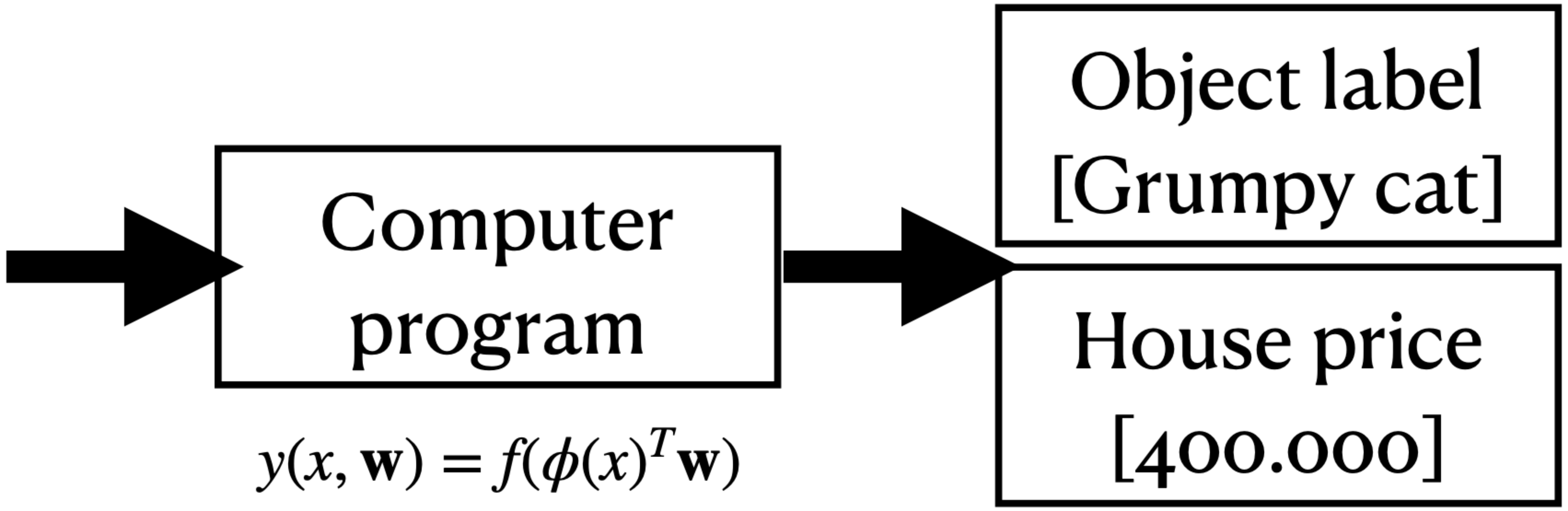
Introduction to Deep Learning

Nicolas Boulle

Setting

Goal: Learn a mapping from input to output



$y(x, \mathbf{w}) = f(\phi(x)^T \mathbf{w})$

Regression and classification

- We transform the input x into some features $\phi(x)$
- <u>Approach 1:</u> We choose the mapping $\phi(x)$ ourselves
- <u>Approach 2:</u> We let $\phi(x)$ be a **neural network** and learn the features from the data

Regression applications

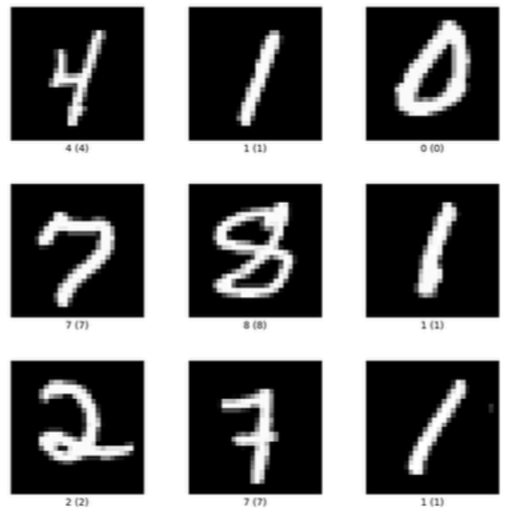
- We aim to map the input to some value
- The mapping is $y(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{M} w_j \phi_j(\mathbf{x})$
- Example:

Number of m2
Area label (notting hill, ...)
Number of rooms
Garden yes/no
...

$\rightarrow$ $[y_1]$ which represents the house price

Classification applications

- Map input to some class / label (discrete output)
- The mapping is: $y(\mathbf{x}, \mathbf{y}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right)$, where f is some nonlinear activation function.
- Example:



$$\longrightarrow \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{10} \end{bmatrix} \longrightarrow \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{10} \end{bmatrix}$$

$f(\cdot)$ = softmax that maps the output to 'probabilities'

where each c_i represents the probability that the input image is part of that class

Workflow

1. Design the neural network **architecture**

Workflow

1. Design the neural network **architecture**
2. Choose the **loss** function.

- Example: Loss$(\theta) = \dfrac{1}{N}\sum_{i=1}^{N} \|f_\theta(\mathbf{x}_i) - \mathbf{y}_i\|_2^2$

Workflow

1. Design the neural network **architecture**

2. Choose the **loss** function.

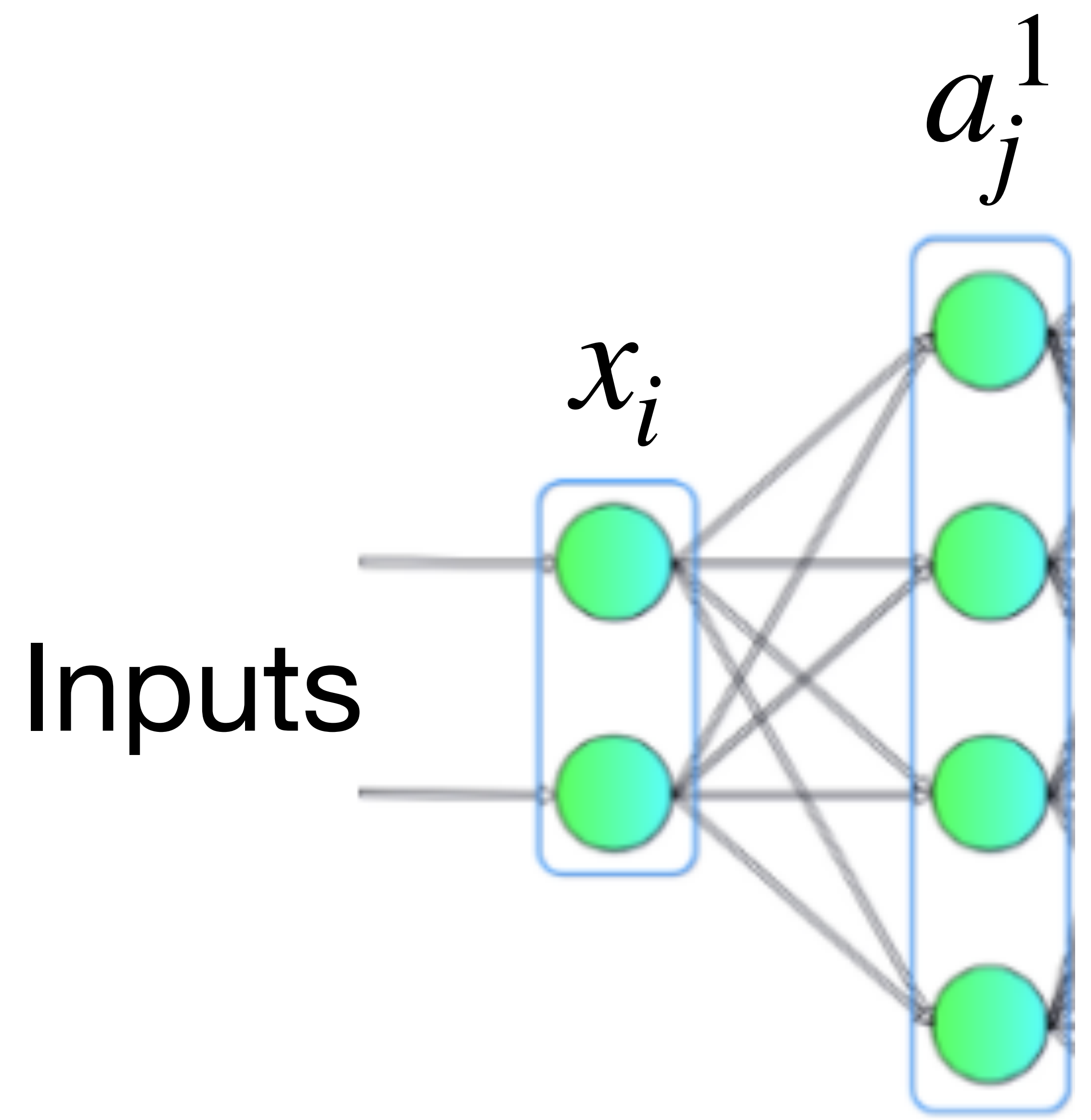
    - <u>Example:</u> Loss$(\theta) = \frac{1}{N}\sum_{i=1}^{N} ||f_\theta(\mathbf{x}_i) - \mathbf{y}_i||_2^2$

3. Train the neural network using a **gradient-based optimisation** algorithm (example: stochastic gradient descent)

Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$
- First layer: construct M linear combinations of the input variables $x_1, \ldots, x_D$ of the form:

$$a_j^1 = \sum_{i=1}^{D} w_{ji}^1 x_i + b_j^1 \text{ for } 1 \leq j \leq M$$

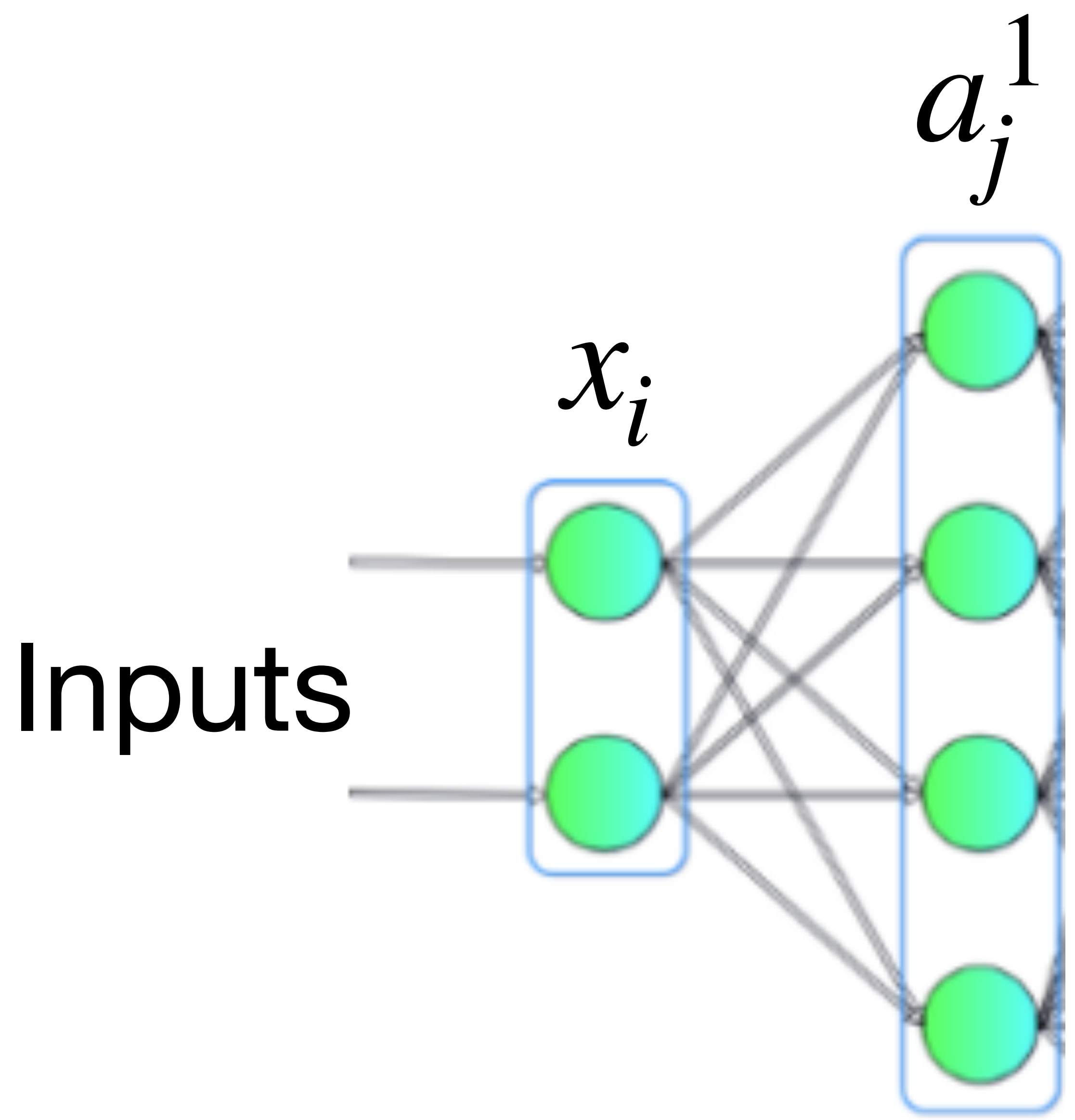


Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$
- First layer: construct M linear combinations of the input variables $x_1, \ldots, x_D$ of the form:

$$a_j^1 = \sum_{i=1}^{D} \boxed{w_{ji}^1} x_i + b_j^1 \text{ for } 1 \leq j \leq M$$

Weights

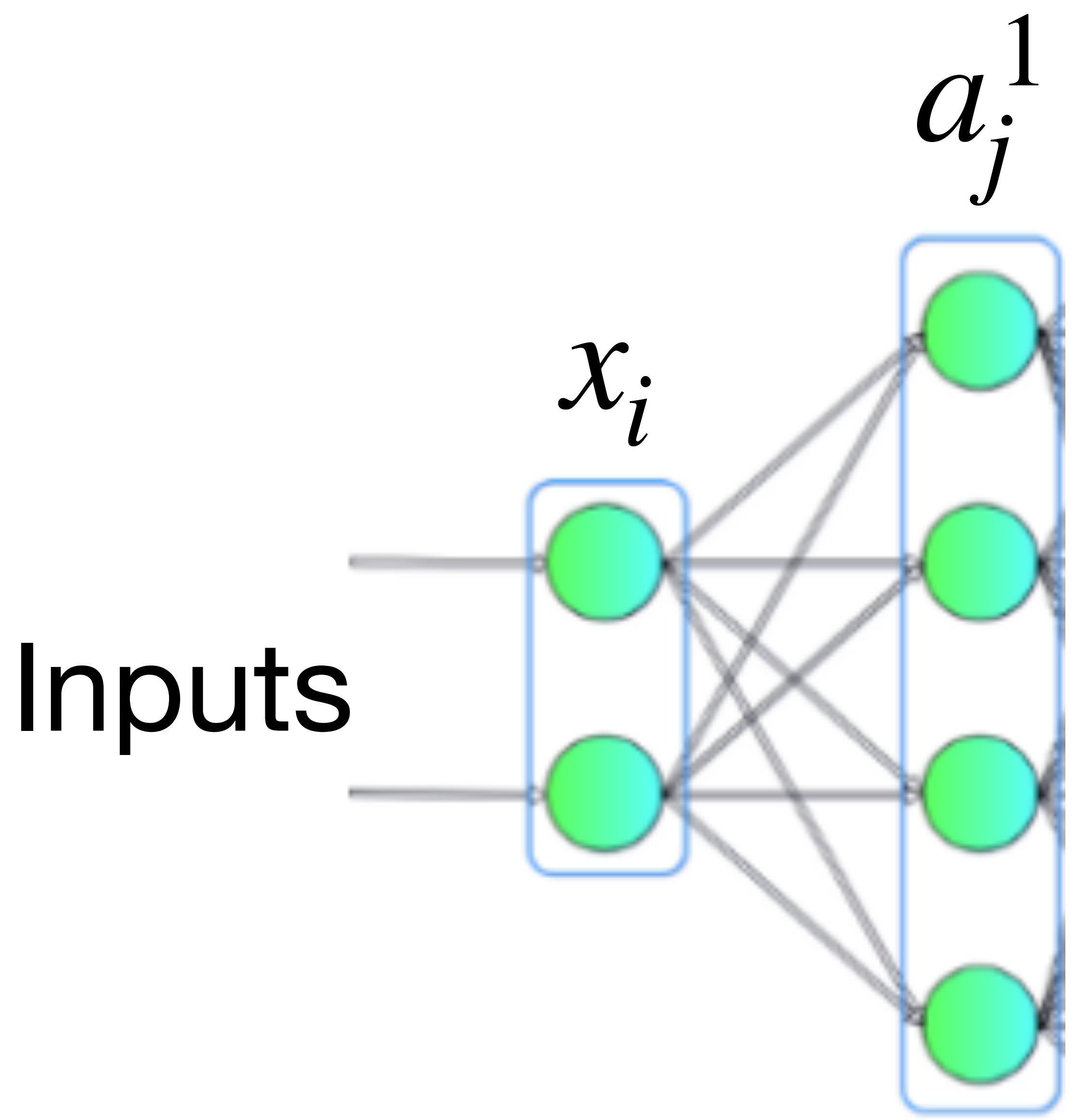


Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$
- <u>First layer:</u> construct M linear combinations of the input variables $x_1, \ldots, x_D$ of the form:

$$a_j^1 = \sum_{i=1}^{D} w_{ji}^1 x_i + \boxed{b_j^1} \text{ for } 1 \leq j \leq M$$

Bias

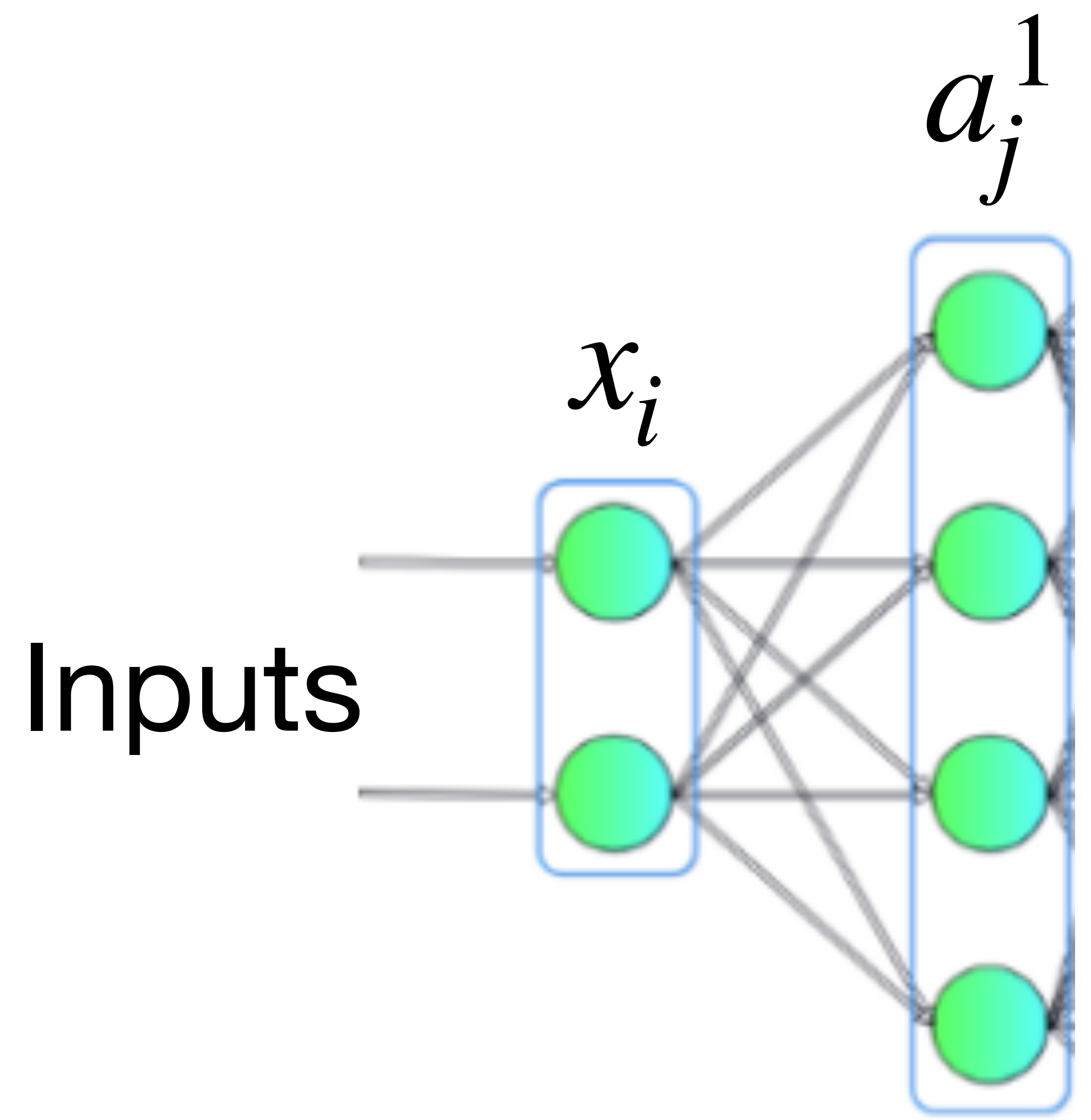


Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$
- First layer: construct M linear combinations of the input variables $x_1, \ldots, x_D$ of the form:

$$\boxed{a_j^1} = \sum_{i=1}^{D} w_{ji}^1 x_i + b_j^1 \text{ for } 1 \leq j \leq M$$

Pre-activations



Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$

- First layer: construct M linear combinations of the input variables $x_1, \ldots, x_D$ of the form:

$$a_j^1 = \sum_{i=1}^{D} w_{ji}^1 x_i + b_j^1 \text{ for } 1 \leq j \leq M$$

- Activation: We transform the a's using a **nonlinear activation function** σ

$$\boxed{h_j^1} = \sigma(a_j^1)$$

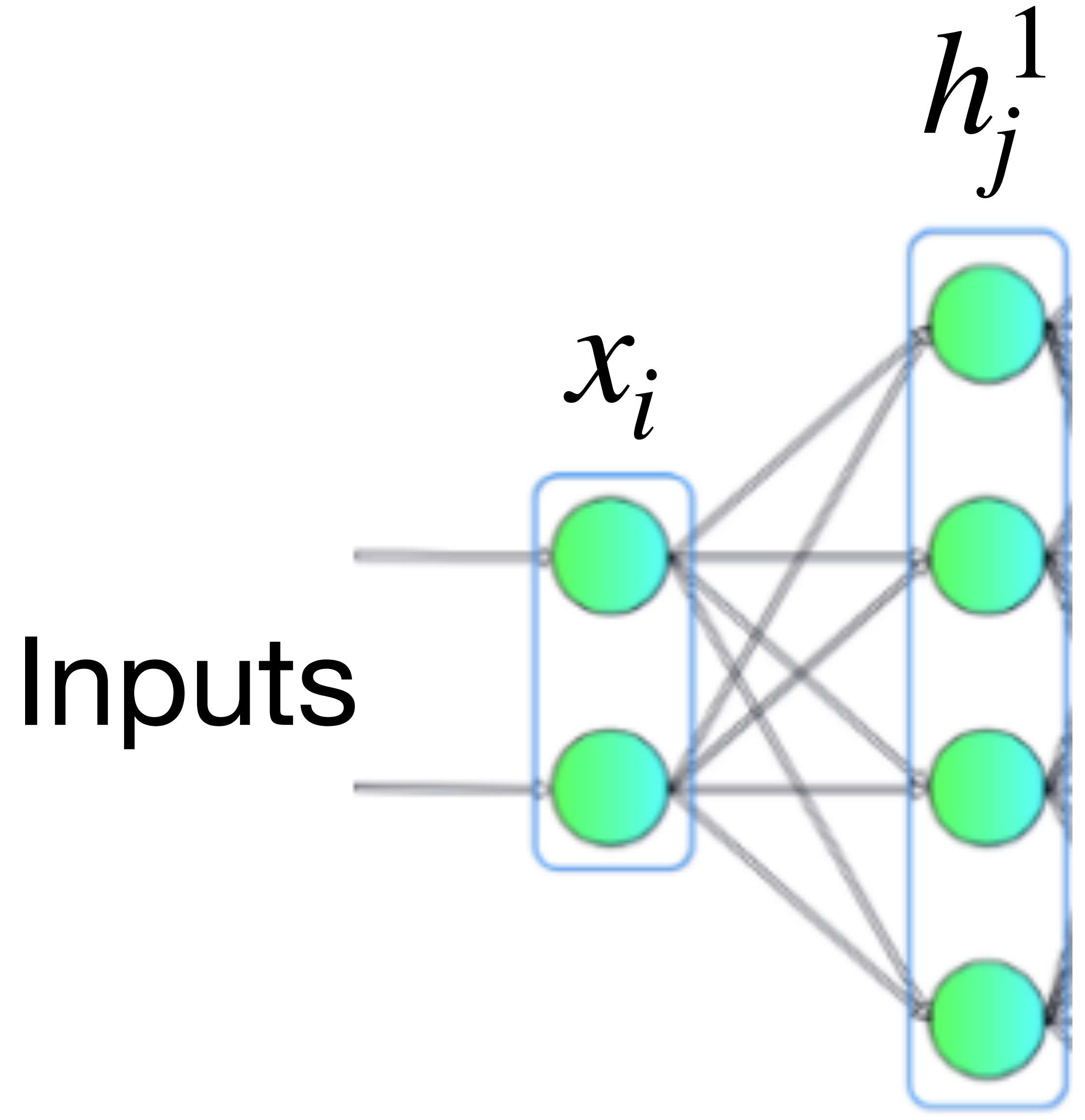
Hidden units

Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$

- Activation: We transform the a's using a **nonlinear activation function** σ

$$h_j^1 = \sigma(a_j^1)$$



Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$
- Second layer: construct K linear combinations of the hidden variables $h_1^1, \ldots, h_M^1$ of the form:

Number of outputs

$$a_j^2 = \sum_{i=1}^{M} w_{ji}^2 h_i^1 + b_j^2 \text{ for } 1 \leq j \leq K$$

Basic neural network model

- Suppose the input $\mathbf{x} = [x_1 \quad \cdots \quad x_D]^\top$

- Second layer: construct K linear combinations of the hidden variables $h_1^1, \ldots, h_M^1$ of the form:

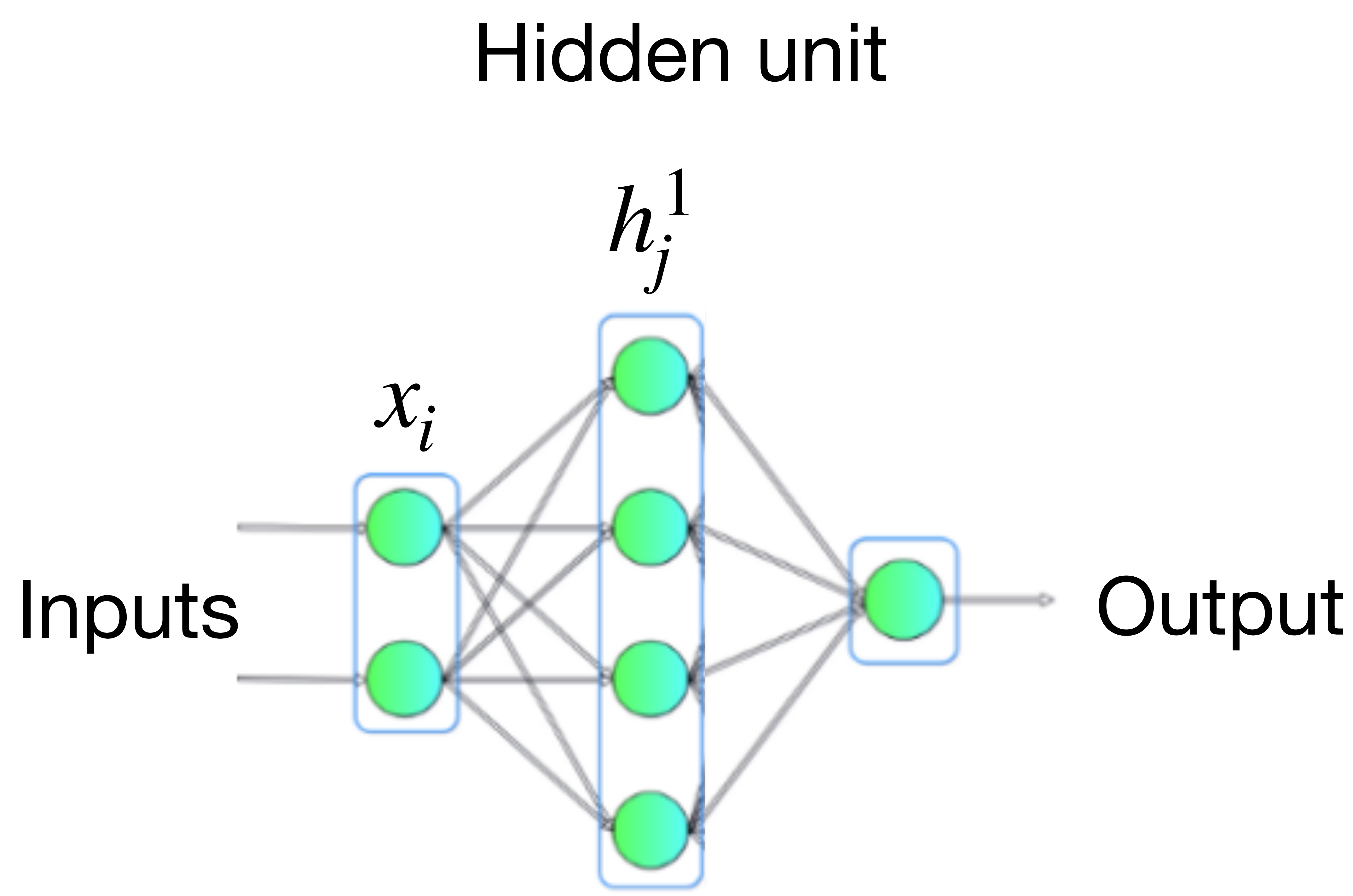
$$a_j^2 = \sum_{i=1}^{M} w_{ji}^2 h_i^1 + b_j^2 \text{ for } 1 \leq j \leq K$$

- Final activation: We transform the a's using a **nonlinear activation function** σ

$$y_j^1 = \sigma(a_j^1)$$

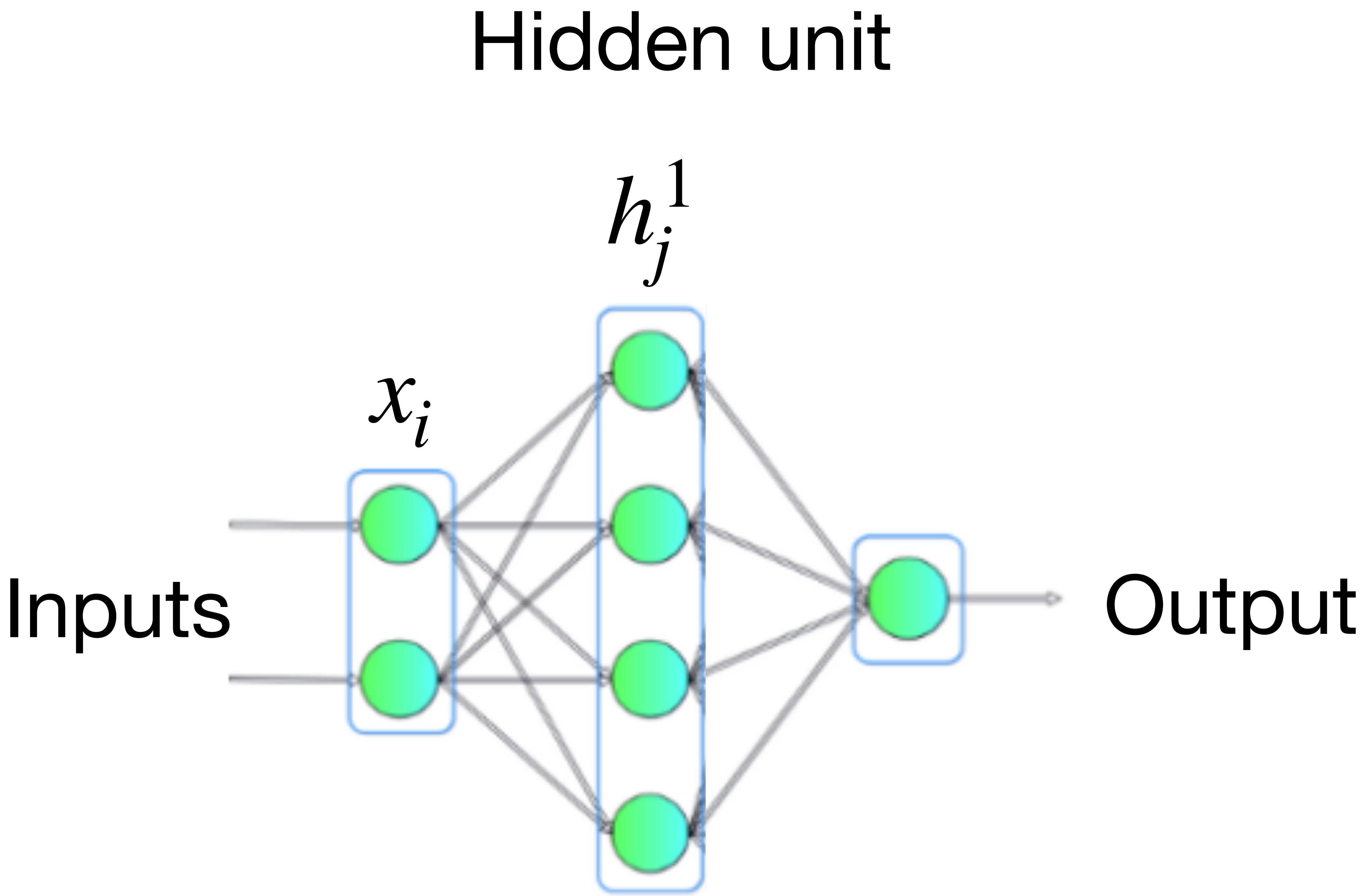
Basic neural network model

Visualisation



Basic neural network model

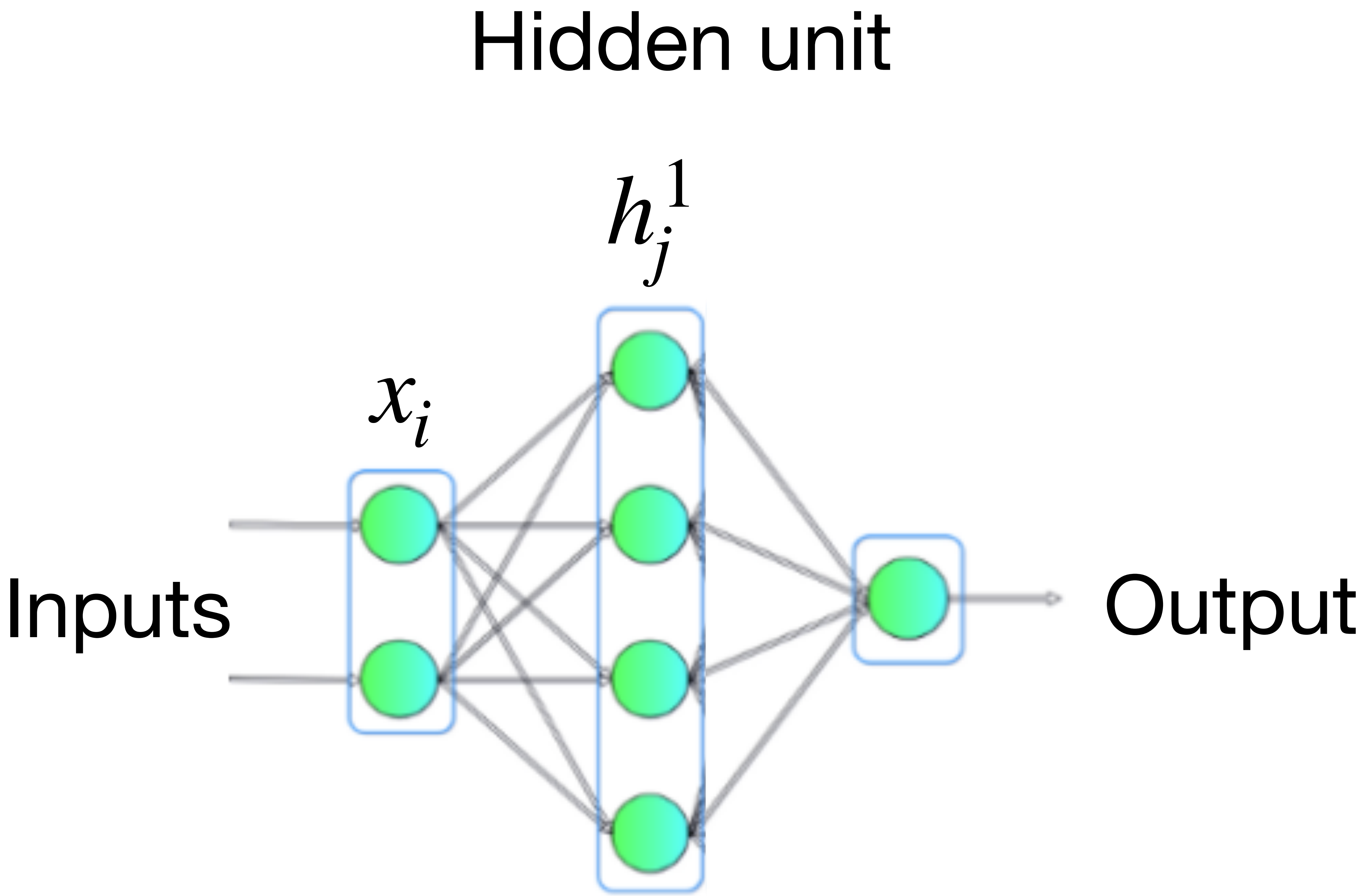
Visualisation



1. Linear transformation by pre-multiplying with a weight matrix W^1 and adding a bias vector b^1

Basic neural network model

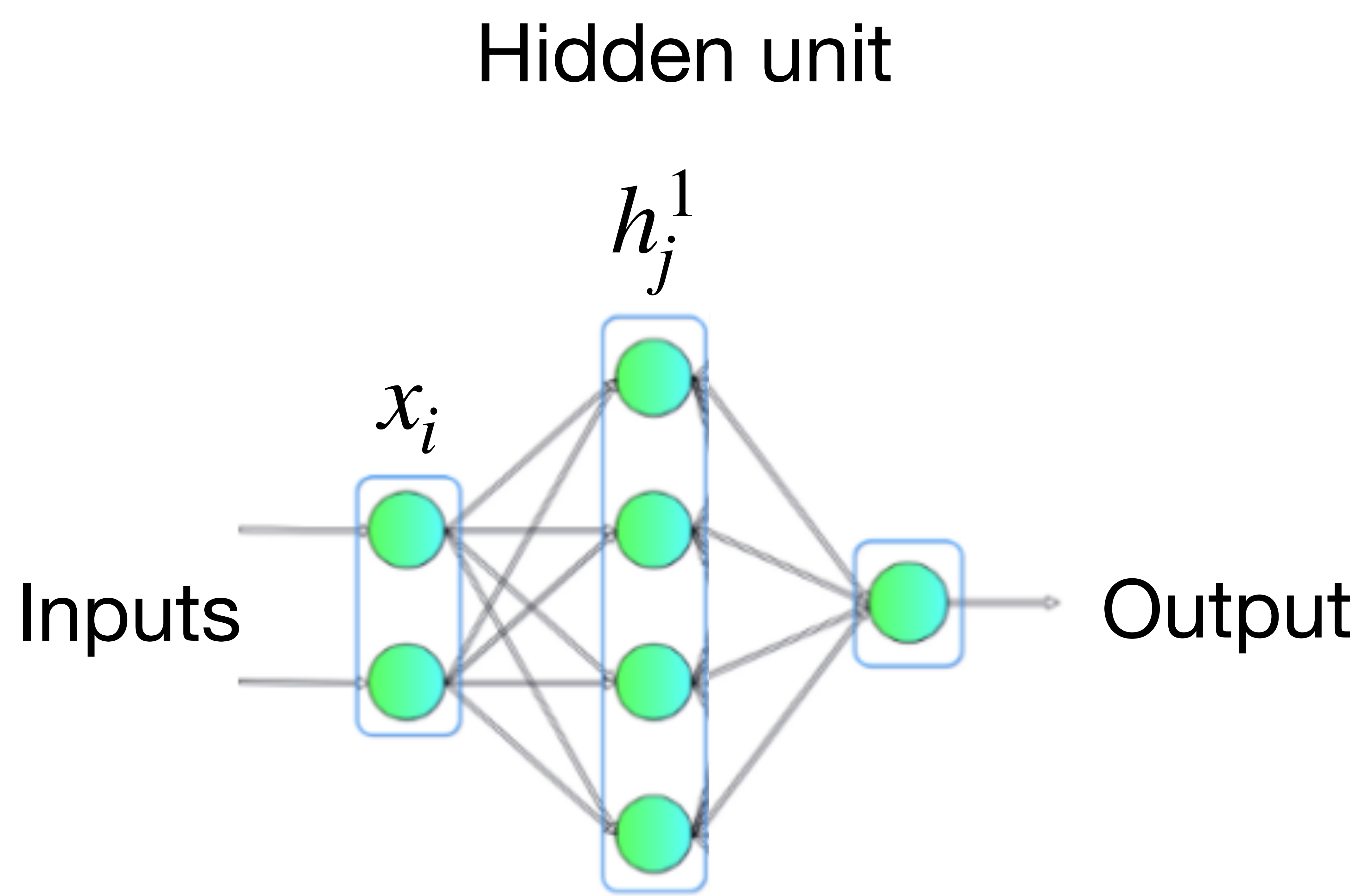
Visualisation



1. Linear transformation by pre-multiplying with a weight matrix W^1 and adding a bias vector b^1
2. Pass through nonlinear activation function

Basic neural network model

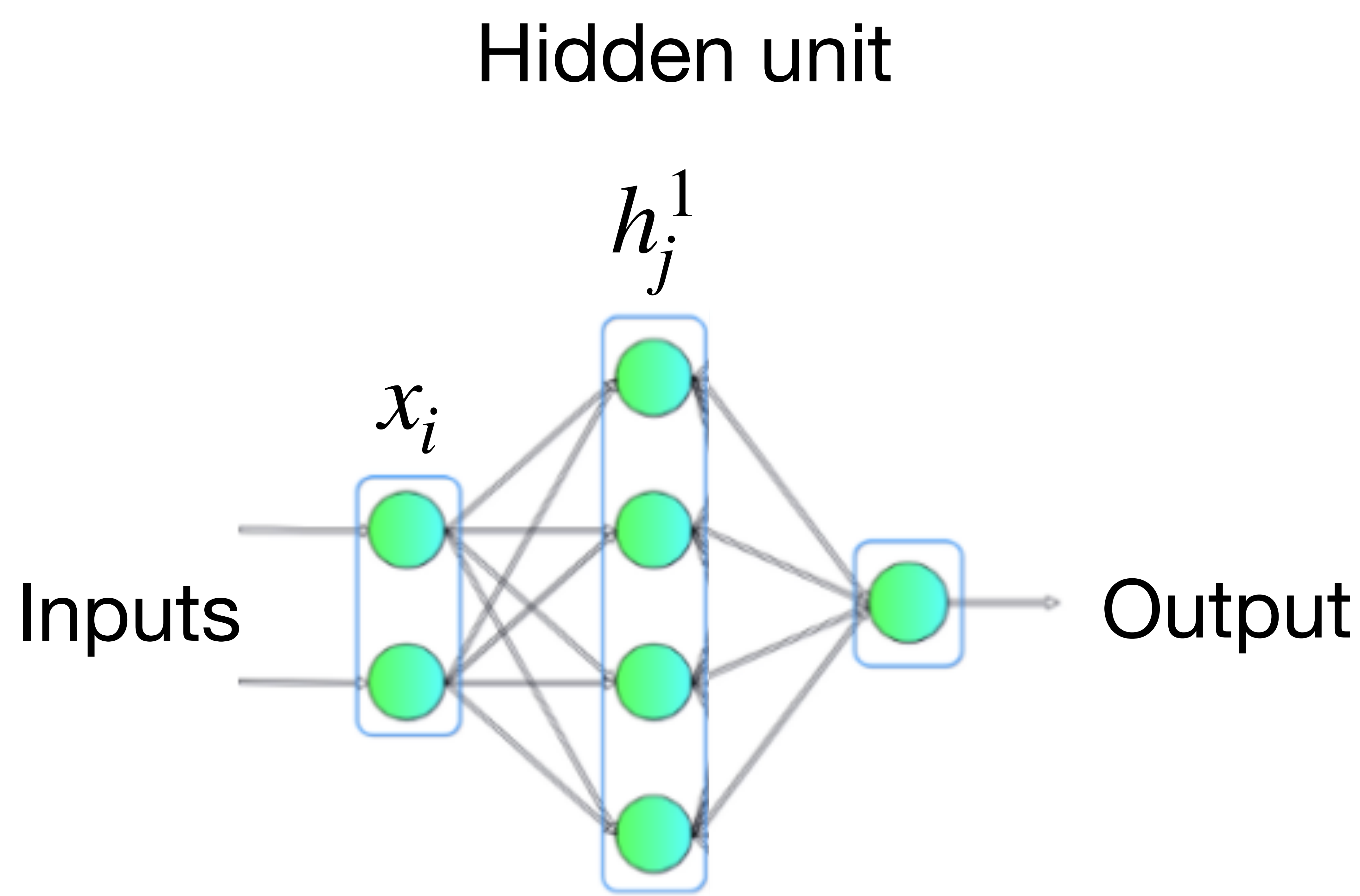
Visualisation



1. Linear transformation by pre-multiplying with a weight matrix W^1 and adding a bias vector b^1
2. Pass through nonlinear activation function
3. Linear transformation with W^2 and b^2

Basic neural network model

Visualisation



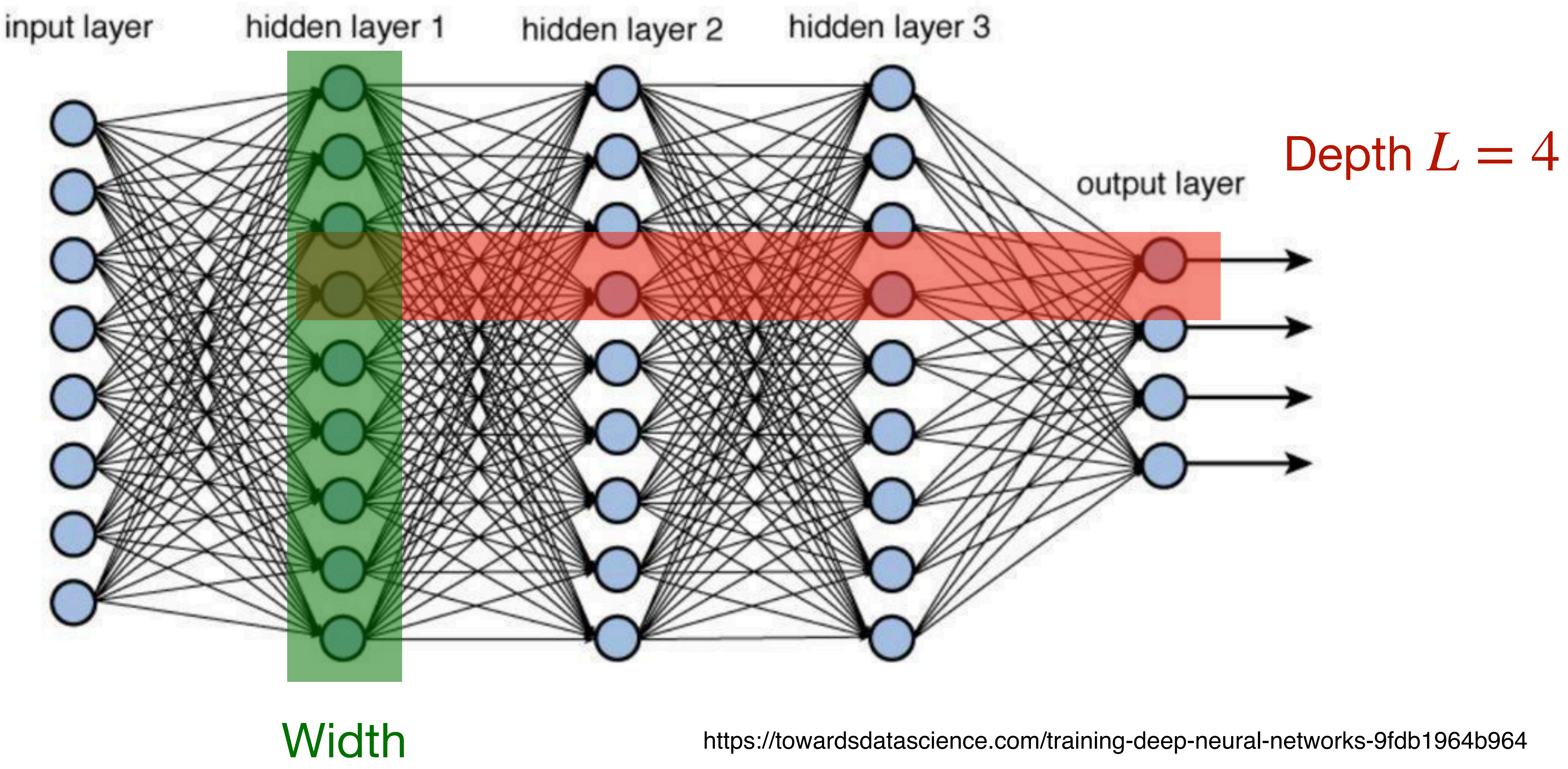
1. Linear transformation by pre-multiplying with a weight matrix W^1 and adding a bias vector b^1

2. Pass through nonlinear activation function

3. Linear transformation with W^2 and b^2

4. Pass through nonlinear activation function (if needed)

Deep neural network



https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

PyTorch

- Machine learning library with Python interface to implement neural networks

- Open source software developed by Meta https://pytorch.org/

- All operations are performed on objects torch.Tensor that are multidimensional matrices

A neural network in PyTorch

- A network with L hidden layers in torch:

```
def forward(self, input):
    m = torch.nn.Linear(dim, nr_hidden)
    x = torch.nn.flatten(input)
    x = m(x)
    for layer in range(L):
        m = torch.nn.Linear(nr_hidden, nr_hidden)
        x = m(x)
        x = torch.nn.functional.relu(x)
    m = torch.nn.linear(nr_hidden, nr_output)
    output = m(x)
    return output
```

Training neural networks

- We have a dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$

Training neural networks

- We have a dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$

- We defined a neural network $\mathbf{x} \mapsto f(\mathbf{x}; \theta)$, where $\theta \in \mathbb{R}^d$ contains the parameters (weights and biases) of the network

Training neural networks

- We have a dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$
- We defined a neural network $\mathbf{x} \mapsto f(\mathbf{x}; \theta)$, where $\theta \in \mathbb{R}^d$ contains the parameters (weights and biases) of the network
- We train the neural networks by optimising a certain loss or cost function, such as the mean-squared error:

$$\min_{\theta \in \mathbb{R}^d} L(\theta) = \min_{\theta \in \mathbb{R}^d} \frac{1}{m} \sum_{i=1}^{m} \|\mathbf{y}_i - f(\mathbf{x}_i; \theta)\|_2^2$$

Optimisation algorithm

- We use a gradient descent algorithm to minimise the cost function:

- Start with initial parameters $\theta^{(0)} \in \mathbb{R}^d$

Optimisation algorithm

- We use a gradient descent algorithm to minimise the cost function:

- Start with initial parameters $\theta^{(0)} \in \mathbb{R}^d$

- Update the parameters of the networks as

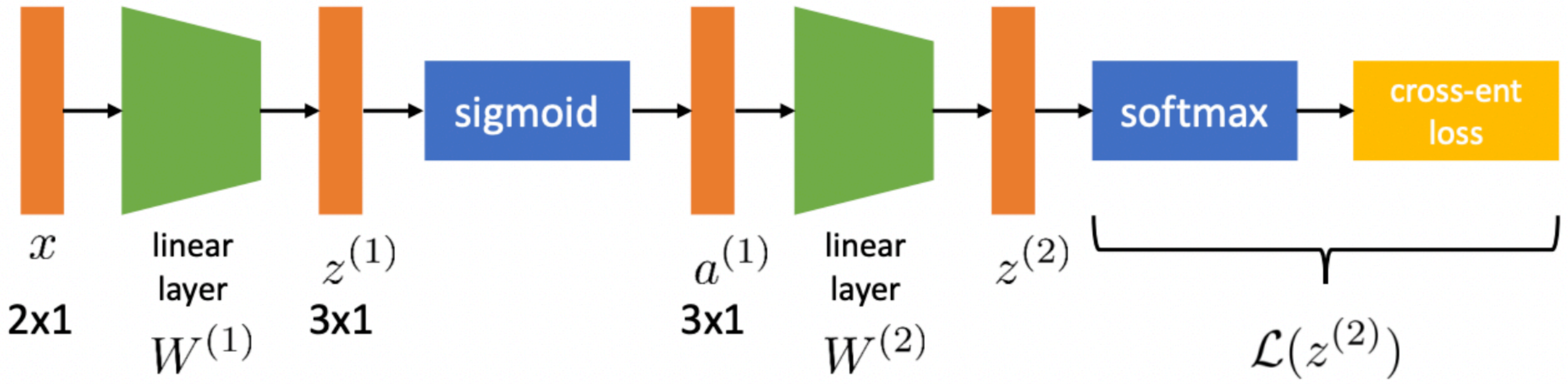
$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_\theta L(\theta^{(k)})$$

$\eta > 0$ is called the learning rate

Backpropagation algorithm

- How do we compute the gradient $\nabla_\theta L(\theta)$?

- We use the chain rule (or backpropagation algorithm):



$$L(\theta) = L([W^1, W^2])$$

$$\frac{d\mathscr{L}}{dW^1} = \frac{d\mathscr{L}}{dz^2}\frac{dz^2}{da^1}\frac{da^1}{dz^1}\frac{dz^1}{dW^1}$$

$$\frac{d\mathscr{L}}{dW^2} = \frac{d\mathscr{L}}{dz^2}\frac{dz^2}{dW^2}$$

Backpropagation algorithm

In PyTorch:

```
# Set gradients to zero
optimizer.zero_grad()
```

Build the model $f(\mathbf{x};\theta)$ and remember operations (similar to pyadjoint)

```
# Forward pass
output = model(x)
```

Compute the cost function $L(\theta)$

```
# Calculate loss
loss = criterion(output, y)
```

Calculate the gradient $\nabla_\theta L(\theta)$

```
# Backward pass
loss.backward()
```

Update the parameters:

$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$

```
# Update weights
optimizer.step()
```