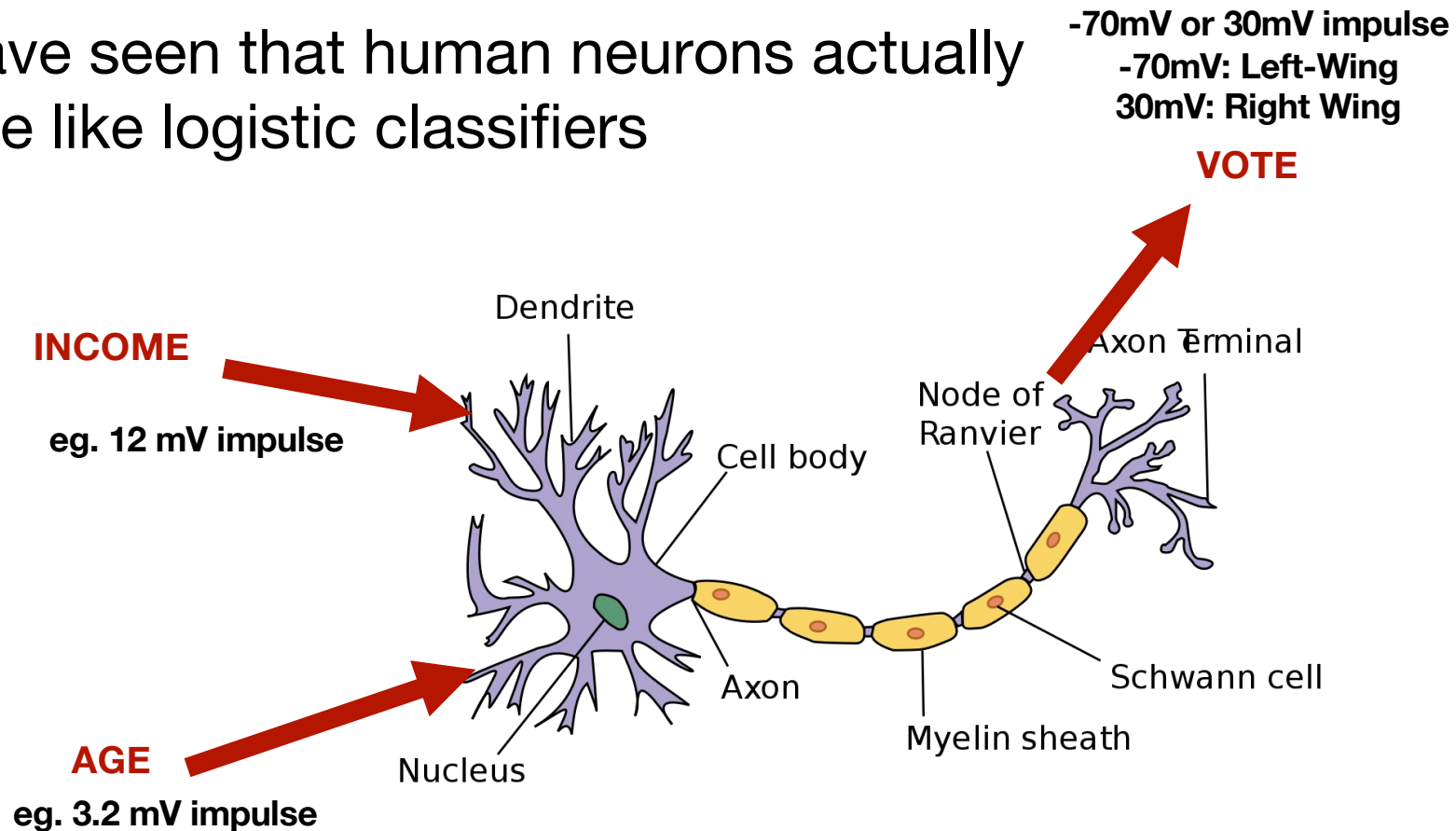
$$V_{\text{model}} = \sigma(\text{score})$$



- And that we can learn the proper parameters by applying gradient descent on the loss given some examples

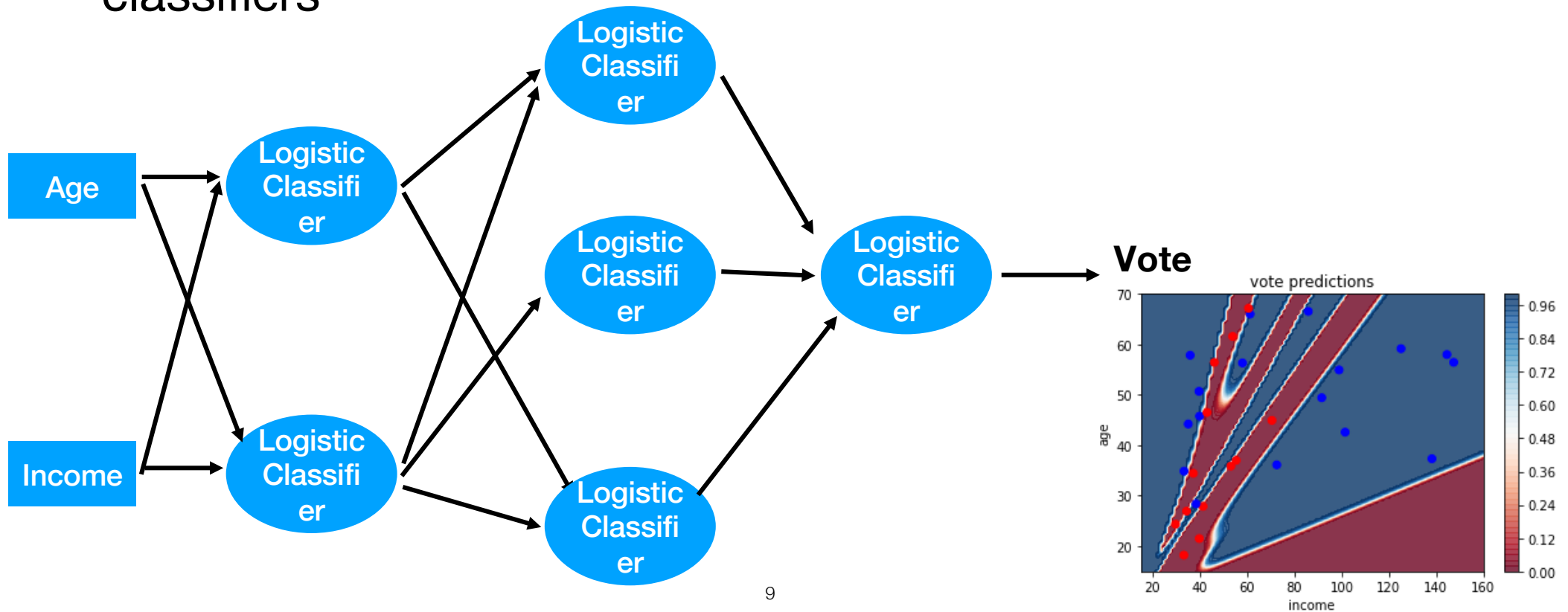
# Previously

- We have seen that human neurons actually behave like logistic classifiers



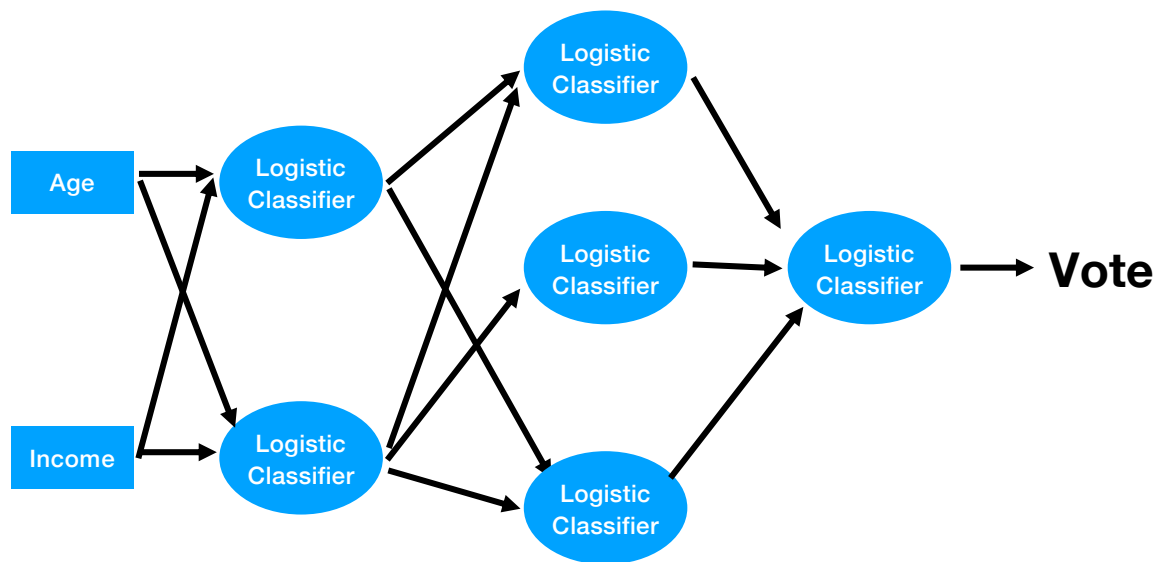
# Previously

- We have seen that we can obtain more powerful classifiers by combining the neuron-like logistic classifiers

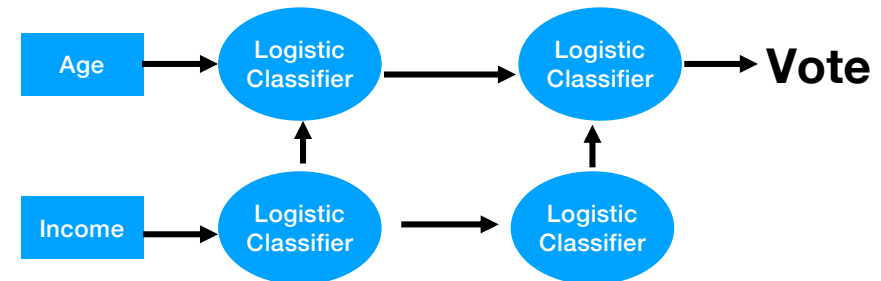


# Neural Network Architectures (1/2)

- We have seen that we could connect neurons to get more powerful classifiers
- How do we design the connections in practice ?
- —> Neural Networks Architecture

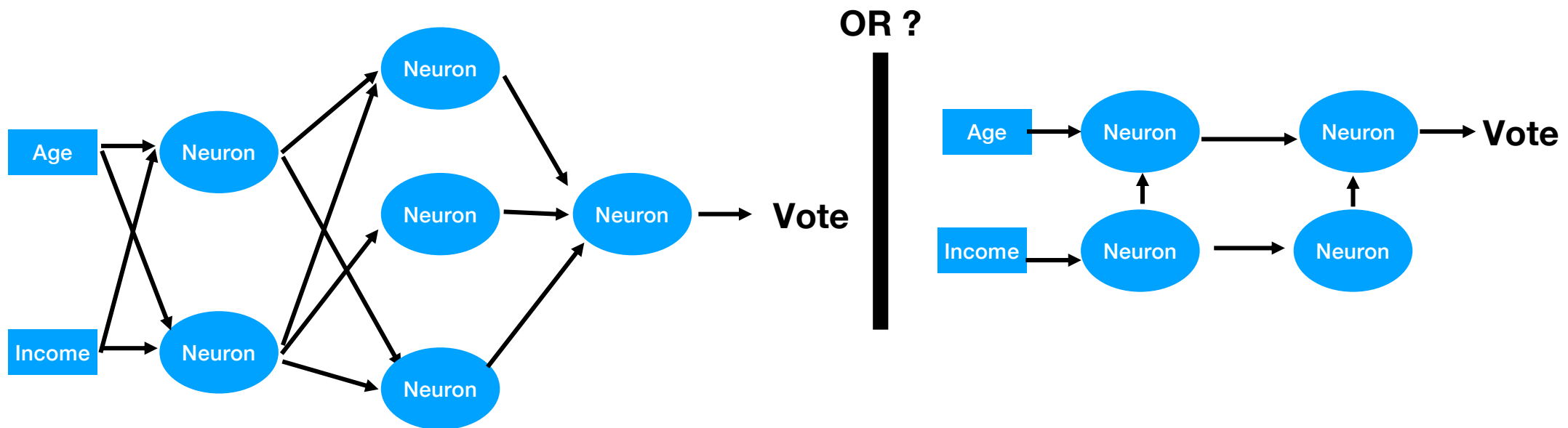


OR ?



# Neural Network Architectures (2/2)

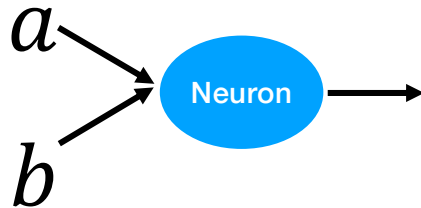
- We have seen that we could connect neurons to get more powerful classifiers
- How do we design the connections in practice ?
- —> Neural Networks Architecture



# Reminder: Neuron

- Remember, what we mean by Neuron:

$$\text{Neuron}_{\theta}(a, b) = \text{activation}(\theta_0 + a \times \theta_1 + b \times \theta_2)$$



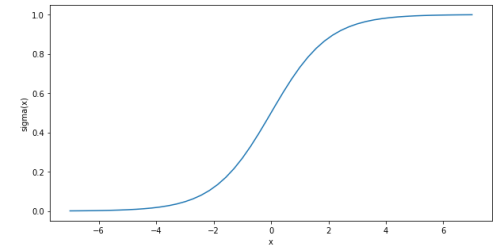
(case of a Neuron  
with 2 inputs)

Terminology note: The “*activation function*” is also sometimes called the “*non-linearity*”

Activation  
function can  
be

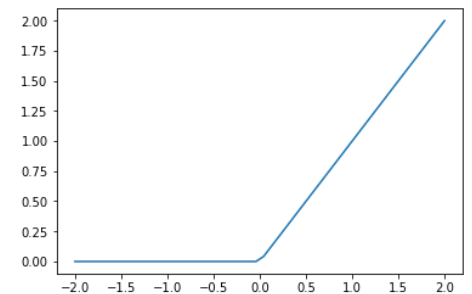
Many other possibilities ...

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Then a Neuron has the same equation  
as a Logistic classifier. It is also a better  
approximation of biological Neurons.

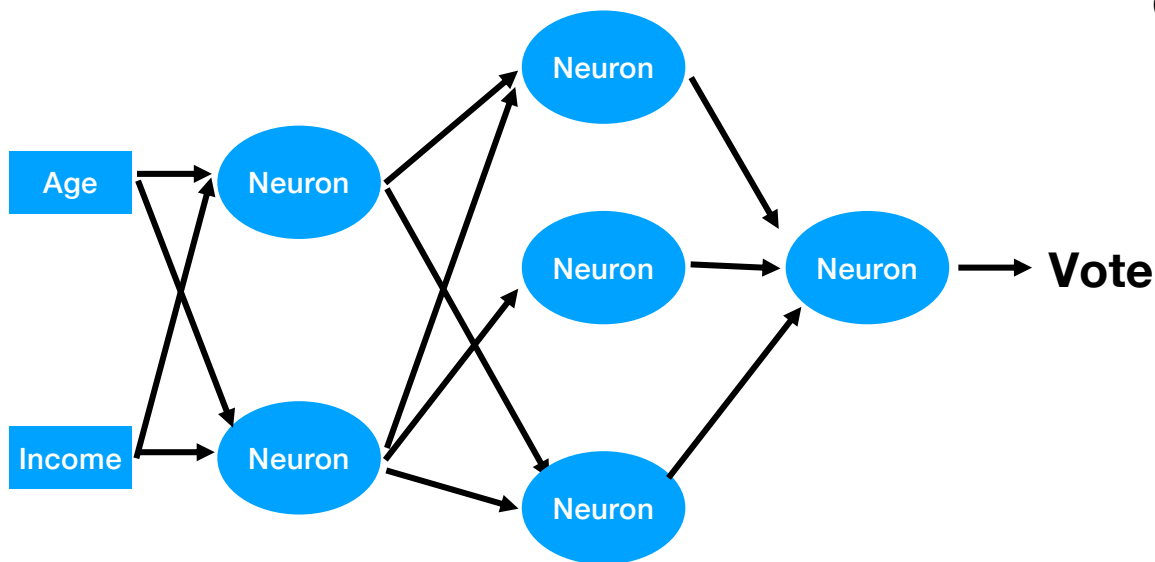
$$\text{ReLU}(x) = \max(x, 0)$$



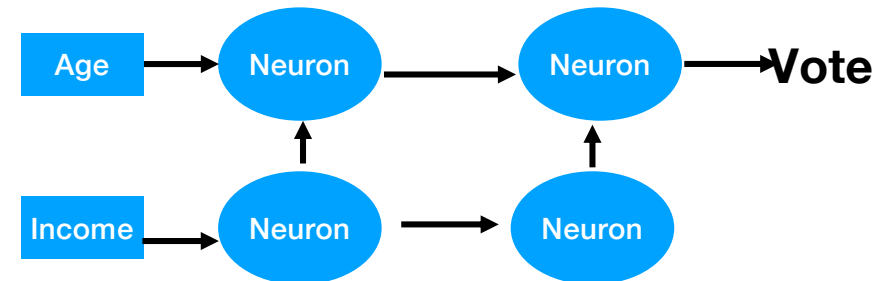
Often more efficient than sigmoid.

# Neural Network Architectures

- We have seen that we could connect neurons to get more powerful classifiers
- How do we design the connections in practice ?
- —> Neural Networks Architecture



OR ?



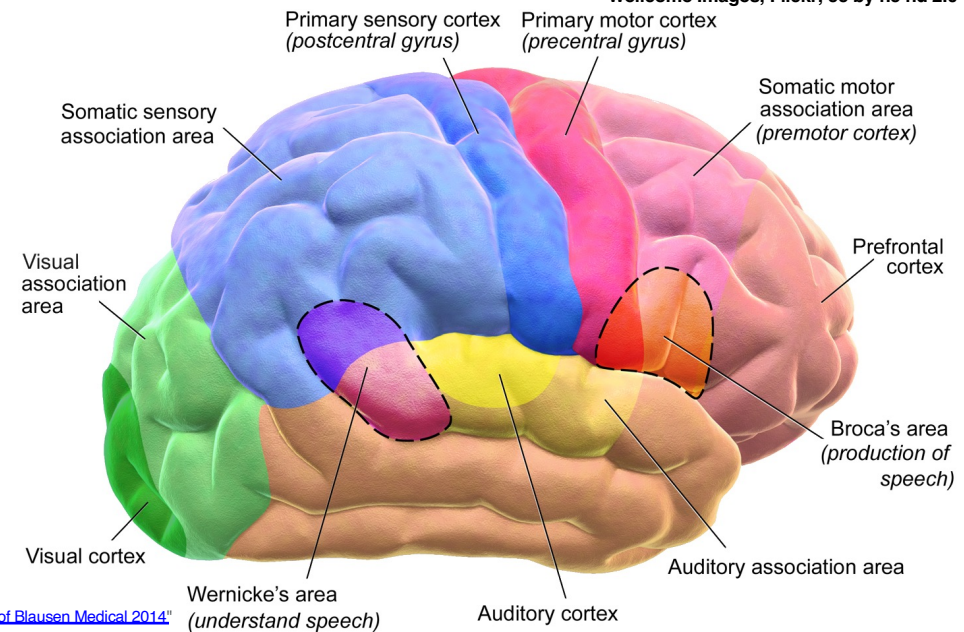
**Remember, the connection (arrow), just mean that the output of a Neuron is used as input to the next one.**

# Quick Biological Analogy

- In our brain too, Neurons are organized in complex elaborated ways
- Remember that an average neuron can connect to 7,000 other neurons
- It seems the organization of the neuron in a zone of the brain will depend on what this zone of the brain is processing
- Neural Network Architecture matters!



Wellcome Images, Flickr, cc by nc nd 2.0

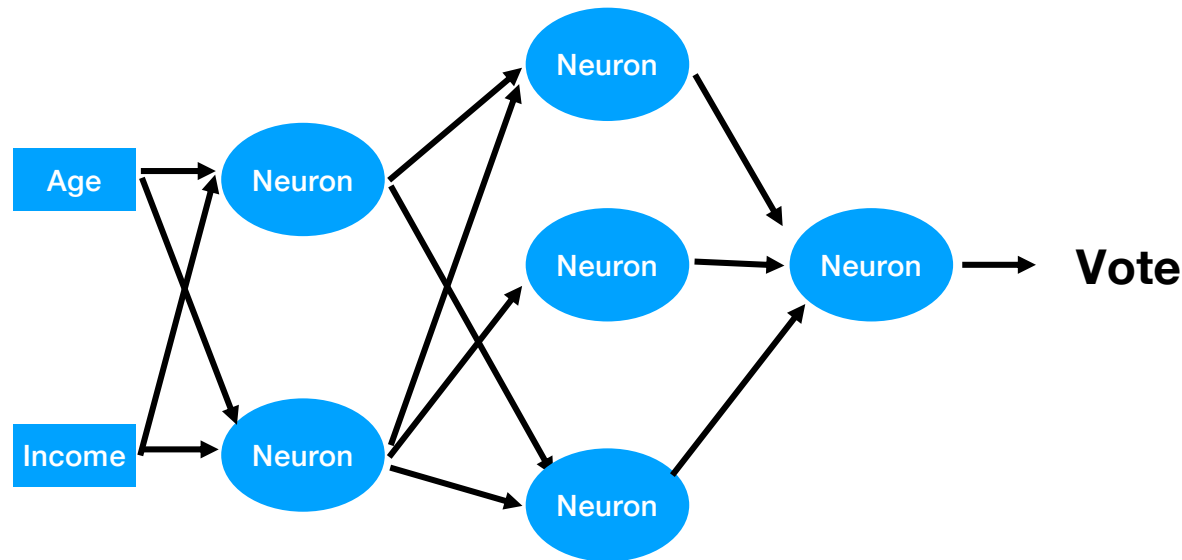


# Overview of Neural Network Architectures

- First, we will distinguish **two** broad categories of architecture:
  - **Feed-Forward Architectures**
  - **Recurrent Architectures**

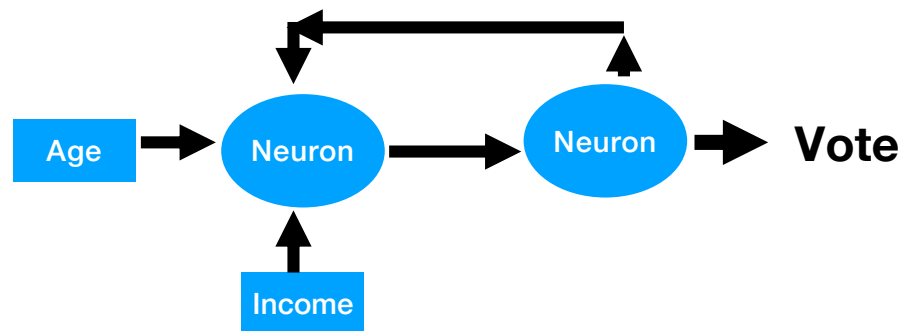
# Feed Forward Architectures

- In a **Feed Forward Architecture**, the “flow” of computation always goes forward



# Recurrent Architectures

- On the other hand, in a **Recurrent Architecture**, the output of a neuron can flow back to a previous neuron



# Overview of Neural Network Architectures

- First, we will distinguish **two** broad categories of architecture:
  - **Feed-Forward Architectures**
    - Used for image processing or general classification
  - **Recurrent Architectures**
    - Used for processing sequences (especially text)

# Overview of Neural Network Architectures

- First, we will distinguish **two** broad categories of architecture:

- **Feed-Forward Architectures**

- Used for image processing or general classification

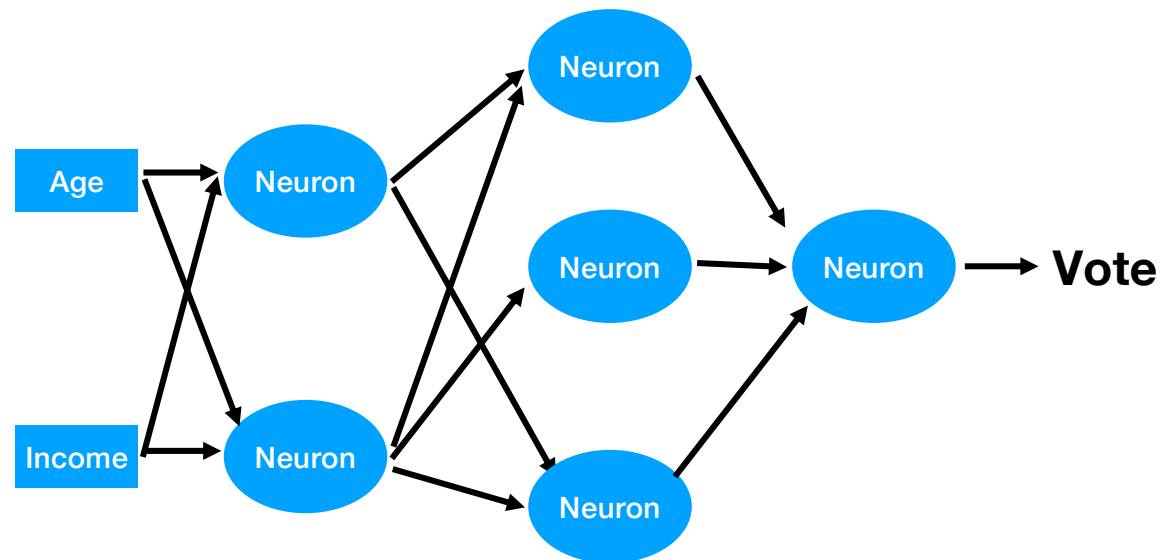
- **Recurrent Architectures**

- Used for processing sequences (especially text)

**Today and  
next 4  
lectures**

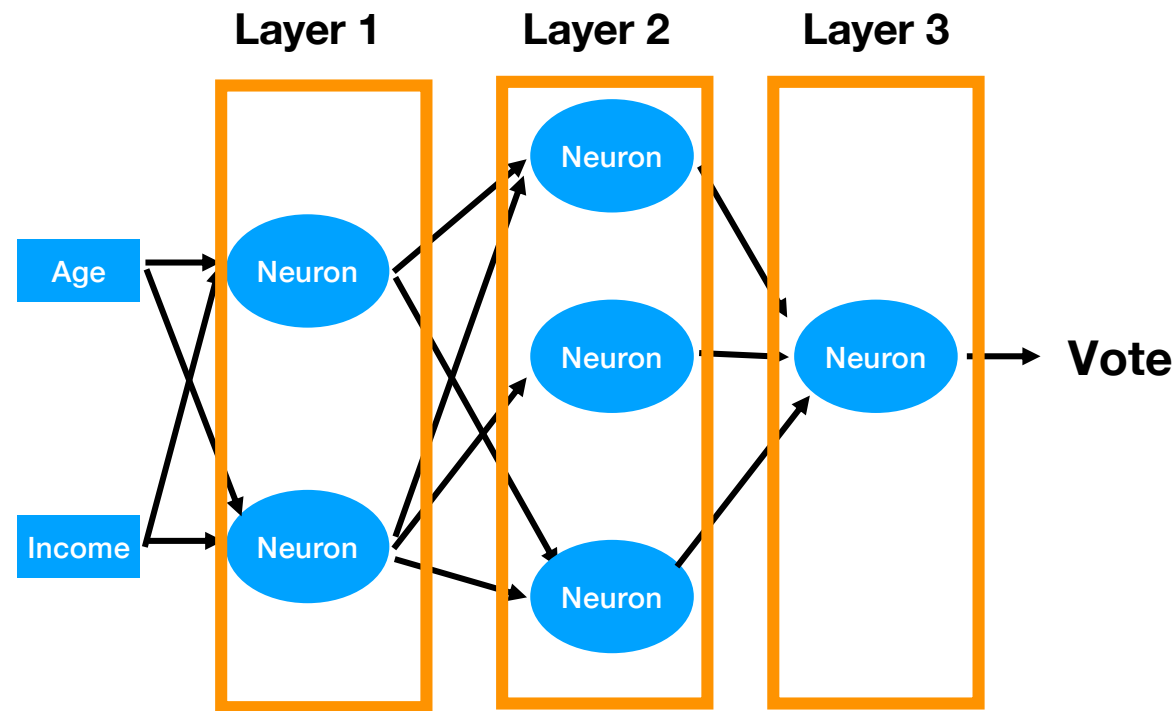
# Feed Forward Architectures (1/3)

- In the case of a feed-forward architecture, we often organize neurons in *layers*



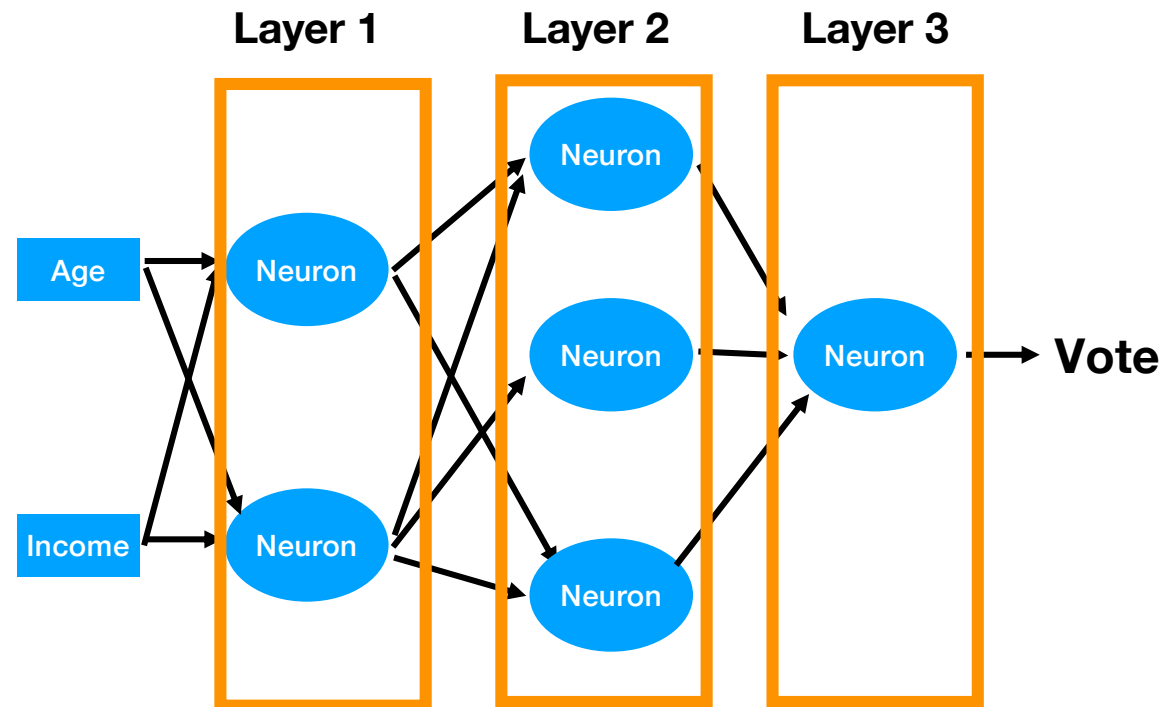
# Feed Forward Architectures (2/3)

- In the case of a feed-forward architecture, we often organize neurons in *layers*



# Feed Forward Architectures (3/3)

- In the case of a feed-forward architecture, we often organize neurons in **layers**
- Rules:
  1. A neuron is never connected to a neuron in the same layer
  2. A neuron output only goes in the input of a neuron in the next layer
- We will call this a Feed Forward Multi-Layer Architecture

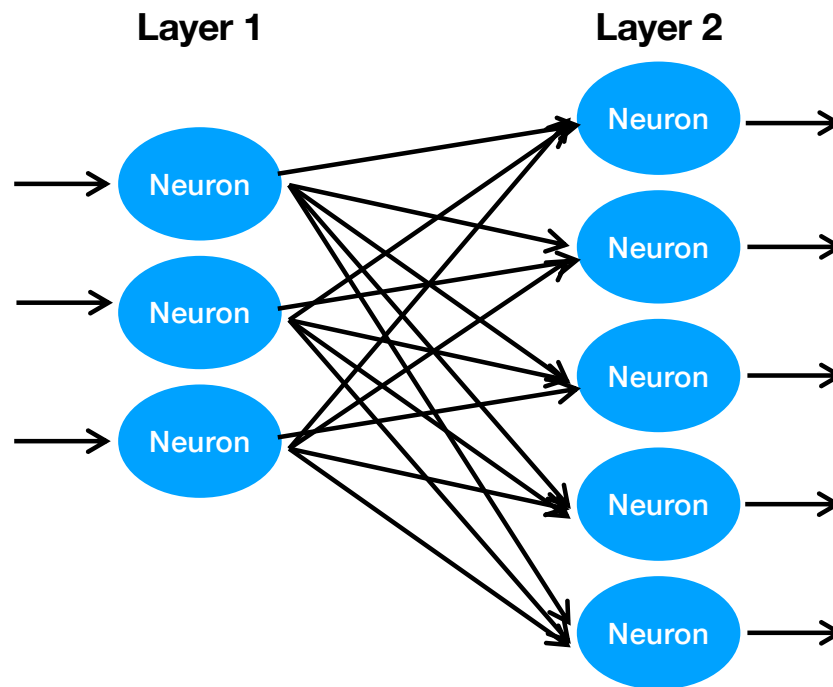


# Type of Feed-Forward Layers

- We will consider **two** types of Feed-Forward Layers:
  - Fully Connected Layers
  - Convolutional Layers

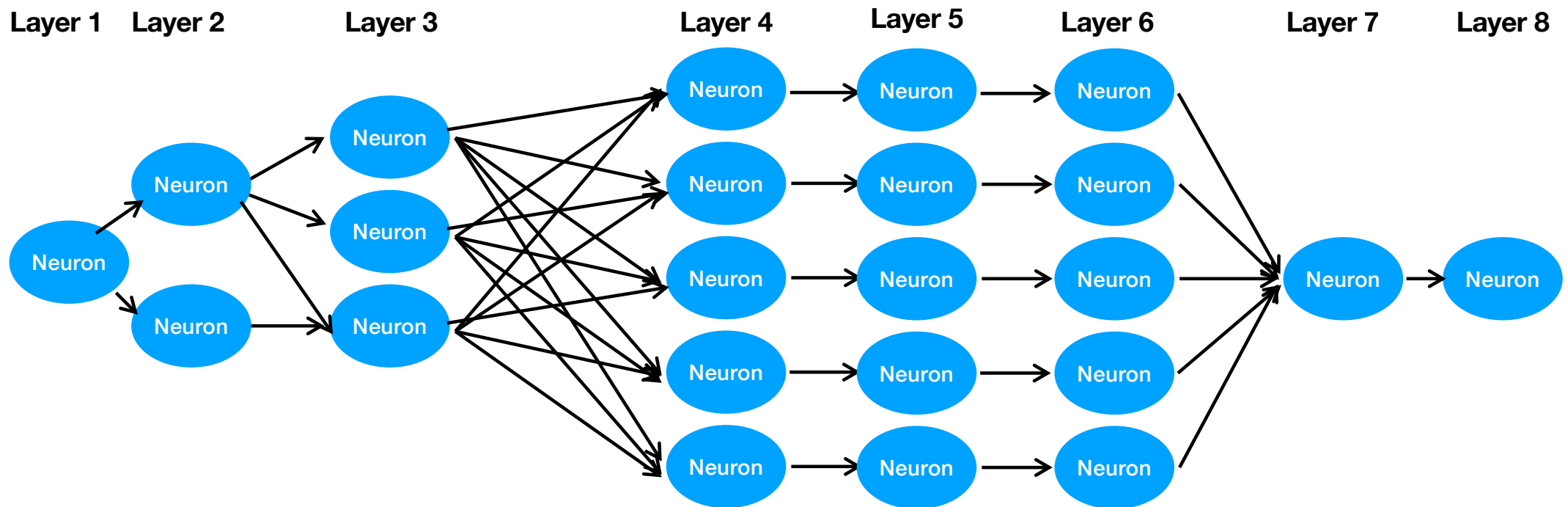
# Fully Connected Layers

- We call a layer “*Fully Connected*” if **EACH** neuron in the layer is connected to **ALL** neurons in the **previous** layer



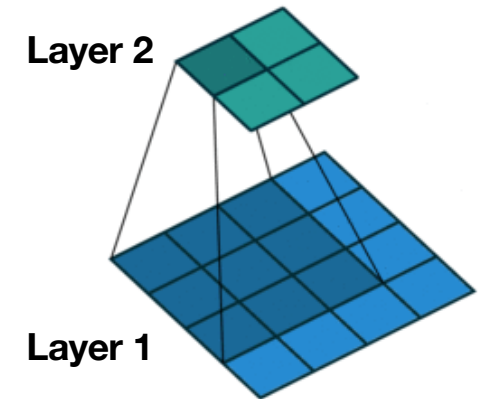
# Quiz

- Which Layers are fully connected?



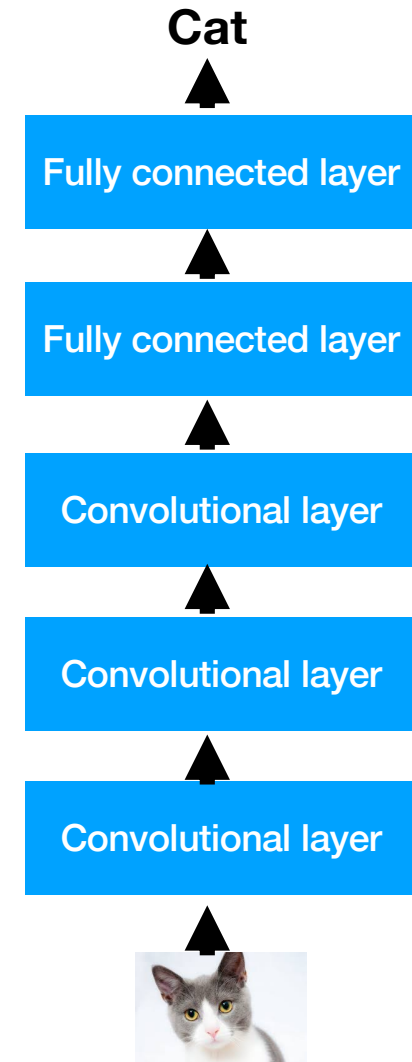
# Convolutional Layers

- Among the layers that are **NOT** fully connected, there is a special type of layer called ***Convolutional layer***
  - Very used for processing images
- Neurons are organized in 2-dimensional layers
- Neurons in 2 layers are only connected if they roughly belong to the same area of their respective layer
  - E.g. The neuron in the top-left corner of layer 2 is only connected to the 9 neurons in the top-left corner of layer 1



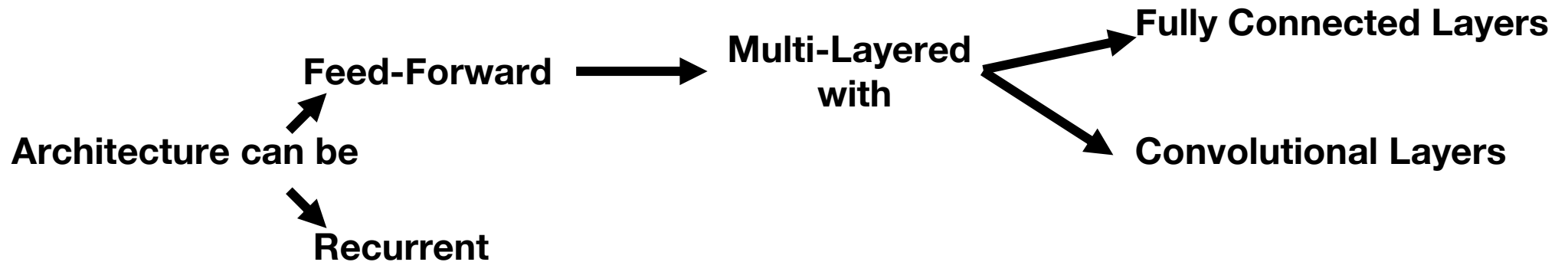
# Mix of Layers

- A typical Neural Network for Image classification will include many ***convolutional layers*** followed by a few ***fully connected layers***



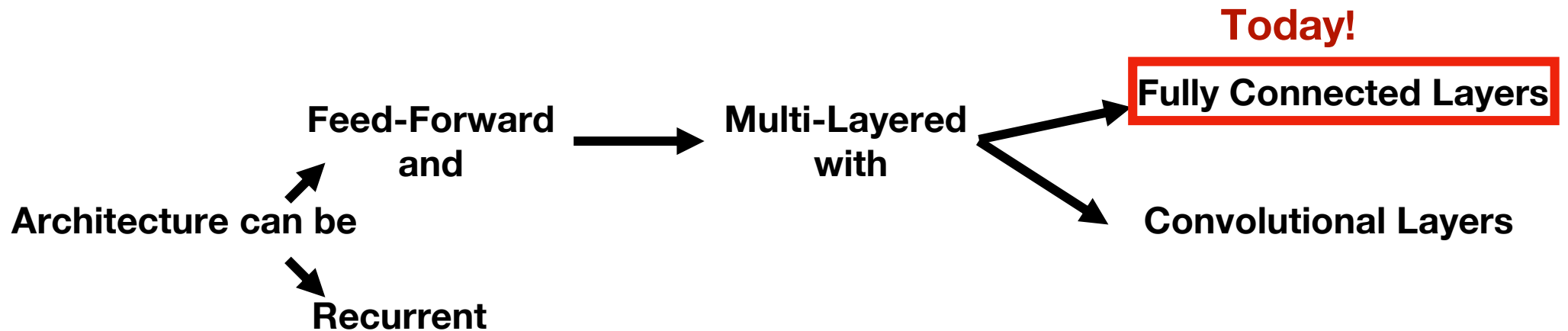
# Neural Network Architectures

- In short:



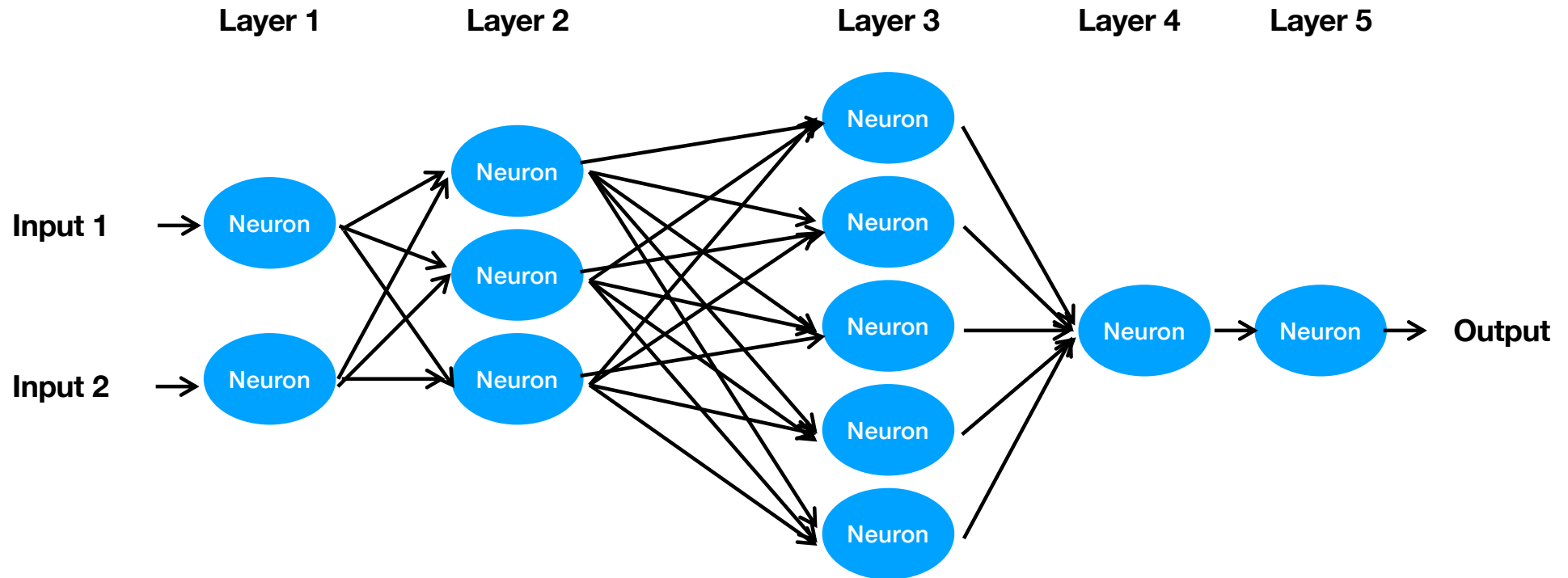
# Neural Network Architectures

- In short:



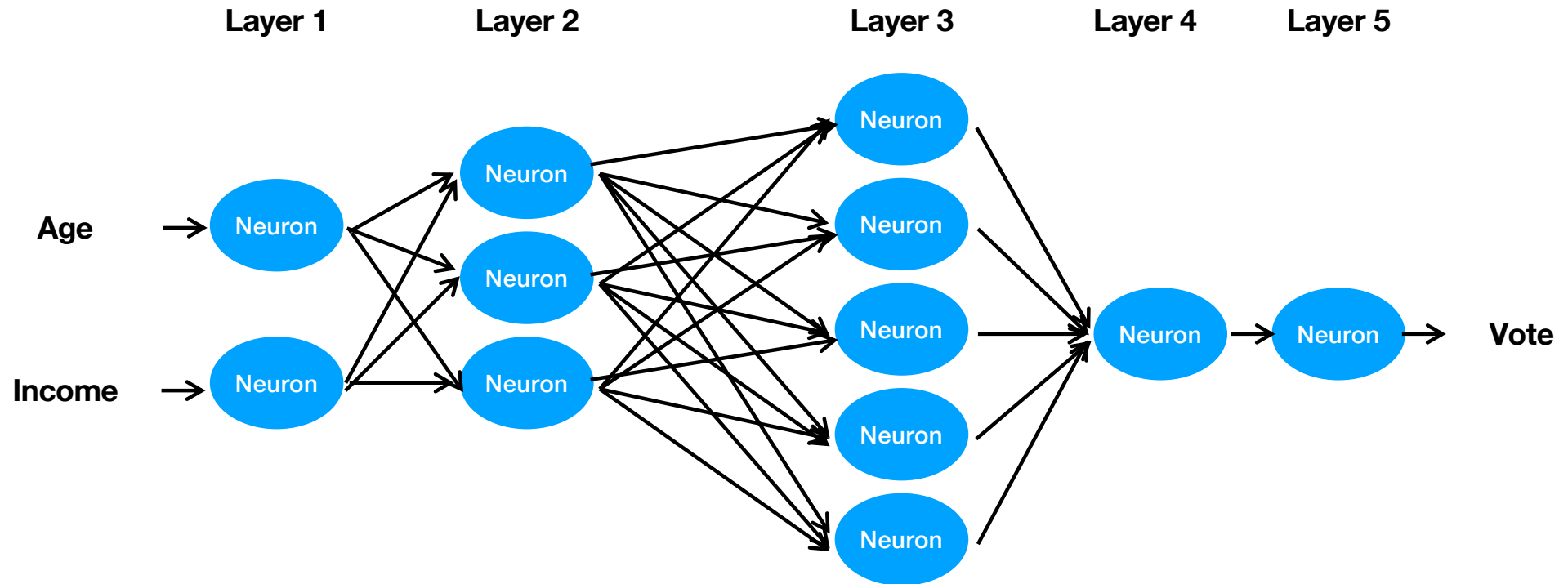
# Feed-Forward Networks with Fully Connected Layers (1/2)

- Therefore, we are going to consider this type of Neural Network:



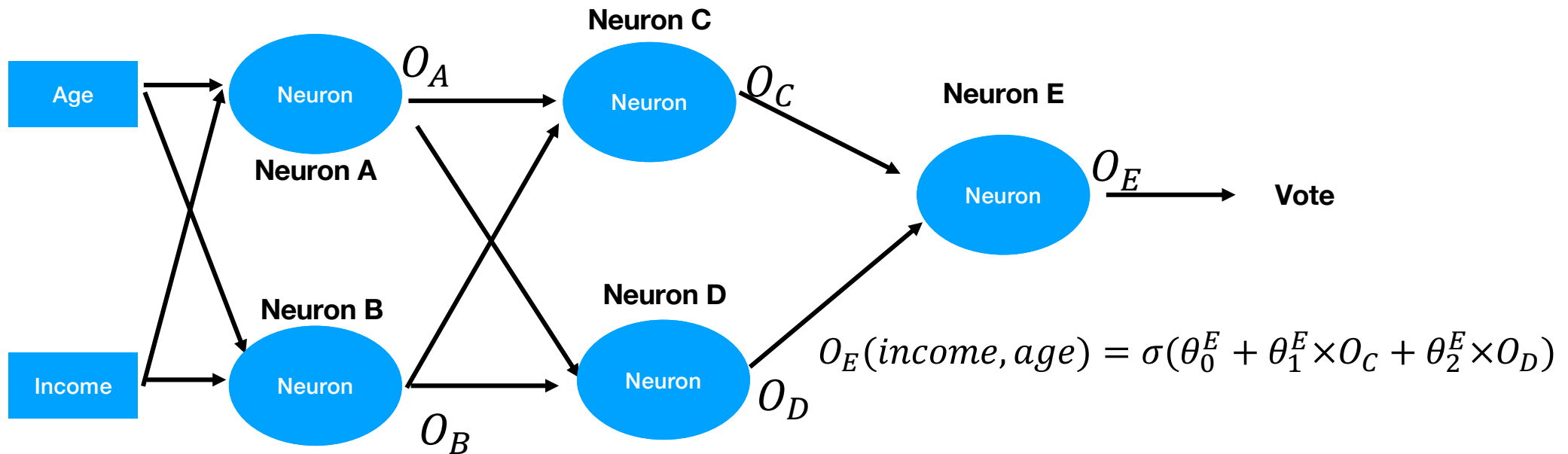
# Feed-Forward Networks with Fully Connected Layers (2/2)

- Therefore, we are going to consider this type of Neural Network:



# Keeping in Mind What This Type of Graph Mean (1/3)

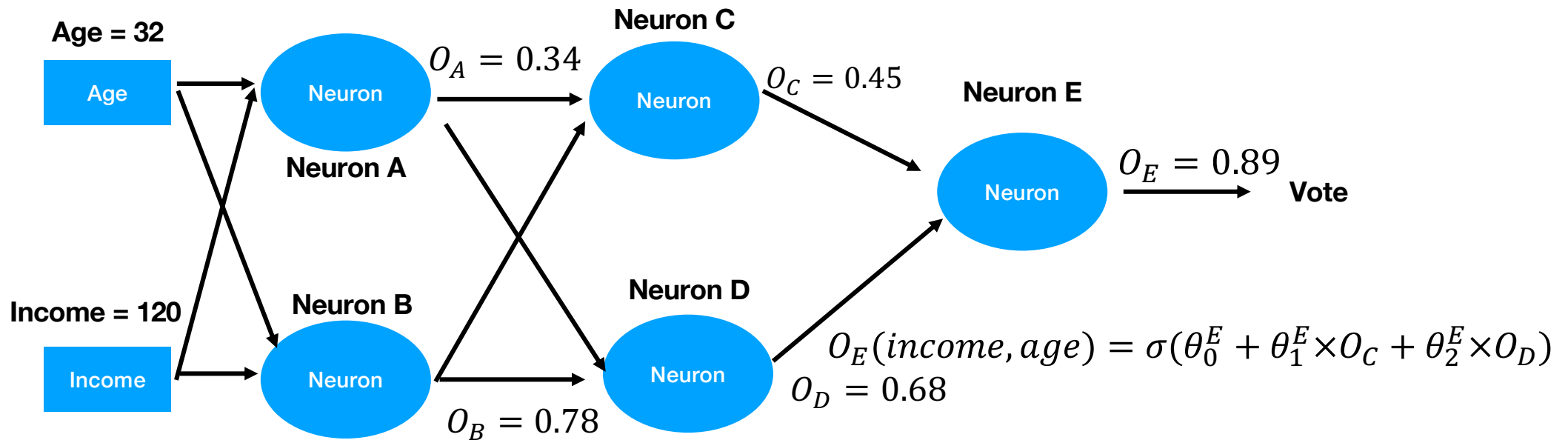
$$O_A(\text{income}, \text{age}) = \sigma(\theta_0^A + \theta_1^A \times \text{income} + \theta_2^A \times \text{age}) \quad O_C(\text{income}, \text{age}) = \sigma(\theta_0^C + \theta_1^C \times O_A + \theta_2^C \times O_B)$$



$$O_B(\text{income}, \text{age}) = \sigma(\theta_0^B + \theta_1^B \times \text{income} + \theta_2^B \times \text{age}) \quad O_D(\text{income}, \text{age}) = \sigma(\theta_0^D + \theta_1^D \times O_A + \theta_2^D \times O_B)$$

# Keeping in Mind What This Type of Graph Mean (2/3)

$$O_A(\text{income}, \text{age}) = \sigma(\theta_0^A + \theta_1^A \times \text{income} + \theta_2^A \times \text{age}) \quad O_C(\text{income}, \text{age}) = \sigma(\theta_0^C + \theta_1^C \times O_A + \theta_2^C \times O_B)$$

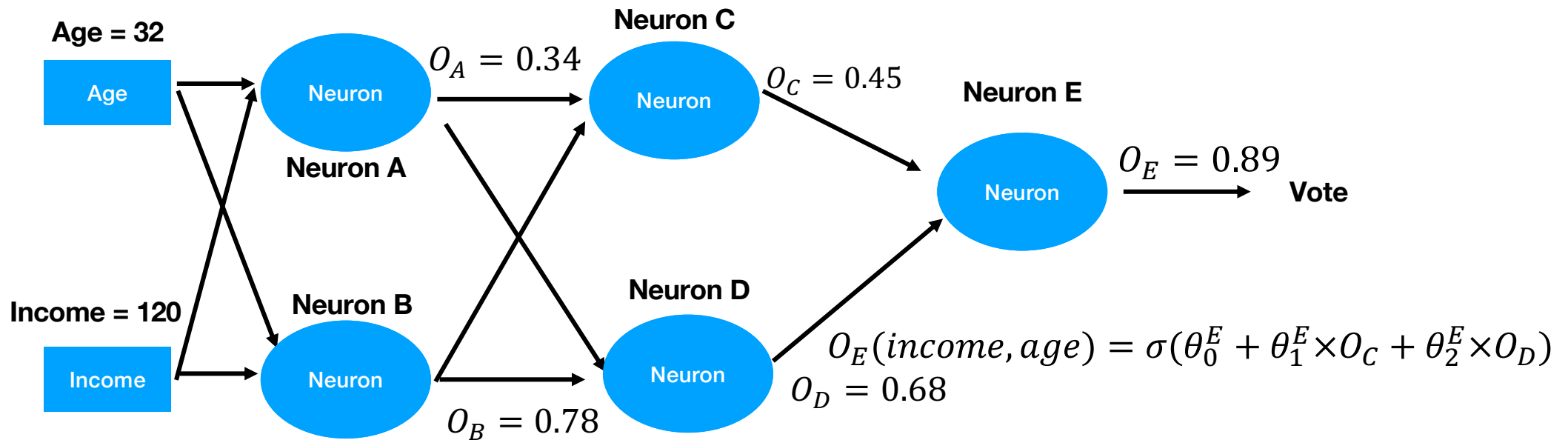


$$O_B(\text{income}, \text{age}) = \sigma(\theta_0^B + \theta_1^B \times \text{income} + \theta_2^B \times \text{age}) \quad O_D(\text{income}, \text{age}) = \sigma(\theta_0^D + \theta_1^D \times O_A + \theta_2^D \times O_B)$$

# Keeping in Mind What This Type of Graph Mean (3/3)

→ Each Neural Network architecture defines a function of the input with parameters  $\theta$

$$O_A(\text{income}, \text{age}) = \sigma(\theta_0^A + \theta_1^A \times \text{income} + \theta_2^A \times \text{age}) \quad O_C(\text{income}, \text{age}) = \sigma(\theta_0^C + \theta_1^C \times O_A + \theta_2^C \times O_B)$$

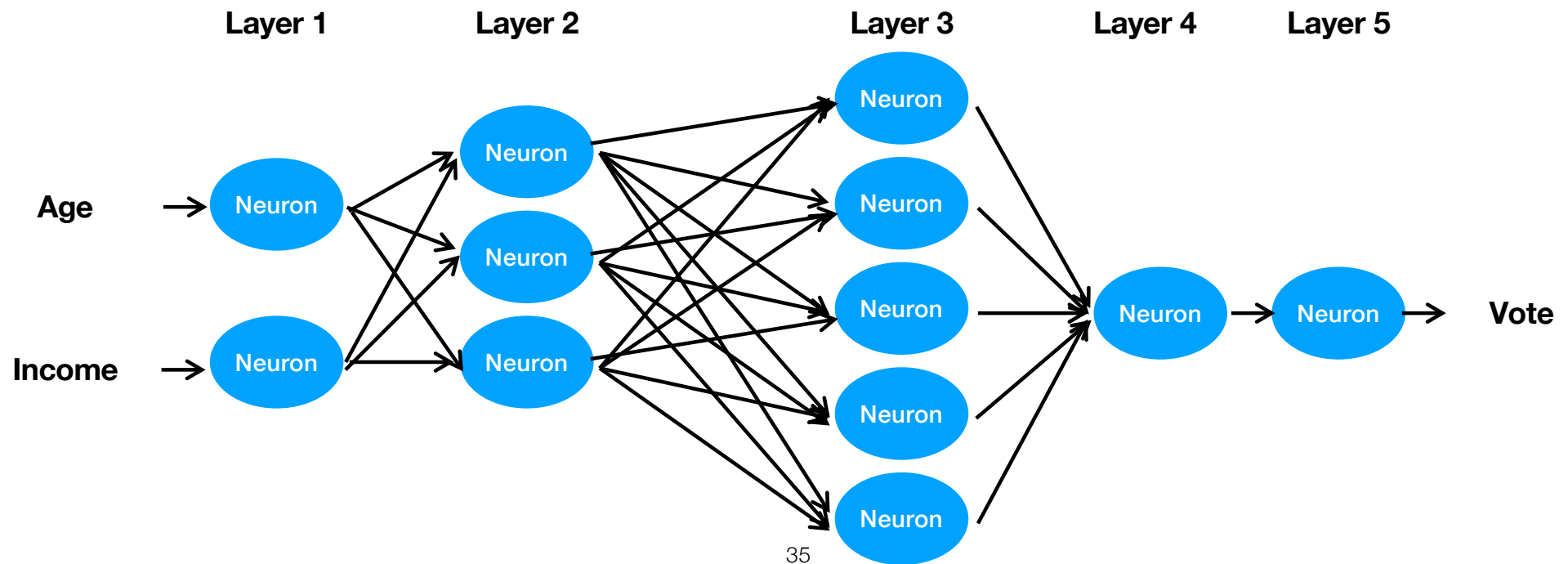


$$O_B(\text{income}, \text{age}) = \sigma(\theta_0^B + \theta_1^B \times \text{income} + \theta_2^B \times \text{age}) \quad O_D(\text{income}, \text{age}) = \sigma(\theta_0^D + \theta_1^D \times O_A + \theta_2^D \times O_B)$$

# Feed-Forward Networks with Fully Connected Layers

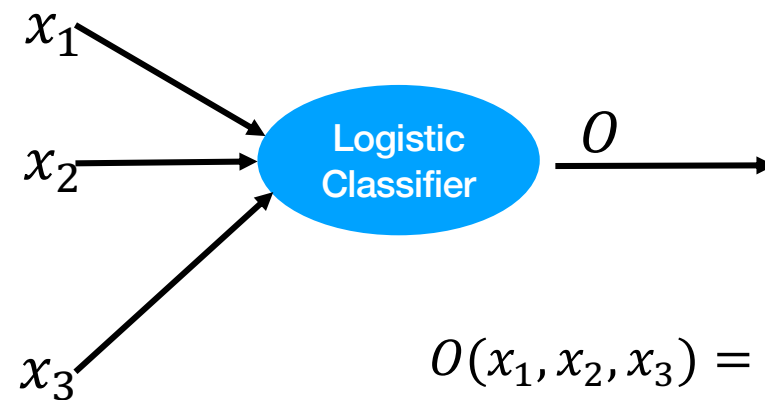
—> Each Neural Network architecture defines a function of the input with parameters  $\theta$

- Therefore, this is just a visual way of defining a complicated parameterized function of **Vote** given **Age** and **Income**:



# Parameters (1/2)

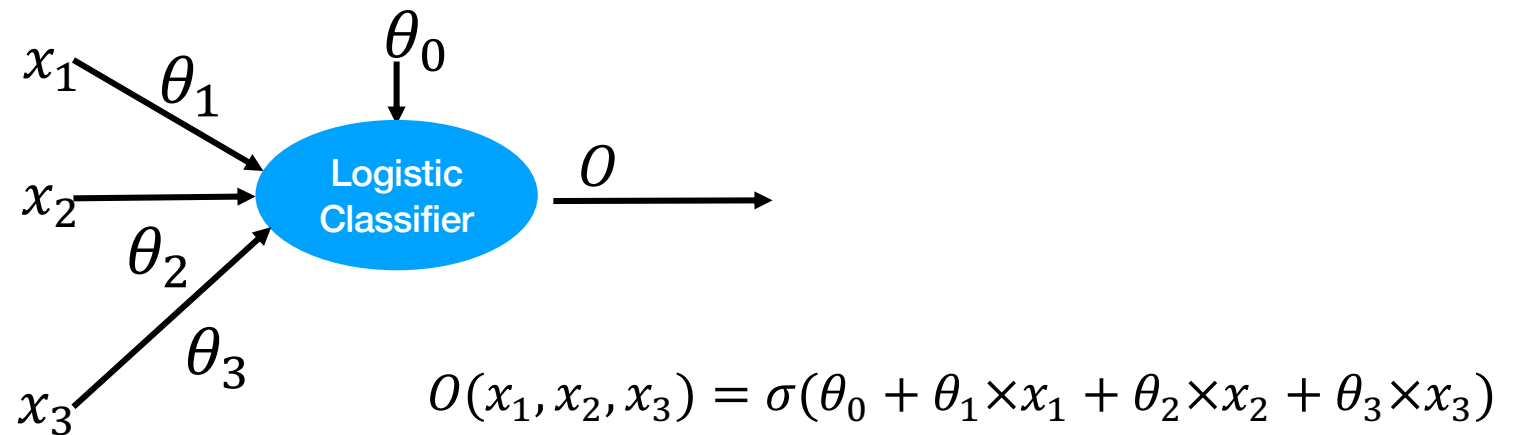
- If a neuron has N inputs, it has N+1 parameters



$$O(x_1, x_2, x_3) = \sigma(\theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \theta_3 \times x_3)$$

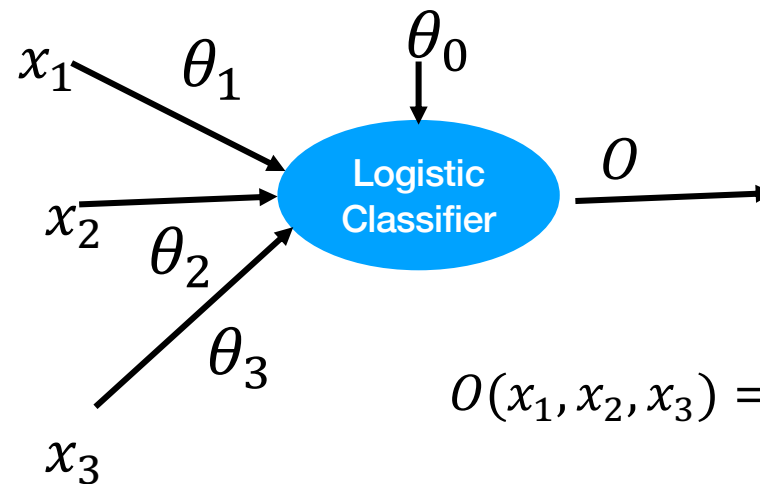
# Parameters (2/2)

- If a neuron has N inputs, it has N+1 parameters
- Visually, we can associate a parameter to each input, and show  $\theta_0$  separately



# Parameters: Terminology (1/2)

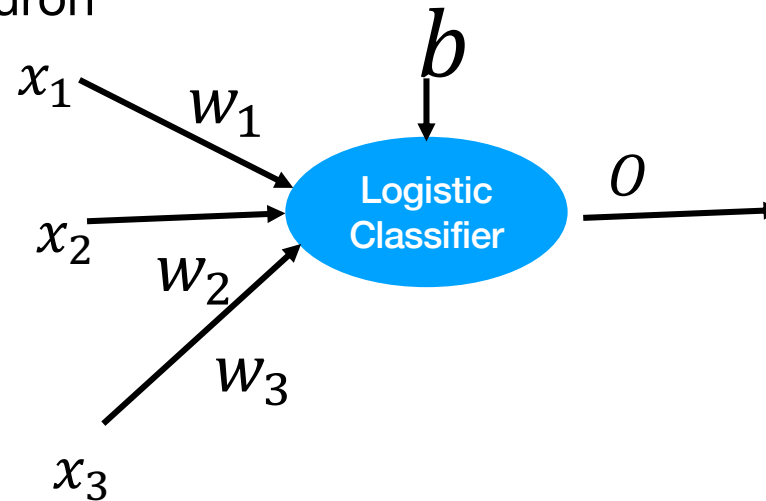
- $\theta_1, \theta_2, \theta_3$  are often called the **weights** of the neuron
- $\theta_0$  is often called the **bias** of the neuron



$$O(x_1, x_2, x_3) = \sigma(\theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \theta_3 \times x_3)$$

# Parameters: Terminology (2/2)

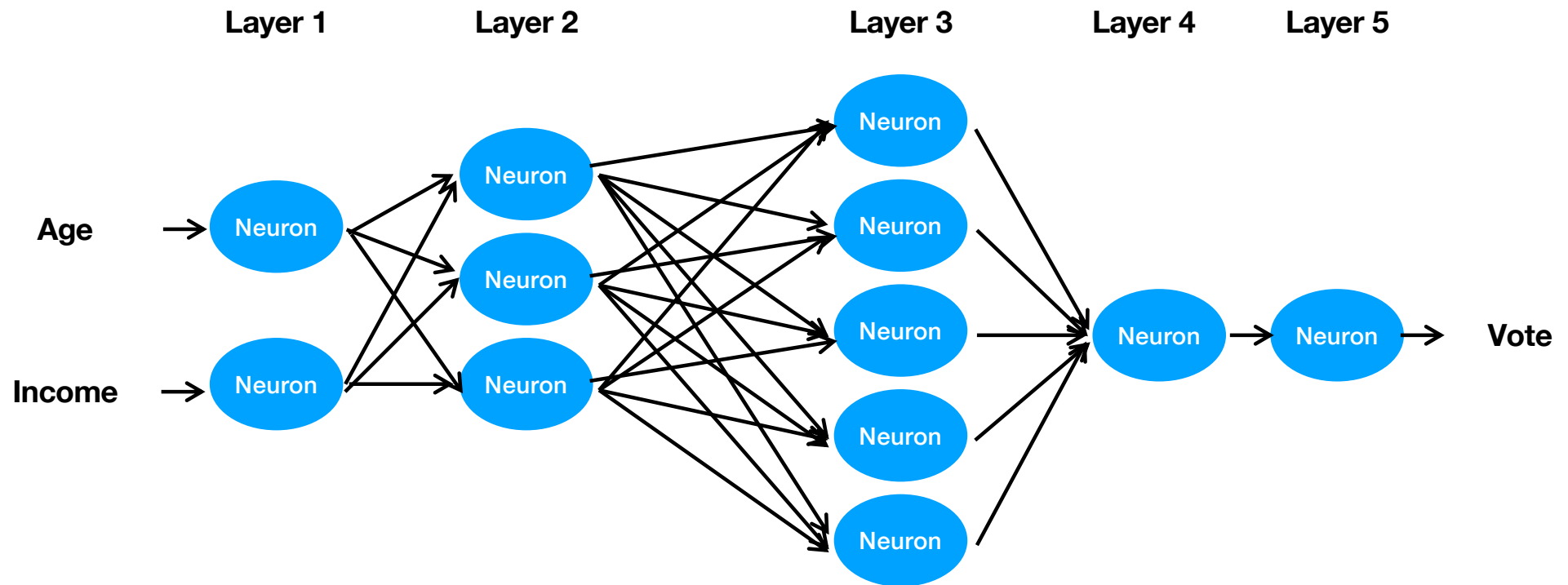
- $\theta_1, \theta_2, \theta_3$  are often called the **weights** of the neuron
  - They are therefore often also noted  $w_1, w_2, w_3$
- $\theta_0$  is often called the **bias** of the neuron
  - It is often noted  $b$



$$O(x_1, x_2, x_3) = \sigma(b + w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3)$$

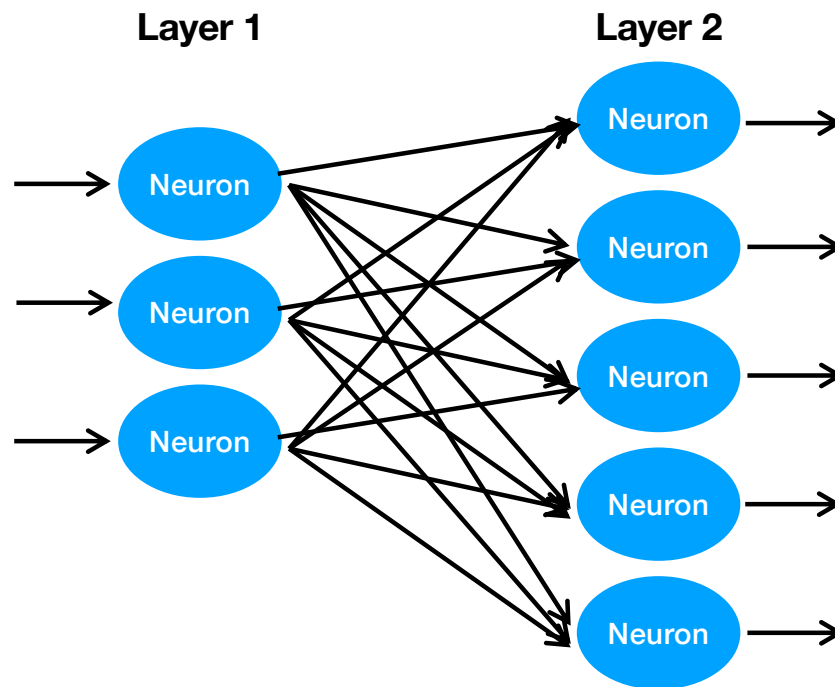
# Quiz

- How many parameters for this Neural Network?



# Parameters of Fully Connected Layers

- For a fully connected layer of  $N$  neurons, and with  $M$  neurons in the previous layer, the number of parameters is :  $N \times M + N$



**Layer 2 has  $5 \times 3 + 5 = 20$  parameters**

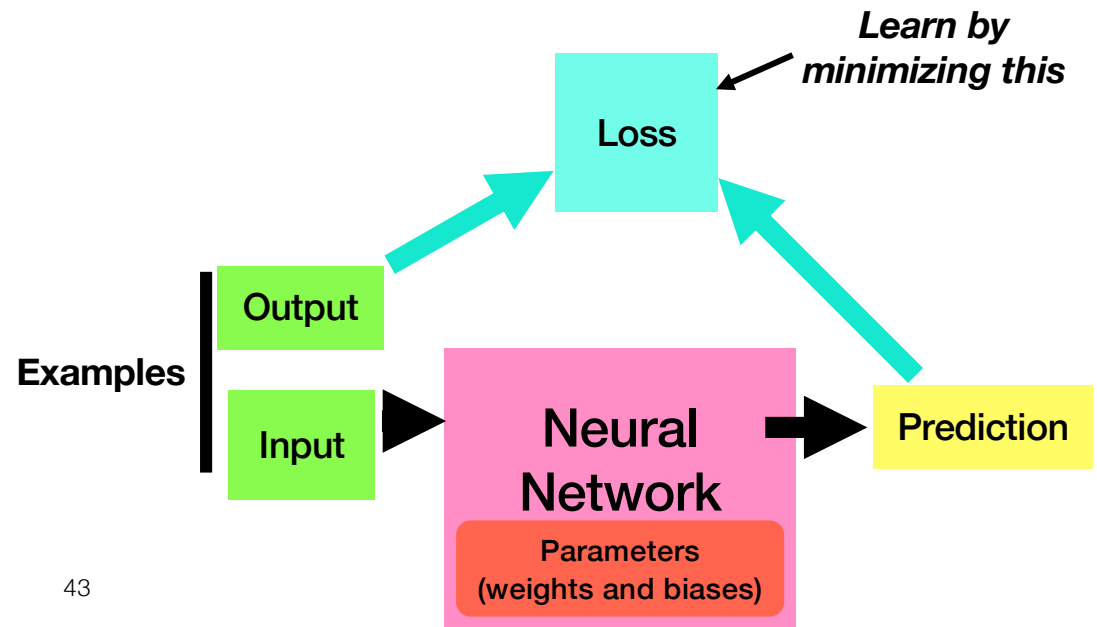
# How to Find Good Parameters

- The result of our Neural Network will depend on the value of the parameters
- How do we find good parameters?

# Supervised Learning (1/2)

## How to find the parameters?

- In supervised learning, we usually have:
  - A **MODEL**: a “parameterized” function that takes input and produces output
  - A **Loss**: A function that computes how different the model output is from the correct output
  - **Examples** of input and correct output (cigarettes smoked, age of death)



# Supervised Learning (2/2)

## How to find the parameters?

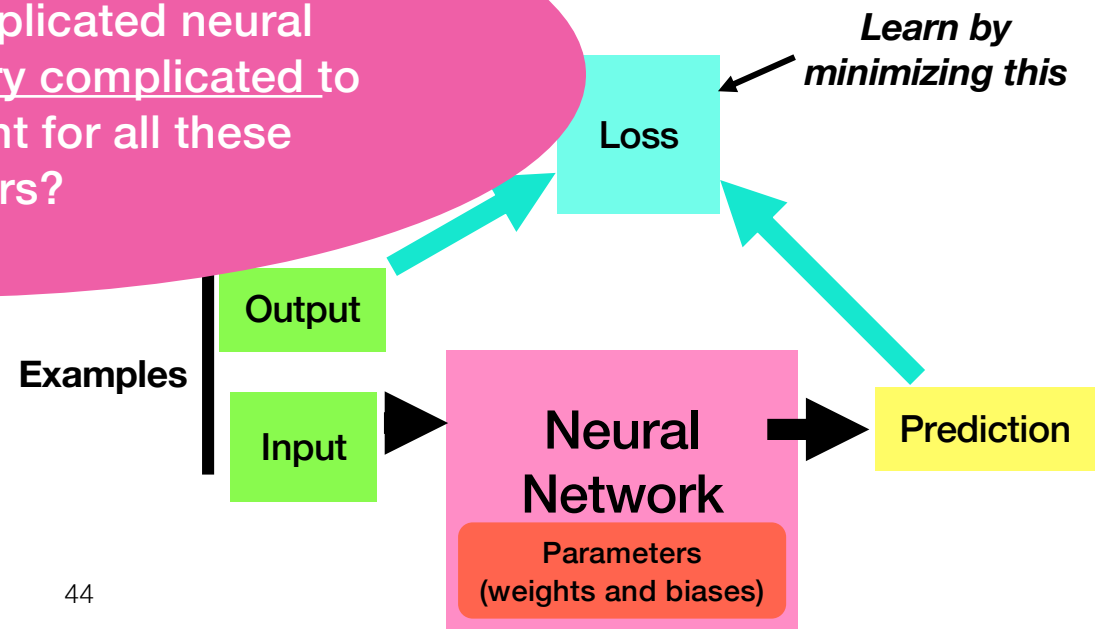
- In supervised learning, we have:
- A model that takes input and produces output

Yeah ok, we have seen that before...  
But if we have a complicated neural network, won't it be very complicated to compute the gradient for all these parameters?



**Loss** function that computes how different the model output is from the correct output

**Examples** of input and correct output (cigarettes smoked, age of death)



# The Backpropagation Algorithm

- Actually there is a method for **automatically** computing the *gradient* of a loss for a given *Feed Forward Neural Network*
- And as you should know now, if we can compute the gradient of the loss, we can find the parameters that minimize the loss by *gradient descent*

# The Backpropagation Algorithm

## (1/9)

- We will not see the details of the algorithm
  - It is actually quite simple, but involves some notions not everybody here is familiar with:
    - Partial derivatives
    - Dynamic programming
  - Anyway, in practice, you will use software that will do the backpropagation for you
    - -> You can actually train a Neural Network without understanding the Backpropagation algorithm (but you should know it exists)
- But let us see the **general idea**

# The Backpropagation Algorithm

## (2/9)

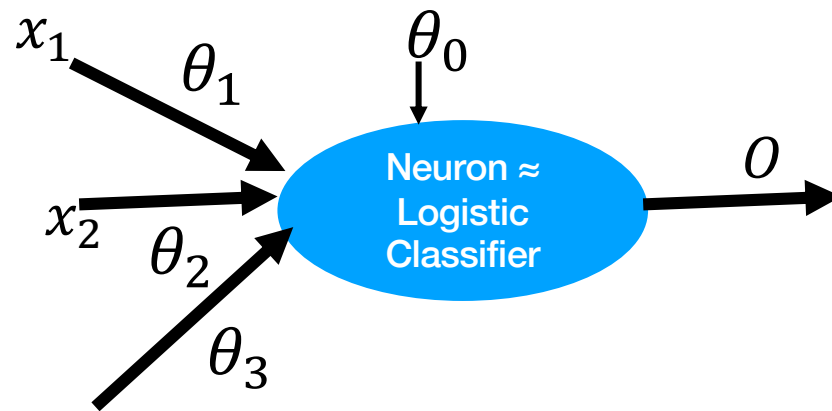
- The role of the backpropagation is to compute the gradient
  - Remember that the gradient is a vector of partial derivatives
- Now, remember the **composition rule** (a.k.a **chain rule**) for derivatives (1 variable case here, but there is a similar rule for the case with several variables):

$$f(x) = g(h(x)) \xrightarrow{\text{Chain rule}} f'(x) = h'(x) \times g'(h(x))$$

**This rule says that if I know how to compute the derivative of functions  $g(x)$  and  $h(x)$ , you know how to compute the derivative of  $g(h(x))$**

# The Backpropagation Algorithm (3/9)

- We know how to compute the gradient for a single neuron
- (see the lecture on logistic classifier for a formula)



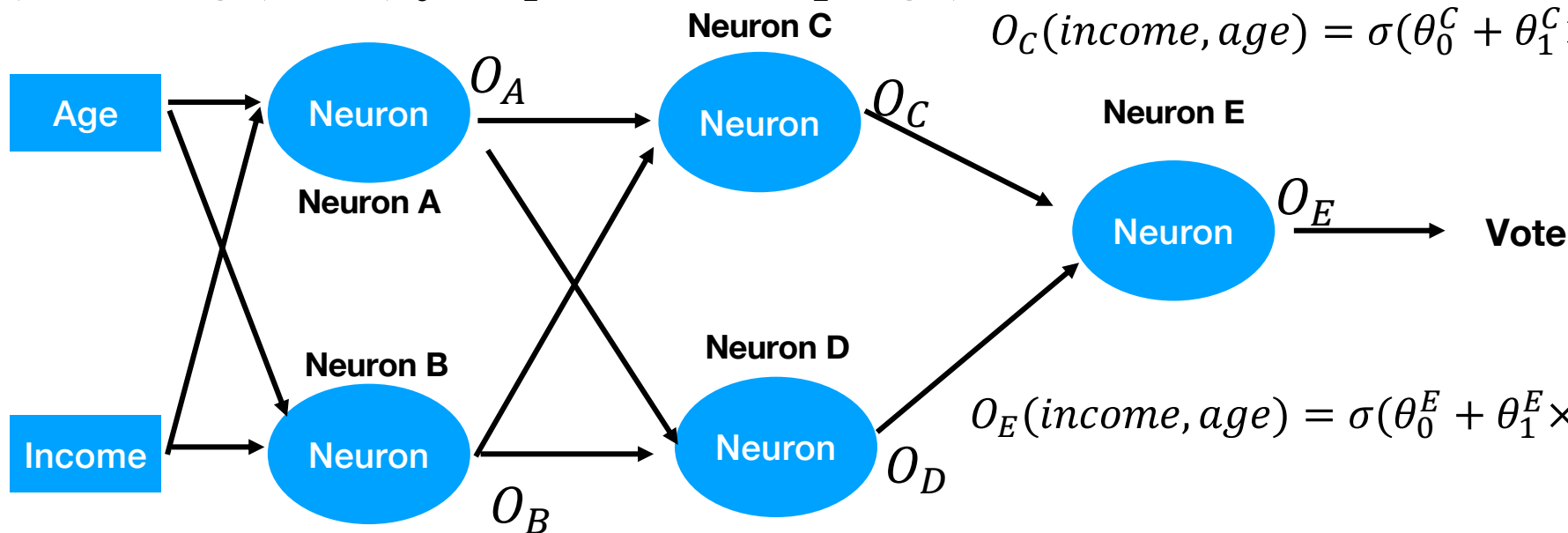
$$O(x_1, x_2, x_3) = \sigma(\theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \theta_3 \times x_3)$$

# The Backpropagation Algorithm (4/9)

- Actually, a neural network is just a composition of functions
- And we know how to compute the gradient for each of these functions

$$O_A(\text{income}, \text{age}) = \sigma(\theta_0^A + \theta_1^A \times \text{income} + \theta_2^A \times \text{age})$$

$$O_C(\text{income}, \text{age}) = \sigma(\theta_0^C + \theta_1^C \times O_A + \theta_2^C \times O_B)$$



$$O_E(\text{income}, \text{age}) = \sigma(\theta_0^E + \theta_1^E \times O_C + \theta_2^E \times O_D)$$

$$O_B(\text{income}, \text{age}) = \sigma(\theta_0^B + \theta_1^B \times \text{income} + \theta_2^B \times \text{age}) \quad O_D(\text{income}, \text{age}) = \sigma(\theta_0^D + \theta_1^D \times O_A + \theta_2^D \times O_B)$$

# The Backpropagation Algorithm (5/9)

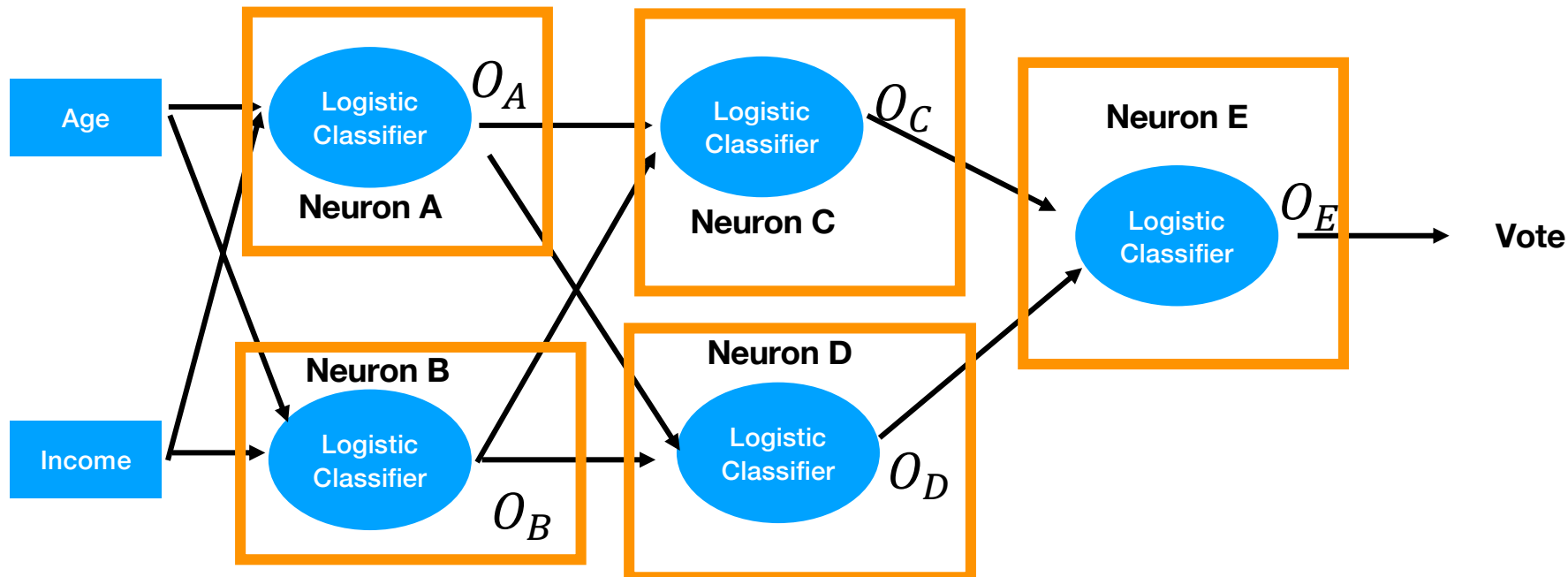
- Actually, a neural network is just a composition of functions
- And we know how to compute the gradient for one of these functions
- And we have seen there is a chain rule that says that if we know how to compute the derivative of simple functions, we can compute the derivative of their composition

$$f(x) = g(h(x)) \xrightarrow{\text{Chain rule}} f'(x) = h'(x) \times g'(h(x))$$

- This is the fundamental principle of the back propagation algorithm

# The Backpropagation Algorithm (6/9)

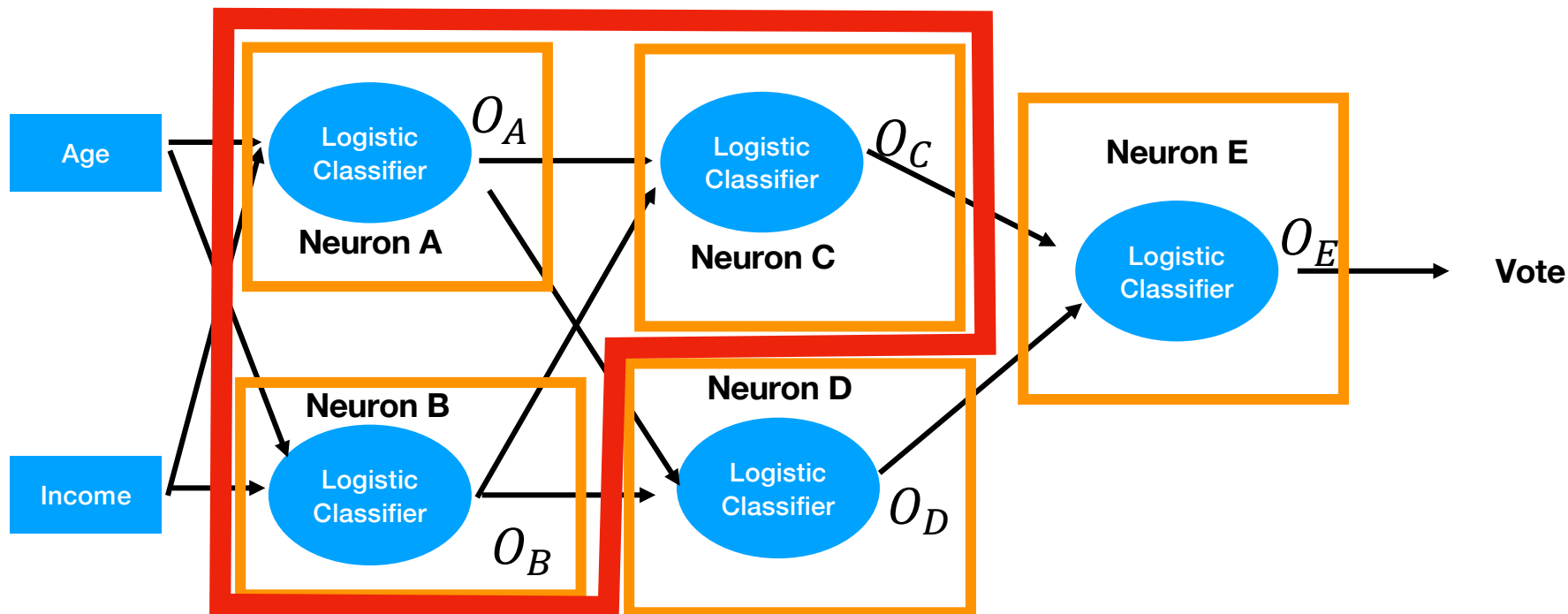
We know how to compute the gradient for the individual neurons:



# The Backpropagation Algorithm (7/9)

We know how to compute the gradient for the individual neurons

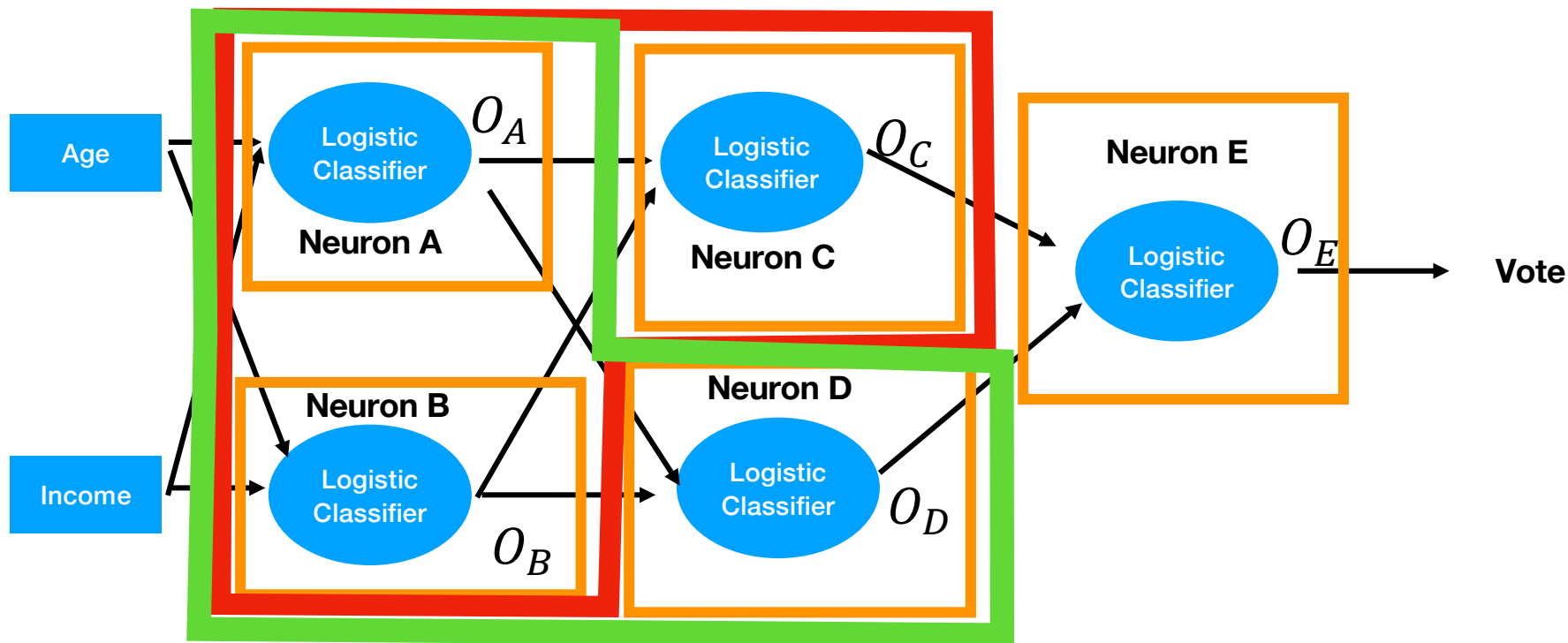
Thanks to the chain rule, we therefore can compute the gradient for this part of the network:



# The Backpropagation Algorithm (8/9)

We know how to compute the gradient for the individual neurons

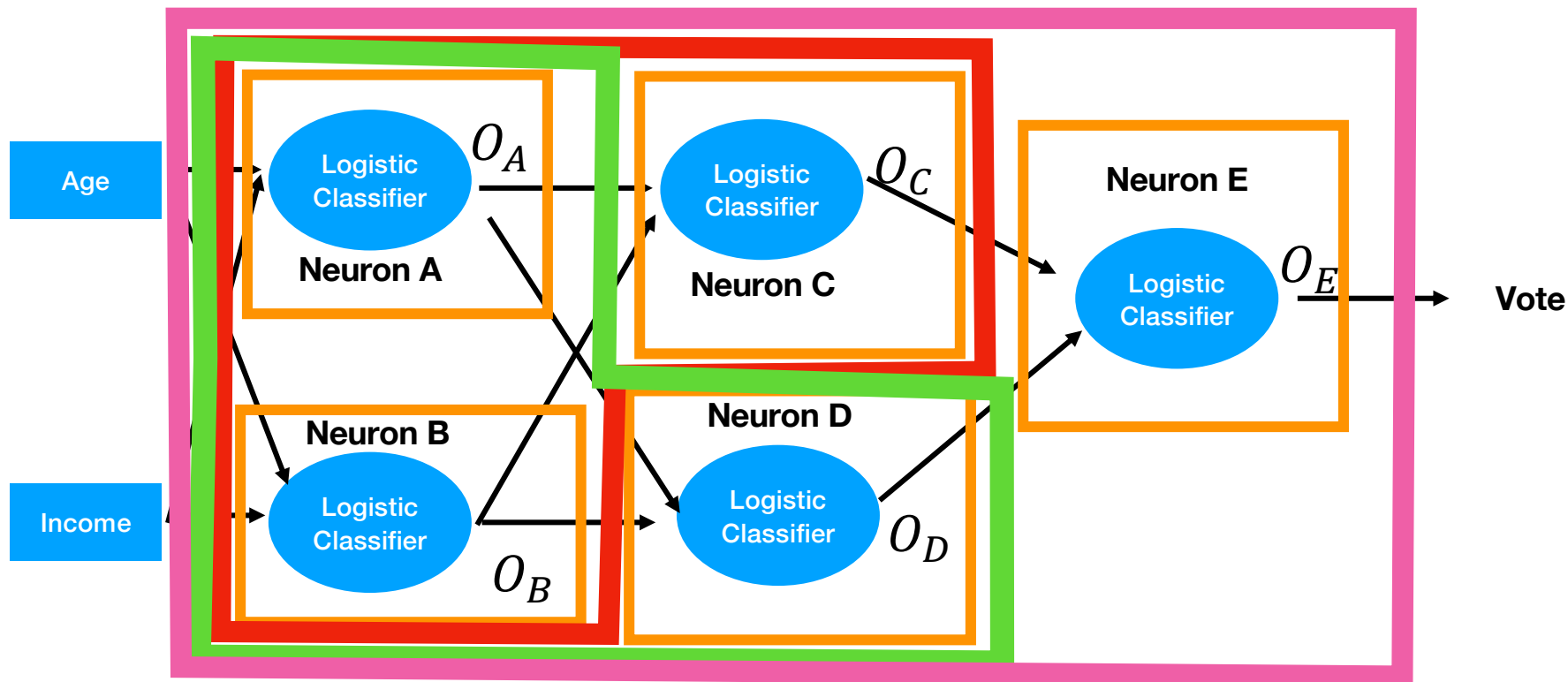
Thanks to the chain rule, we therefore can compute the gradient for this other part of the network:



# The Backpropagation Algorithm (9/9)

We know how to compute the gradient for the individual neurons

Finally, thanks to the chain rule, we can compute the gradient for the whole network:



# In Summary

- That is the general idea: if you have the formula for computing the gradient for **each** part of the neural network, you can compute the gradient for the **whole** network

# Backpropagation in Practice (1/9)

- Note that backpropagation does not work only with Neural Networks
- It can be used to compute the derivative of any composition of function
- In order to make you “feel” the process of the backpropagation algorithm (rather than describe it), let us apply it on a simple composition of functions

# Backpropagation in Practice (2/9)

- To make you feel how it actually work, let us consider the composition of simple functions:

$$h(x) = x + 1$$

$$g(x) = 2(x - 1)^2$$

$$f(x) = x^2 - x + 1$$

- We define  $K(x) = f(g(h(x)))$
- We want to compute the value and the derivative of  $K$  for  $x=1$  (for example)

# Backpropagation in Practice (3/9)

- To make you feel how it actually work, let us consider the composition of simple functions:

$$h(x) = x + 1 \qquad g(x) = 2(x - 1)^2 \qquad f(x) = x^2 - x + 1$$

- We define  $K(x) = f(g(h(x)))$
- We want to compute the value and the derivative of  $K$  for  $x=1$  (for example)
- One way is to compute  $K$  explicitly:  $K(x) = 4x^4 - 2x^2 + 1$ 
  - Then  $K(1) = 4 - 2 + 1 = 3$
- We can also compute  $K'(x)$  explicitly:  $K'(x) = 16x^3 - 4x$ 
  - Then  $K'(1) = 16 - 4 = 12$

# Backpropagation in Practice (4/9)

- To make you feel how it actually work, let us consider the composition of simple functions:

$$h(x) = x + 1 \qquad g(x) = 2(x - 1)^2 \qquad f(x) = x^2 - x + 1$$

- We define  $K(x) = f(g(h(x)))$
- We want to compute the value and the derivative of  $K$  for  $x=1$  (for example)
- Now, let us do it using backpropagation!

# Backpropagation in Practice (5/9)

- First, let us make sure we know the derivative of each individual function:

$$h(x) = x + 1$$

$$g(x) = 2(x - 1)^2$$

$$f(x) = x^2 - x + 1$$

$$h'(x) = 1$$

$$g'(x) = 4(x - 1)$$

$$f'(x) = 2x - 1$$

- Let us try to see  $K(x) = f(g(h(x)))$  as if it was a neural network:



**Note:** this graph is called the “computation graph” of  $K$

# Backpropagation in Practice (6/9)

- First, let us make sure we know the derivative of each individual function:

$$h(x) = x + 1$$

$$h'(x) = 1$$

$$g(x) = 2(x - 1)^2$$

$$g'(x) = 4(x - 1)$$

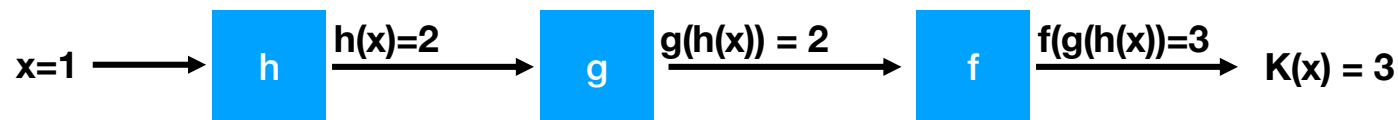
$$f(x) = x^2 - x + 1$$

$$f'(x) = 2x - 1$$

- Let us try to see  $K(x) = f(g(h(x)))$  as if it was a neural network:



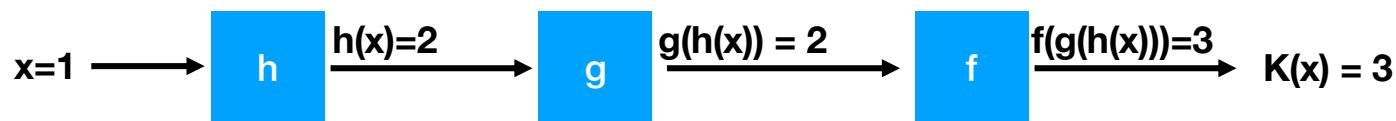
- Then let us compute  $K(1)$ :



# Backpropagation in Practice (7/9)

- First, let us make sure we know the derivative of each individual function:

$$\begin{array}{lll} h(x) = x + 1 & K(x) = f(g(h(x))) & h'(x) = 1 \\ g(x) = 2(x - 1)^2 & & g'(x) = 4(x - 1) \\ f(x) = x^2 - x + 1 & & f'(x) = 2x - 1 \end{array}$$



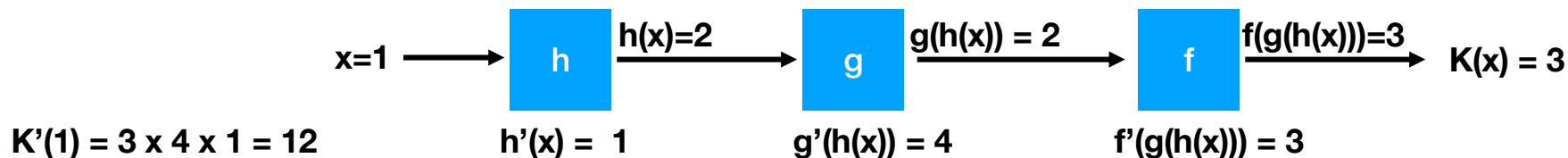
- By applying the chain rule twice, we have:  $K'(x) = f'(g(h(x))) \times g'(h(x)) \times h'(x)$
- Note that we have already computed  $h(x)$  and  $g(h(x))$ !
- Let us compute  $K'(1)$ :

# Backpropagation in Practice (8/9)

- First, let us make sure we know the derivative of each individual function:

$$\begin{array}{lll} h(x) = x + 1 & K(x) = f(g(h(x))) & h'(x) = 1 \\ g(x) = 2(x - 1)^2 & & g'(x) = 4(x - 1) \\ f(x) = x^2 - x + 1 & & f'(x) = 2x - 1 \end{array}$$

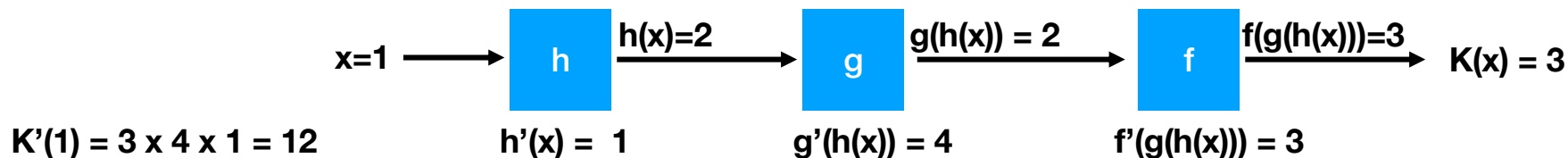
- By applying the chain rule twice, we have:  $K'(x) = f'(g(h(x))) \times g'(h(x)) \times h'(x)$
- Note that we have already computed  $h(x)$  and  $g(h(x))$ !
- Let us compute  $K'(1)$ :



# Backpropagation in Practice (9/9)

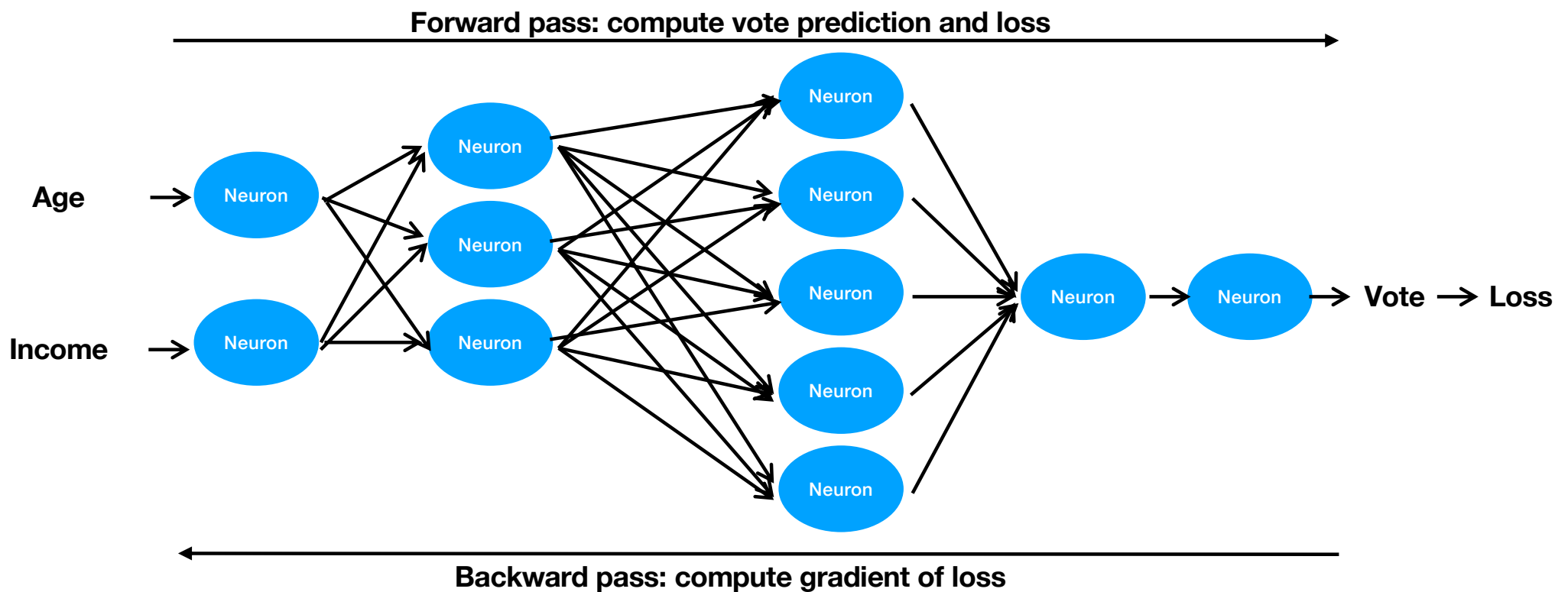
- Note that we did the computation in two passes:
  - First pass compute  $K(1)$
  - Second pass reuse the computations of the first pass to compute  $K'(1)$
- Backpropagation works always like that:
  - The first pass is called the **forward pass**
  - The second pass is called the **backward pass**

$$K'(x) = f'(g(h(x))) \times g'(h(x)) \times h'(x)$$



# The Backpropagation Algorithm

- Backpropagation works exactly the same way on neural networks:



# A Word About the Different Approaches for Computing Derivatives (1/3)

- Note that ***backpropagation*** is a specific case of ***Automatic Differentiation***
- There exists 3 methods for computing a derivative with a computer:
  - Automatic differentiation
  - Symbolic differentiation
  - Numerical differentiation

# A Word About the Different Approaches for Computing Derivatives (2/3)

- Going back to our example, let us illustrate how each method computes the derivative of  $K(x) = f(g(h(x)))$ :
  - Automatic differentiation uses the computation graph (like we saw)
  - Symbolic differentiation is the first thing we tried: compute  $K$  and  $K'$  explicitly:  $K(x) = 4x^4 - 2x^2 + 1 \quad K'(x) = 16x^3 - 4x \longrightarrow K'(1) = 12$
  - Numerical differentiation  $K(1.0001) - K(1)$

$$K'(1) \approx \frac{K(1.0001) - K(1)}{0.0001} = \frac{3.0012002 - 3}{0.0001} \approx 12$$

# A Word About the Different Approaches for Computing Derivatives (3/3)

- Note that ***backpropagation*** is a specific case of ***Automatic Differentiation***
- There exists 3 methods for computing a derivative with a computer:
  - Automatic differentiation
    - Commonly used for neural network
  - Symbolic differentiation
    - Sometimes used in combination with backpropagation for neural network (but usually less efficient)
  - Numerical differentiation
    - Way too slow! (and approximative)

# Further Readings on Backpropagation

- Section 6.5 of the “Deep Learning” Book
  - <https://www.deeplearningbook.org/contents/mlp.html>
- Somewhat easier introductions:
  - <https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/>
  - <https://medium.com/coinmonks/backpropagation-concept-explained-in-5-levels-of-difficulty-8b220a939db5>
  - <https://ayearofai.com/rohan-lenny-1-neural-networks-the-backpropagation-algorithm-explained-abf4609d4f9d>

# Neural Network Libraries

- As mentioned, we normally use libraries that will do all this backpropagation work for us

# Flow of Training With a Library:

- Define Model M  We will see how next time

- Repeat:

- Take some example (input, desired\_output) (eg. ( [age,income], vote))

- prediction: = Model(x)

- loss := loss\_function(prediction, desired\_output)

- loss.backward()

- optimizer.update(model)

**Forward pass**

**Ask library to compute  
backward pass**

**Ask library to perform a  
gradient descent update**

# Which Libraries

- There are many existing libraries for using Neural Network: Tensorflow, Torch, PyTorch, Chainer, Keras,....
- Let us describe a few of them

# Tensorflow

- ***Tensorflow*** is the library developed by Google
  - It is used internally by Google developers (eg. Google Translate run on Tensorflow)
  - Open Source
  - Use Python or C++ programming language

# Chainer

- **Chainer** is a library developed the Japanese company Preferred Networks
  - Open Source
  - Uses Python
  - Easy to use

# PyTorch

- ***Torch*** and **PyTorch**
  - Currently sponsored and used by Facebook
  - Open Source
  - Torch uses the LUA programming language
  - PyTorch uses the Python programming language
  - (PyTorch was initially a fork of ***chainer*** that got adapted to uses the torch libraries)

# Theano

- ***Theano***
  - One of the oldest library
  - Developed “Universite de Montreal”
  - Open Source
  - Can use symbolic differentiation
  - A bit difficult to use, and now discontinued

# Keras

- ***Keras***
  - Library that run on top of Tensorflow or Theano
  - Make them easier to use

# Next Time

- Looking at one of the most important type of Neural Network architectures: Feed-Forward with Fully-Connected Layers. And how they relate to Matrix Multiplication.
- We also look at how to implement a Fully-Connected Feed-Forward Neural Network

# Report

- We define  $K(x) = f(g(h(x)))$  with these functions. Compute  $K(1)$  and  $K'(1)$  with both the symbolic method and the backpropagation method

$$h(x) = x^2 \quad g(x) = 3x \quad f(x) = 1 + x^2$$

$$K(x) = f(g(h(x)))$$

$$K'(x) = f'(g(h(x))) \times g'(h(x)) \times h'(x)$$

- Write a report **in pdf** and submit via Panda
- Submission due: **next lecture**
- Name the pdf file as **student id\_name**.