

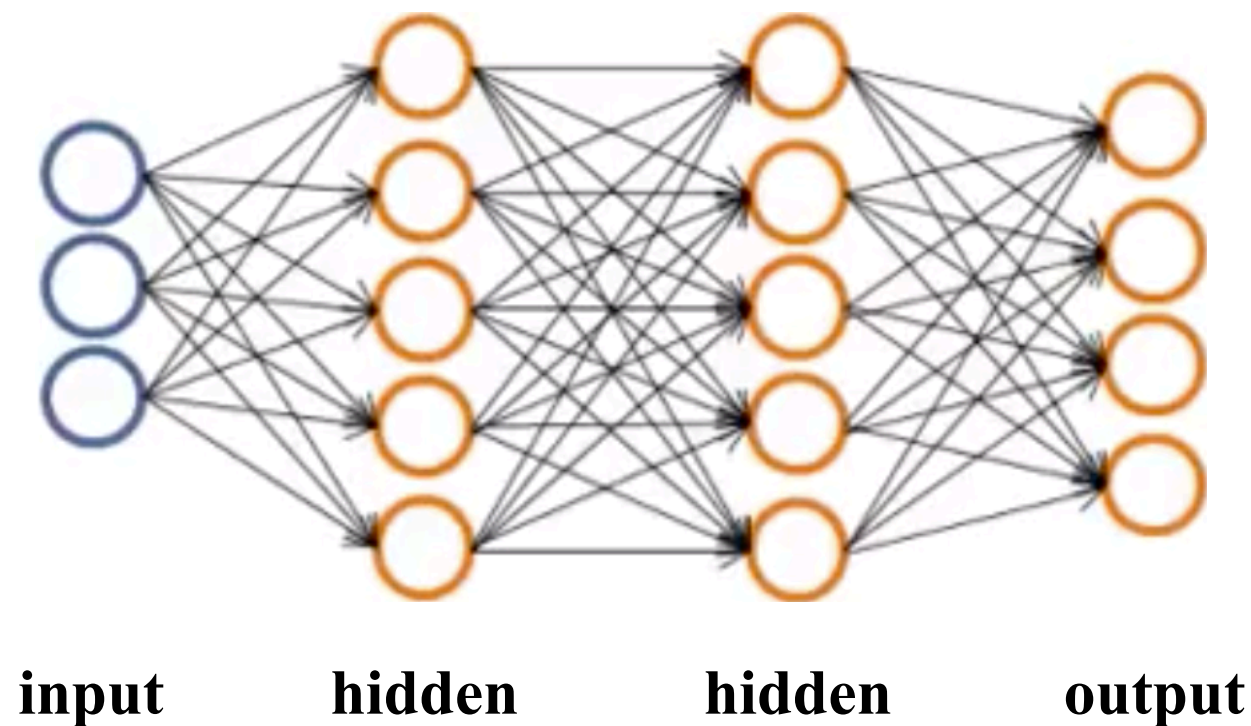
# Neural Networks: Feedforward

Teeradaj Racharak (เอ็กซ์)  
[r.teeradaj@gmail.com](mailto:r.teeradaj@gmail.com)



# Recap

- Given a dataset  $(\mathbf{x}^{(i)}, y^{(i)})$ , the goal of neural network is to find intermediate features that will best predict each  $y^{(i)}$  from the corresponding  $\mathbf{x}^{(i)}$ .
- Neural networks are sometime called ‘black boxes’ since it is difficult to understand the ‘meaning’ of induced intermediate features.



# Neural Network's Notation

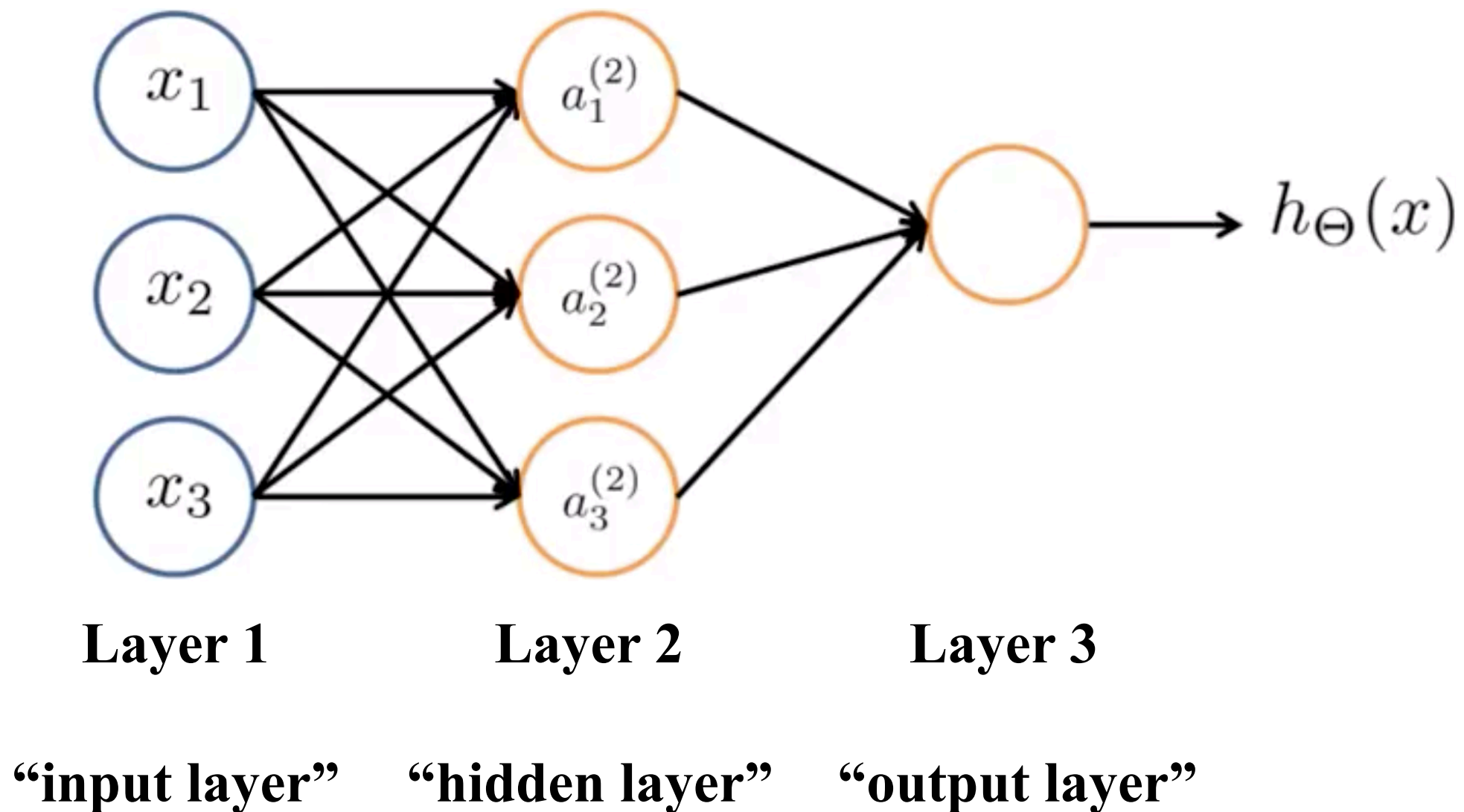
Suppose we have an input layer composed of features  $x_1, x_2, \dots$

Let's use the notation  $a_i^{(j)}$  to indicate the **activation** of the  $i^{\text{th}}$  unit in the  $j^{\text{th}}$  layer *e.g.*

- $a_1^{(2)}$  is the output of the first hidden unit in the first hidden layer;
- $a_1^{(3)}$  is the output of the first unit in the second hidden layer (or the output layer in a network with only one hidden layer).

To unify the notation, we let  $a_i^{(1)} = x_i$  *i.e.* we treat the input as layer 1.

# Neural Network's Notation



# Activation Functions

We saw a simple **ReLU** network already.

**Logistic regression** can also be treated as a simple neural network with one output unit and no hidden units *i.e.*

$$g(\mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

can be written in neural networks as two steps:

1. Calculate the **linear response**  $z = \theta_0 + \sum_{i=1}^n \theta_i x_i$
2. Calculate the **activation function**  $a = \sigma(z)$  where  $\sigma(z) = \frac{1}{1 + e^{-z}}$

# Activation Functions

Usually,  $g(z)$  is non-linear.

The most common activation functions are:

1. Sigmoid:  $g(z) = \frac{1}{1 + e^{-z}}$
2. ReLU (the default activation function in modern neural networks):  $g(z) = \max(z, 0)$
3. Hyperbolic tangent:  $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

# Forward Propagation

- The full calculation proceeds as follows:
- For the first hidden unit in the first hidden layer, we calculate:

$$z_1^{(2)} = \Theta^{(1)T} \mathbf{x} + \theta_0^{(1)}$$

and

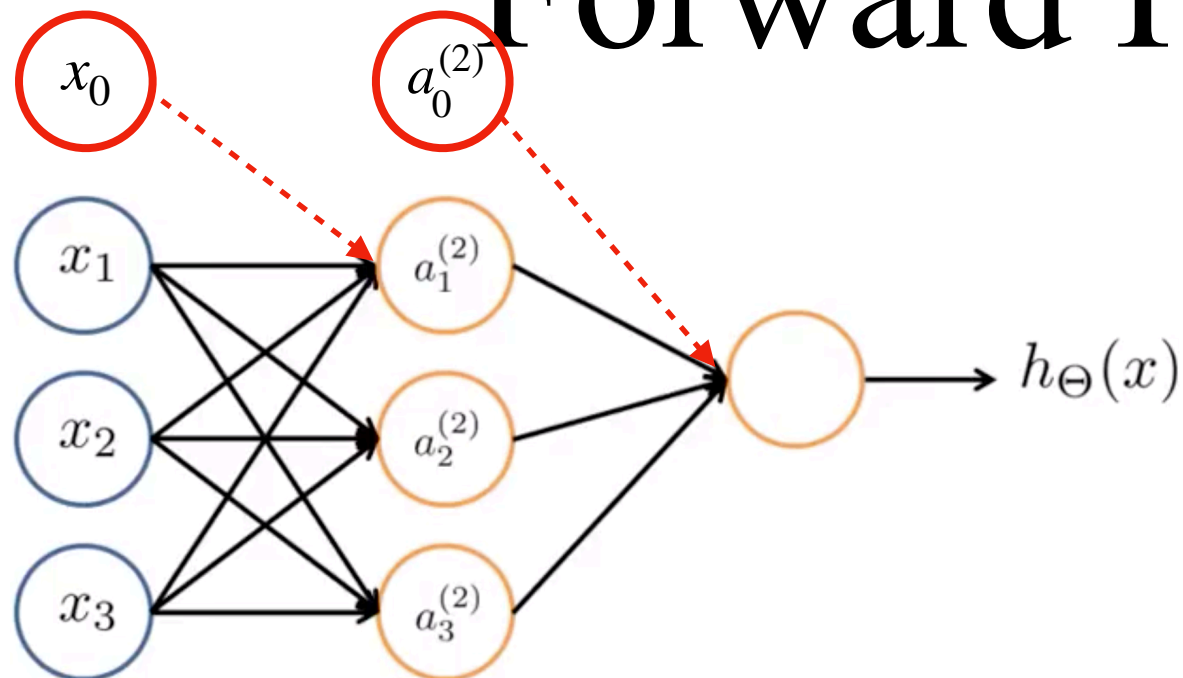
$$a_1^{(2)} = g(z_1^{(2)})$$

*Play a role of **bias**, which always output 1*

where  $\Theta^{(j)}$  is a matrix of parameters or weights **controlling function mapping** from layer  $j$  to layer  $j + 1$ .

- We **repeat for each unit in each layer** to get the final output layer.

# Forward Propagation



3 input units

3 hidden units

$a_i^{(j)}$  = ‘activation’ of unit  $i$  in layer  $j$   
 $\Theta^{(j)}$  = matrix of weights controlling  
 function mapping from  
 layer  $j$  to layer  $j + 1$

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \end{aligned} \quad \Theta^{(i)} \in \mathbb{R}^{3 \times 4}$$

$$h_{\Theta}(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

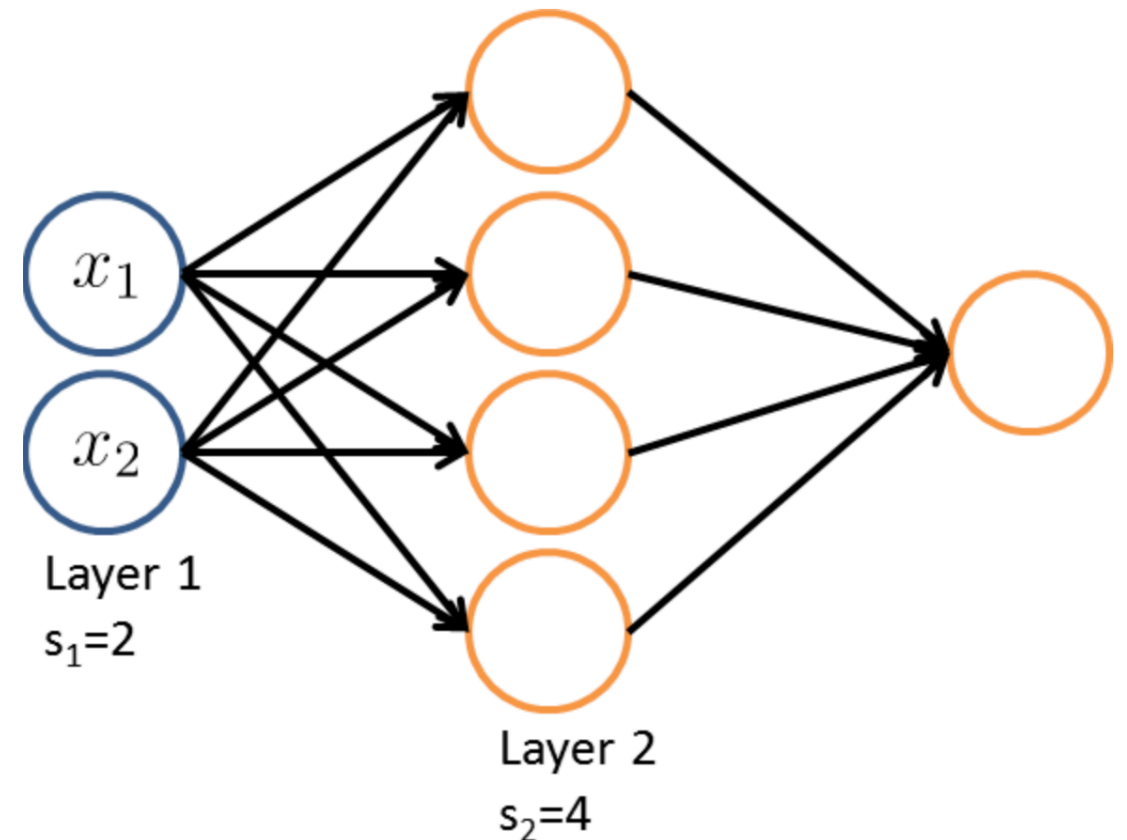
**Formally,** If network has  $s_j$  units in layer  $j$ ,  $s_{j+1}$  units in layer  $j + 1$ , then  $\Theta^{(j)}$  will be of dimension  $s_{j+1} \times (s_j + 1)$ .



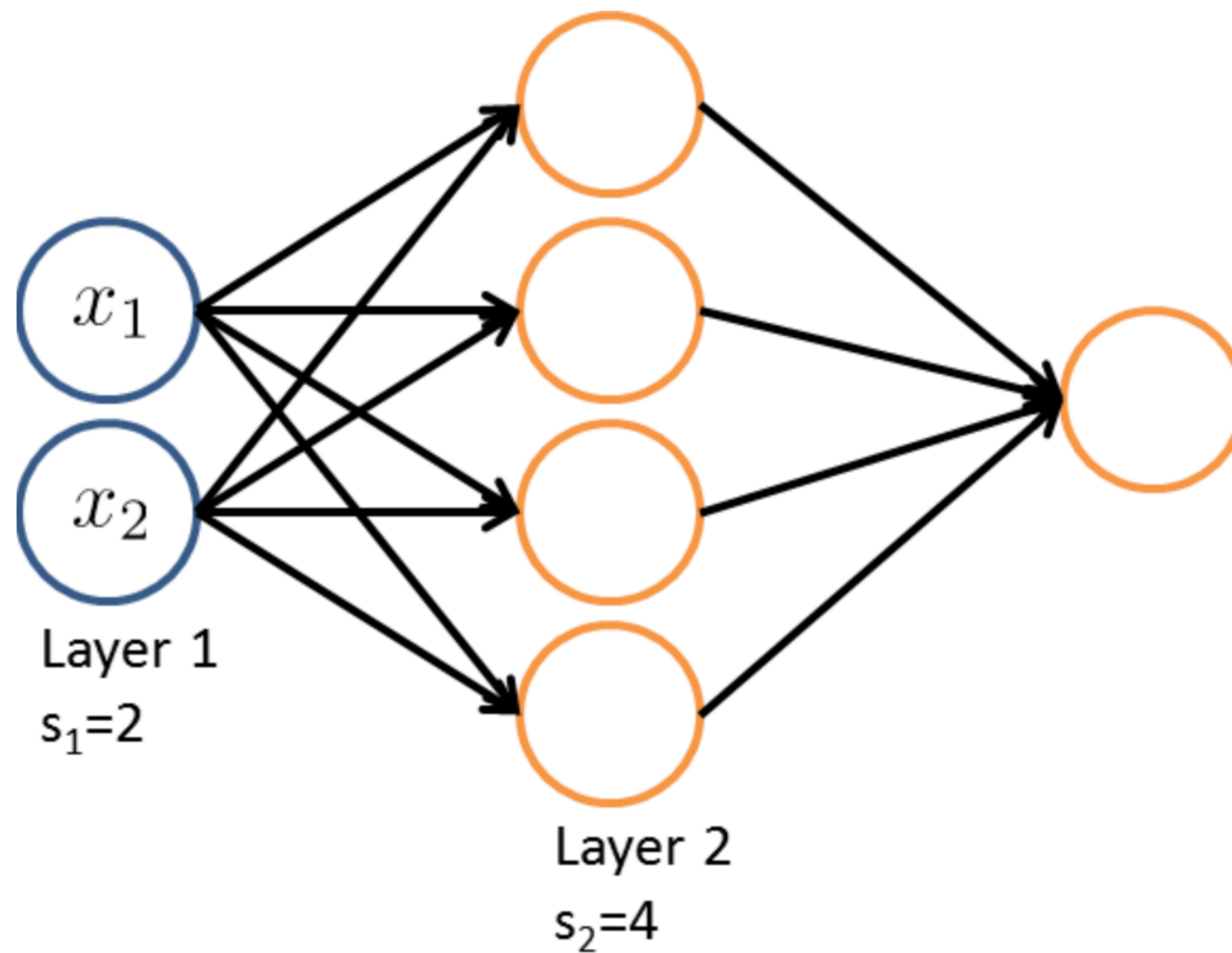
# Question

Consider the neural network.  
What is the dimension of  $\Theta^{(1)}$  ?

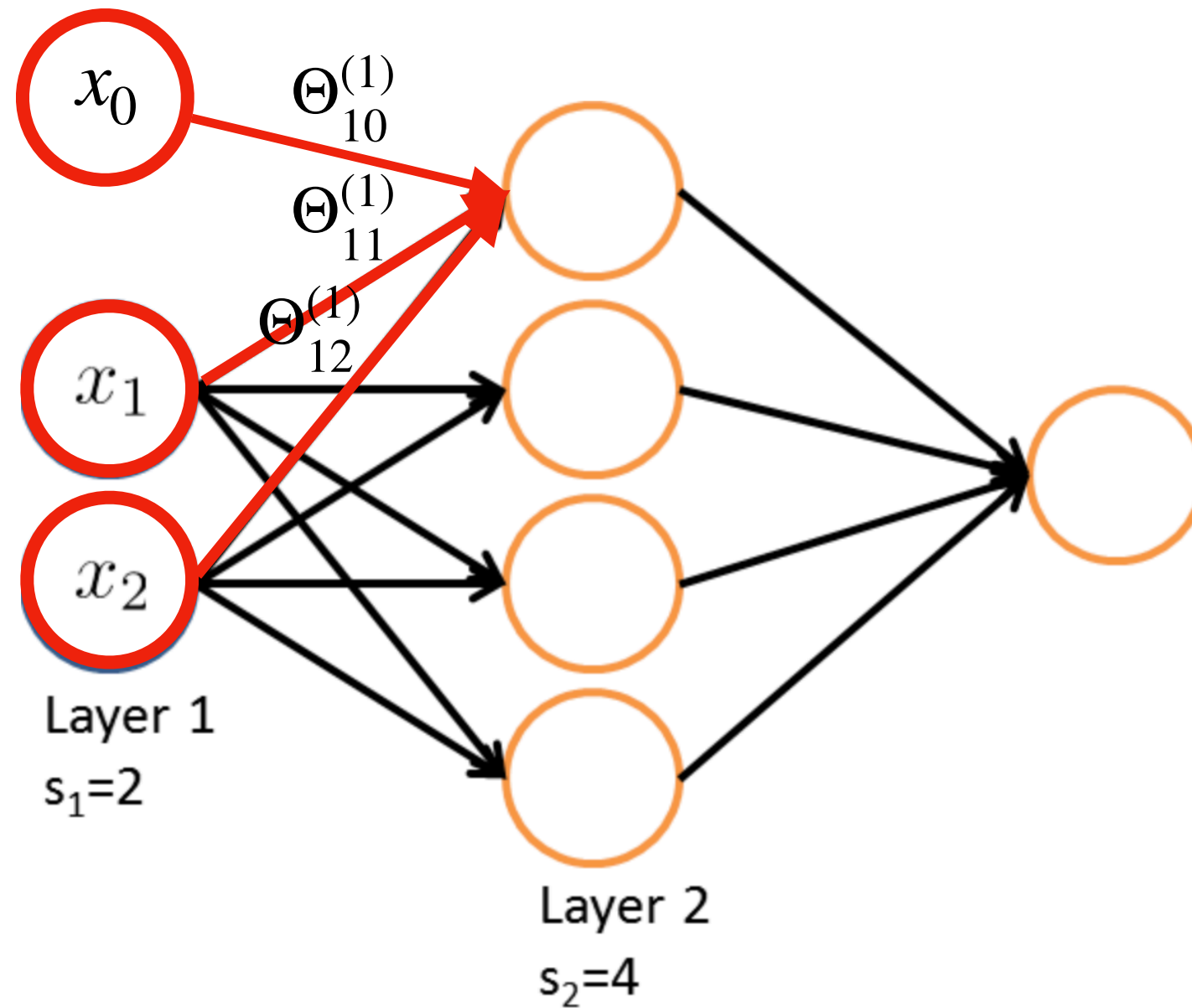
- (i) 2 x 4
- (ii) 4 x 2
- (iii) 3 x 4
- (iv) 4 x 3



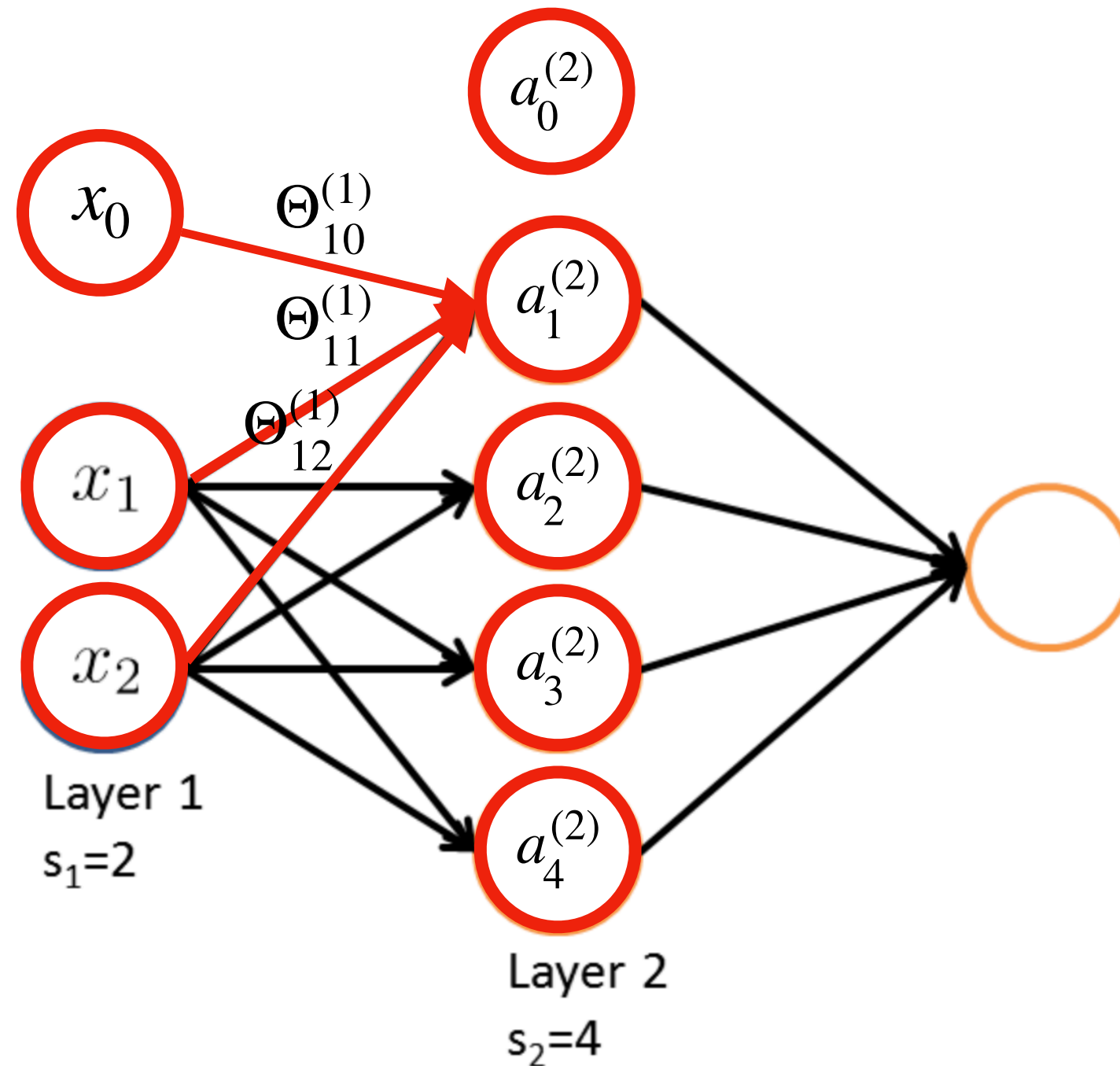
# Feedforward Propagation



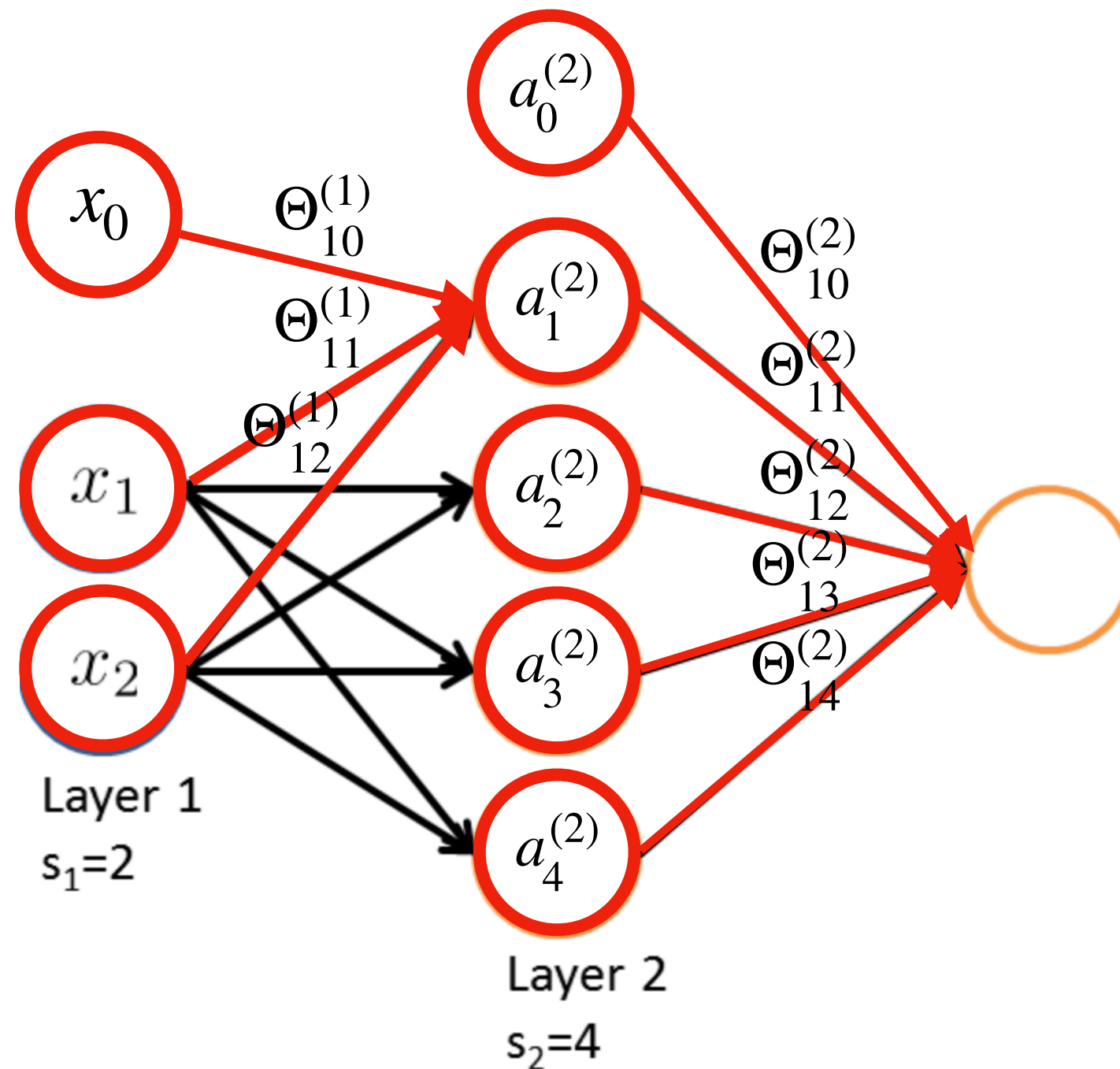
# Feedforward Propagation



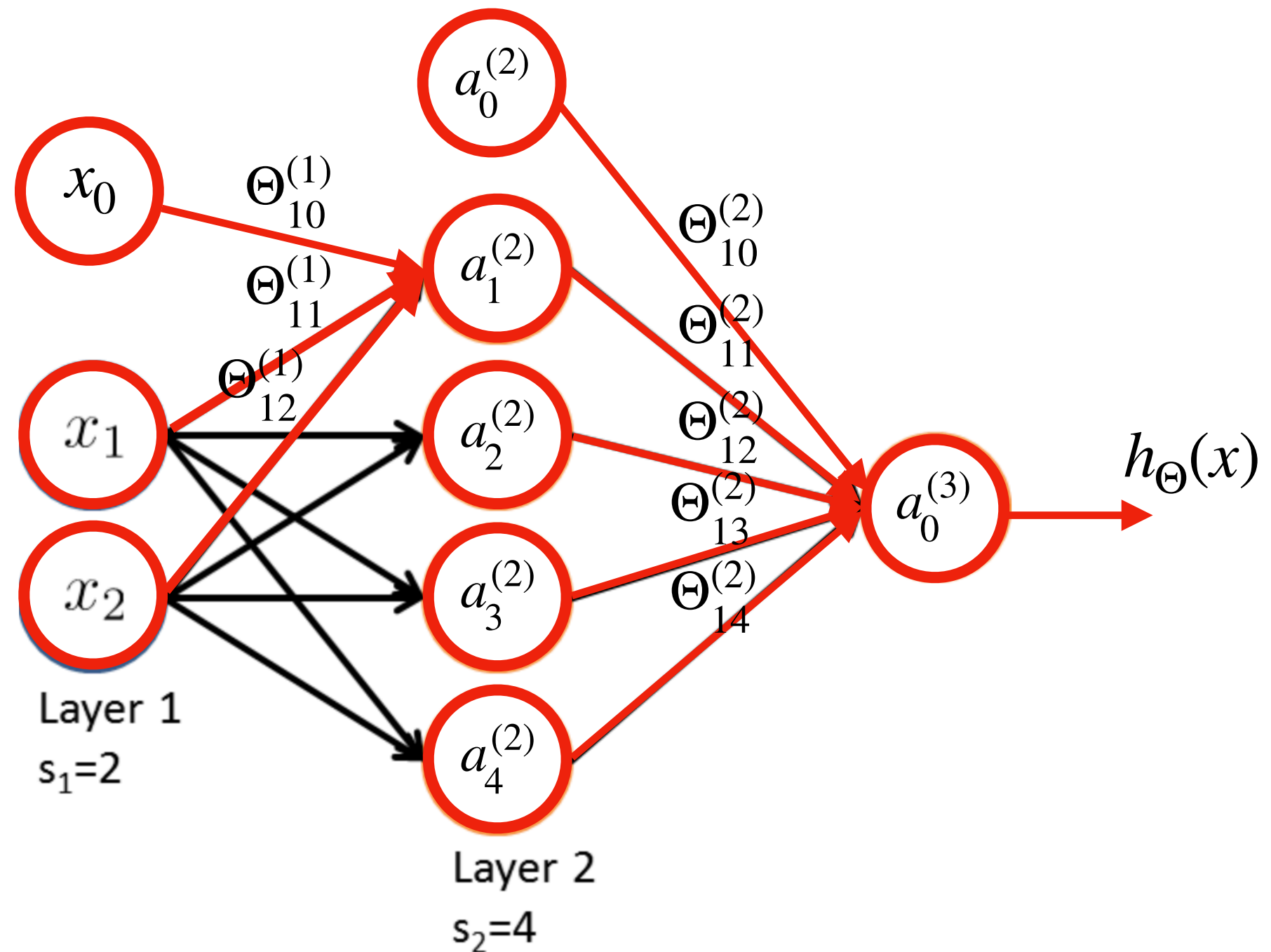
# Feedforward Propagation



# Feedforward Propagation



# Feedforward Propagation



# Forward Propagation

## What remains?

Now that we understand the feed-forward computation of a neural network. Then, let's talk about:

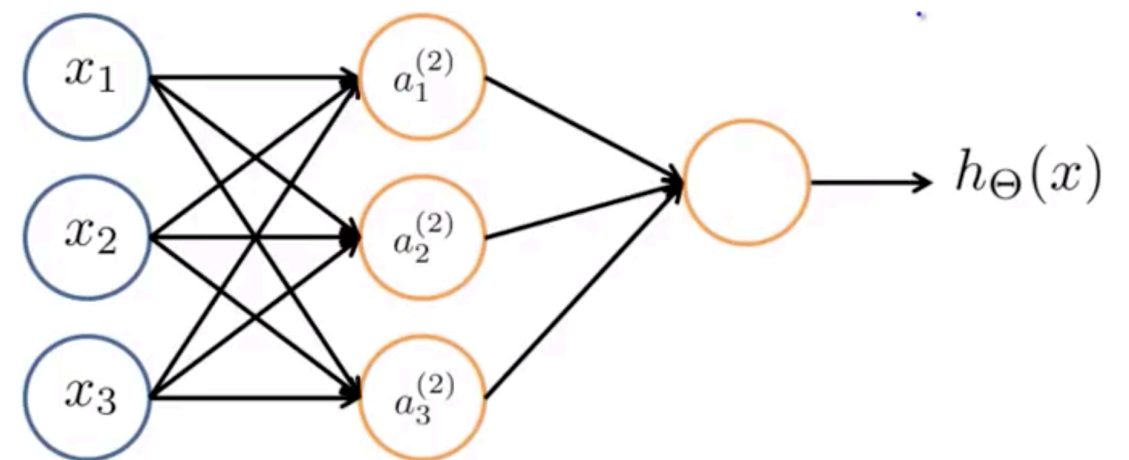
- **Efficient execution** of the feed-forward computation,
- The **gradient descent** process used to learn the **weights**  $\Theta$

**Remark:** Ng uses the notation  $\Theta$  for weights but some of other materials use **W**. So, our lecture slides will use them interchangeably.

# Efficient Computation

Consider calculating hidden unit activations. We have:

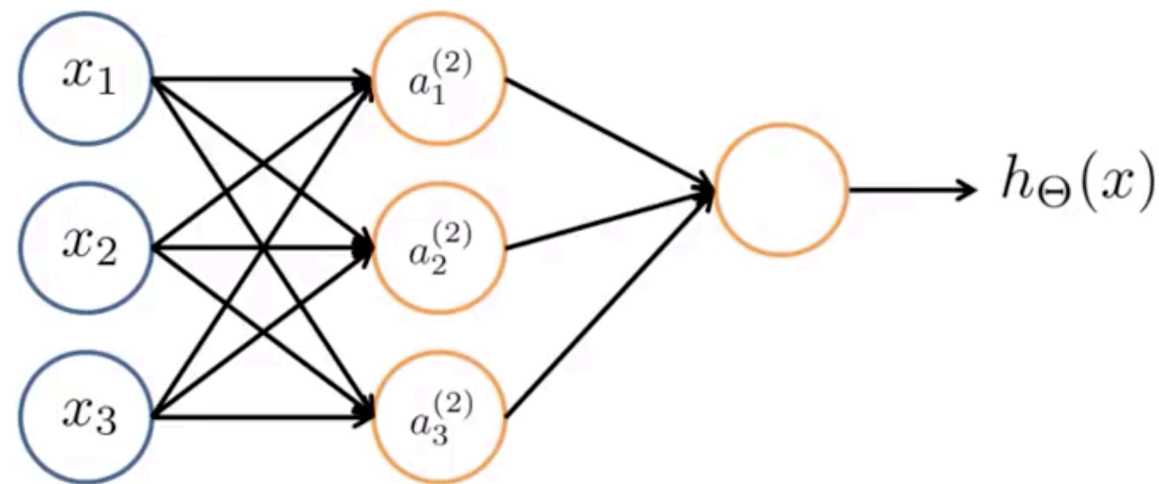
$$\begin{array}{lll} \mathbf{z}_1^{(2)} = \mathbf{W}_1^{(1)T} \mathbf{x} + \mathbf{b}_1^{(1)} & \text{and} & \mathbf{a}_1^{(2)} = g(\mathbf{z}_1^{(2)}) \\ \vdots & & \vdots \\ \mathbf{z}_4^{(2)} = \mathbf{W}_4^{(1)T} \mathbf{x} + \mathbf{b}_4^{(1)} & \text{and} & \mathbf{a}_4^{(2)} = g(\mathbf{z}_4^{(2)}) \end{array}$$





# Efficient Computation

Consider calculating hidden unit activations of the following network:



Depending on the ‘deepness’ of an architecture, for a single input, we may be doing a calculation of activations for 100 or 1,000 of units.

Code to implement this procedure using ‘for’ loops and the like will run **very slowly**, especially if implemented in a bytecode based on language like Python or Java.

# Efficient Computation

What is needed is the ability to perform matrix algebra with a single library call or instruction that is highly optimized, using CPU instructions provided for **vector operations**.

**BLAS** is a well-known library that does this and is used by numpy:

**Basic Linear Algebra Subprograms (BLAS)** is a [specification](#) that prescribes a set of low-level routines for performing common [linear algebra](#) operations such as [vector](#) addition, [scalar multiplication](#), [dot products](#), linear combinations, and [matrix multiplication](#). They are the *[de facto](#)* standard low-level routines for linear algebra libraries; the routines have bindings for both [C](#) and [Fortran](#). Although the BLAS specification is general, BLAS implementations are often optimized for speed on a particular machine, so using them can bring substantial performance benefits. BLAS implementations will take advantage of special floating point hardware such as vector registers or [SIMD](#) instructions.

[\(from Wikipedia\)](#)

Alternatively, we may parallelize computation by recruiting GPU resources.

# Efficient Computation

For an entire layer, to use vectorized computation, we need to perform the operation in one operation:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \end{aligned}$$

This can be implemented in a single Python statement:

```
W1 = np.matrix(np.random.normal(0, 1, (3, 4)))  
x = np.matrix(np.random.normal(0, 1, (4, 1)))  
z2 = np.dot(W1, x)
```

} This would run much faster than the equivalent doubly-nested for loop !

# Efficient Computation

For an entire layer, to use vectorized computation, we need to perform the operation in one operation:

$$\mathbf{z}_1^{(2)} = \mathbf{W}_1^{(1)T} \mathbf{x} + \mathbf{b}_1^{(1)}$$

This can be implemented in a single Python statement:

```
W1 = np.matrix(np.random.normal(0, 1, (3, 4)(3, 3)))  
b1 = np.matrix(np.random.normal(0, 1, (4, 1)(3, 1)))  
x = np.matrix(np.random.normal(0, 1, (3, 1)))  
z2 = W1.T * x + b1
```

This would run much  
faster than the equivalent  
doubly-nested for loop !

# Efficient Computation

Then, to calculate  $a^{(2)}$  as a vector operation, we can hopefully use vectorized functions in our implementation language.

If we have the sigmoid function, for example, we implement:

$$g(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}$$

as

```
def g(z):  
    return 1 / (1 + exp(-z))
```

```
z2 = np.matrix([[1, 2, 3]]).T  
a2 = g(z2)
```

This will run much faster than  
executing the `exp( )` function  
within a Python loop !

# Efficient Computation

Next, consider performing a computation over many training examples.

Performing the operation:

$$\mathbf{z}_1^{(2)} = \mathbf{W}_1^{(1)T} \mathbf{x}^{(i)} + \mathbf{b}_1^{(1)}$$

inside a loop for every training example  $i$  would be slower than the vectorized operation.

$$\mathbf{z}_1^{(2)} = \mathbf{W}_1^{(1)T} \mathbf{X}^T + \mathbf{b}_1^{(1)}$$

# Efficient Computation

Noted that some languages *e.g.* Python also allow **broadcasting** of the addition operation horizontally:

```
A = np.matrix([[1, 2, 3], [2, 3, 4]])
```

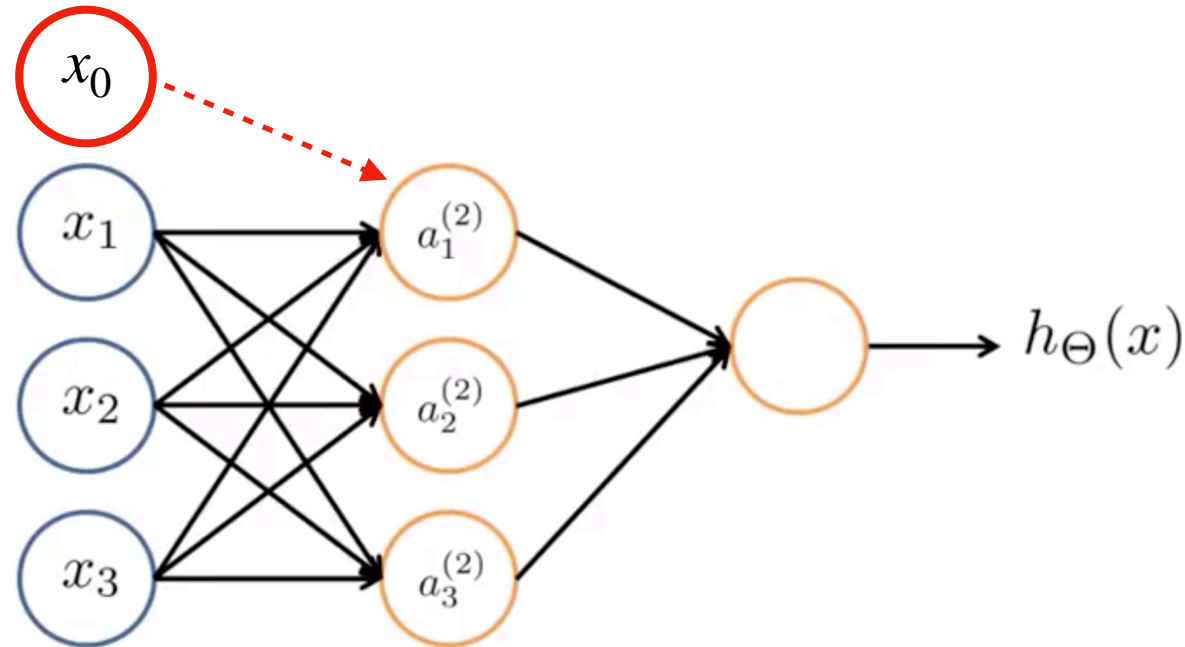
```
b = np.matrix([[2, 3]]).T
```

```
>> A + b
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 5 \\ 5 & 6 & 7 \end{bmatrix}$$

With broadcasting, the computation will be **much more efficient** than for loop!

# Efficient Computation



$z_i^{(j)}$ : a weighted linear combination of the inputs that go into a neuron *e.g.*  $a_1^{(2)} = g(z_1^{(2)})$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$\xrightarrow{\text{dashed arrow}} z_1^{(2)}$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$\xrightarrow{\text{dashed arrow}} z_2^{(2)}$

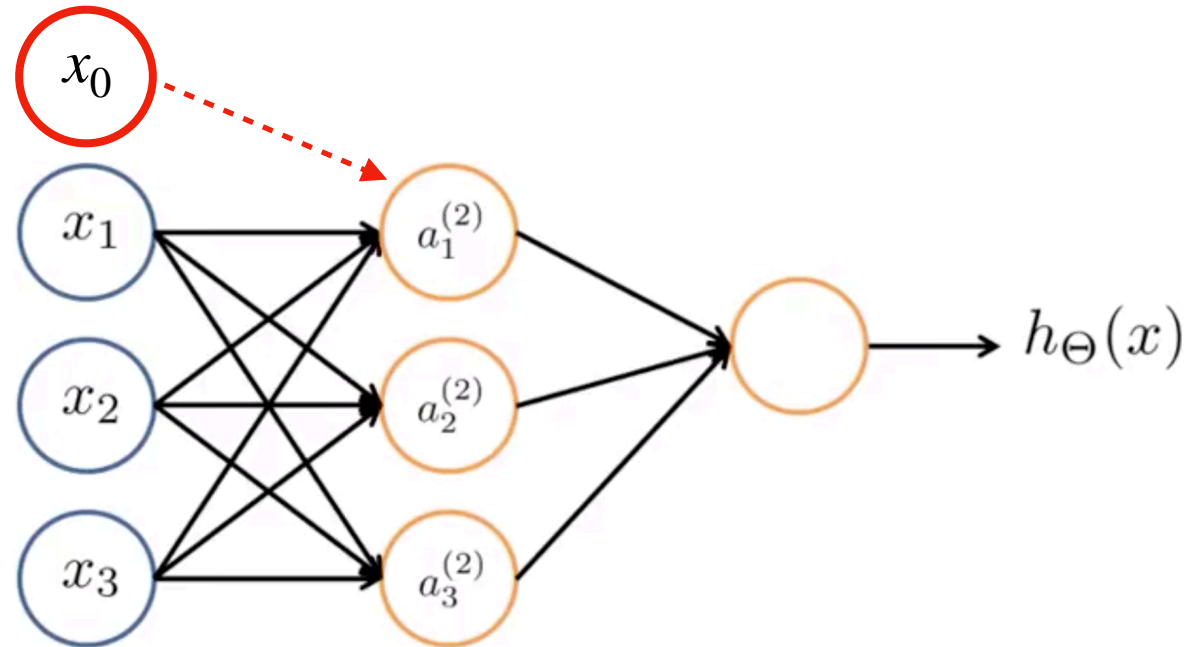
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$\xrightarrow{\text{dashed arrow}} z_3^{(2)}$

$$h_{\Theta}(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$



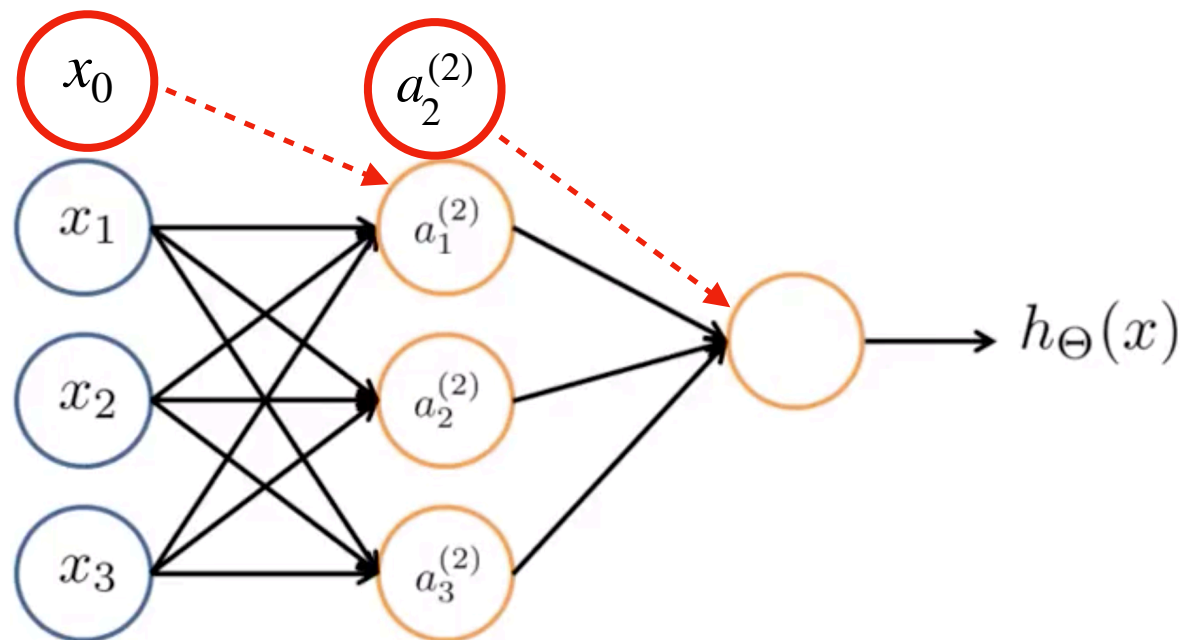
# Efficient Computation



$z_i^{(j)}$ : a weighted linear combination of the inputs that go into a neuron *e.g.*  $a_1^{(2)} = g(z_1^{(2)})$

$$\left. \begin{aligned} z_1^{(2)} &= \Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3 \\ z_2^{(2)} &= \Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3 \\ z_3^{(2)} &= \Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3 \end{aligned} \right\} \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

# Efficient Computation



$$a_0^{(2)} = 1 \implies a^{(2)} \in \mathbb{R}^4$$

$$\therefore z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$h_{\Theta}(x) = a^{(3)} = g(z^{(3)})$$

$z_i^{(j)}$ : a weighted linear combination of the inputs that go into a neuron *e.g.*  $a_1^{(2)} = g(z_1^{(2)})$

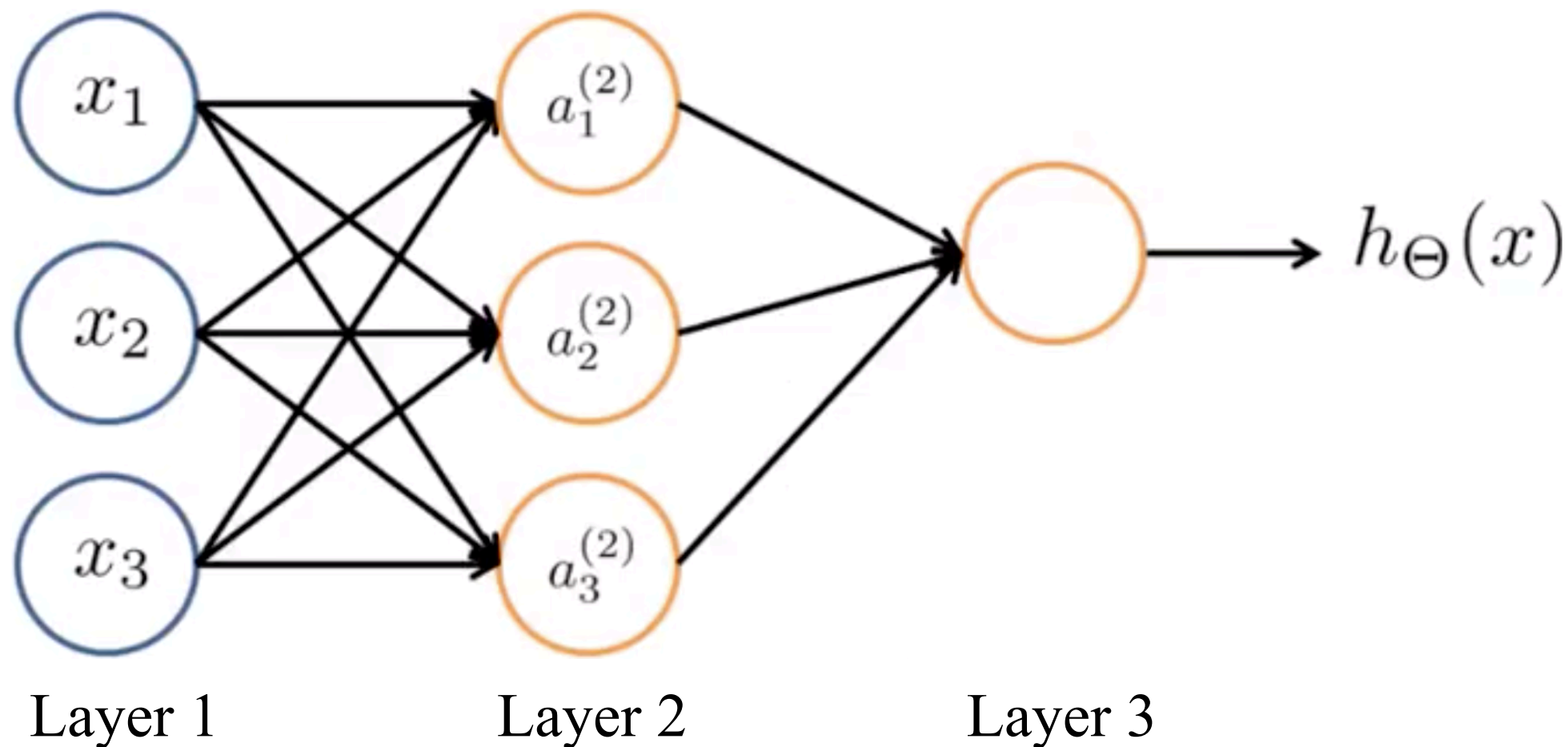
$$\left. \begin{aligned} z_1^{(2)} &= \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \\ z_2^{(2)} &= \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\ z_3^{(2)} &= \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \end{aligned} \right\} \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\text{i.e. } z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

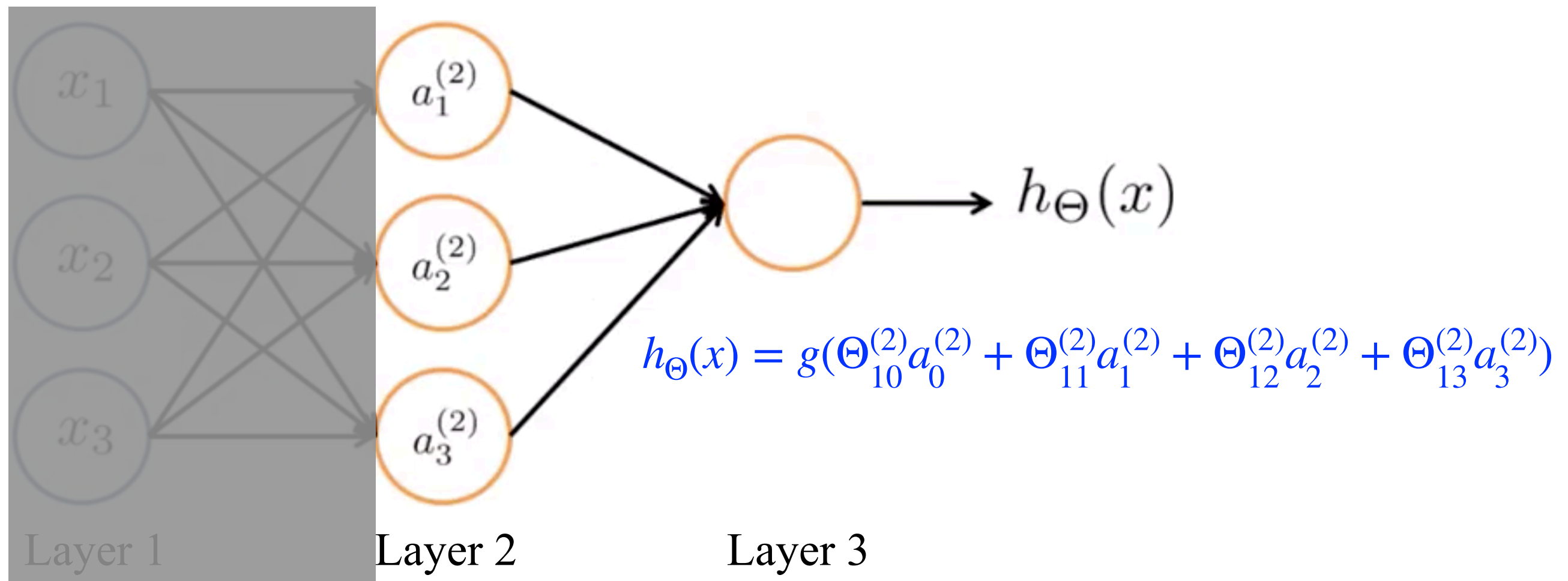
# Forward Propagation

As aforementioned, if there are no hidden units, depending on a type of an activation function, a network may become a regression setting.



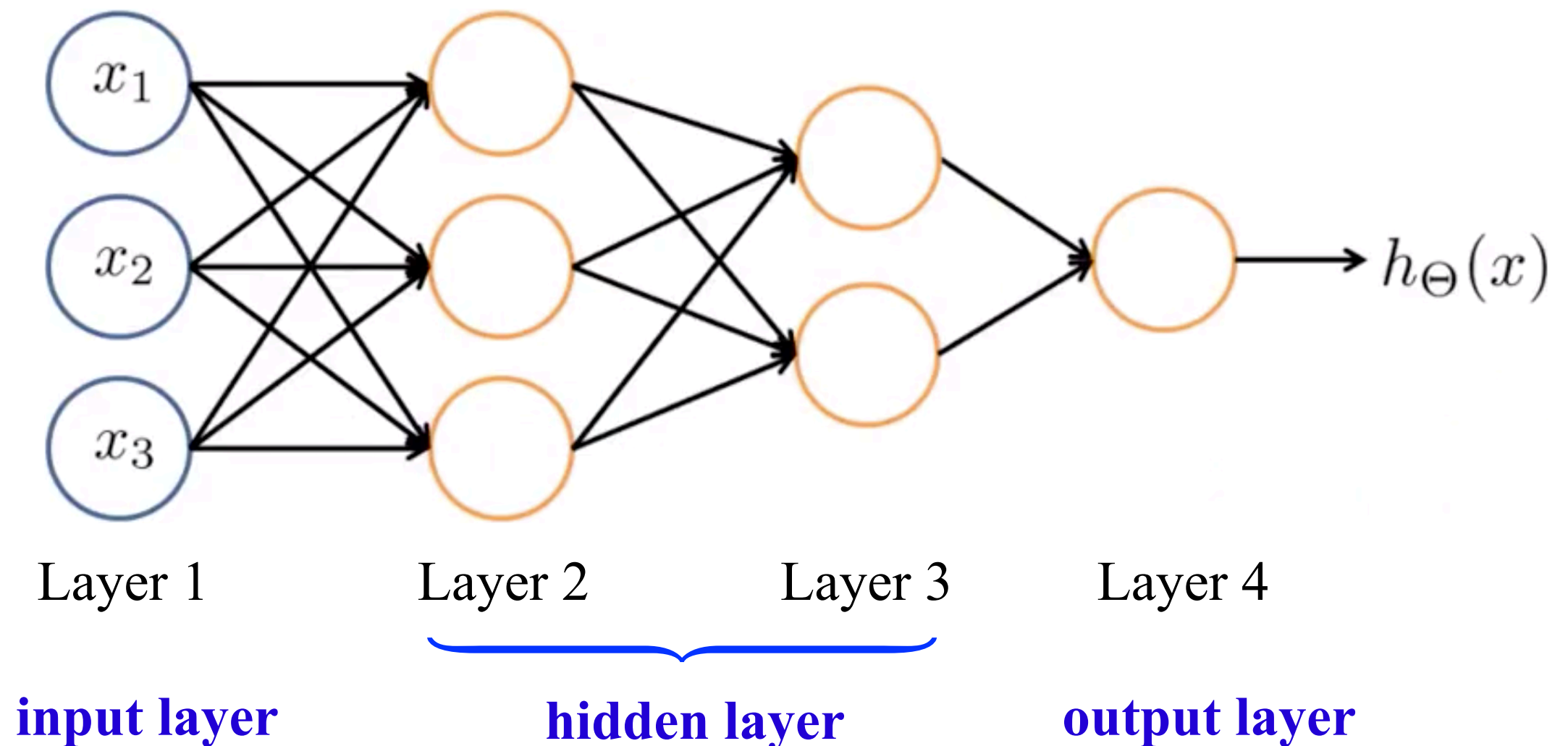
# Forward Propagation

As aforementioned, if there are no hidden units, depending on a type of an activation function, a network may become a regression setting.



# Other Network Architectures

The term ‘**architecture**’ refers to how different neurons are connected to each other (e.g. number of layers, units, type of units, connectivity between units).



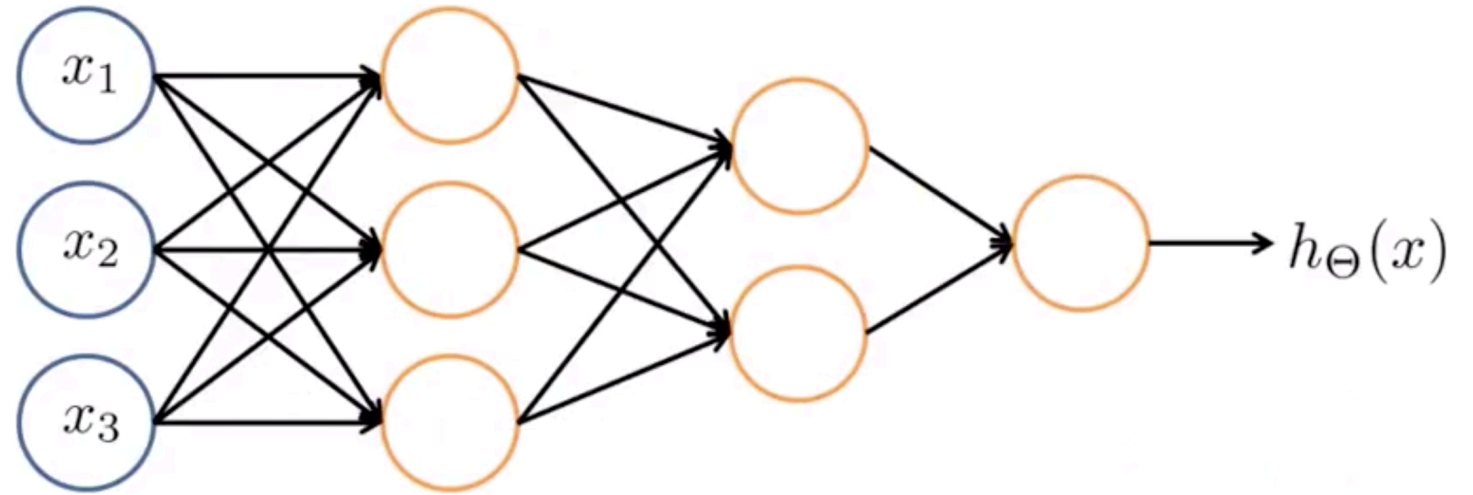
# Question

Consider the network.

Let  $a^{(1)} = x \in \mathbb{R}^{n+1}$

denote the input  
(with  $a_0^{(1)} = 1$ ).

How would you  
compute  $a^{(2)}$ ?



(i)  $a^{(2)} = \Theta^1 a^{(1)}$

(ii)  $z^{(2)} = \Theta^2 a^{(1)}; a^{(2)} = g(z^{(2)})$

(iii)  $z^{(2)} = \Theta^1 a^{(1)}; a^{(2)} = g(z^{(2)})$

(iv)  $z^{(2)} = \Theta^2 g(a^{(1)}); a^{(2)} = g(z^{(2)})$