Artificial Neural Networks

CS 486/686: Introduction to Artificial Intelligence

Introduction

Machine learning algorithms can be viewed as approximations of functions that describe the data

In practice, the relationships between input and output can be **extremely** complex.

We want to:

- Design methods for learning arbitrary relationships
- Ensure that our methods are efficient and do not overfit the data

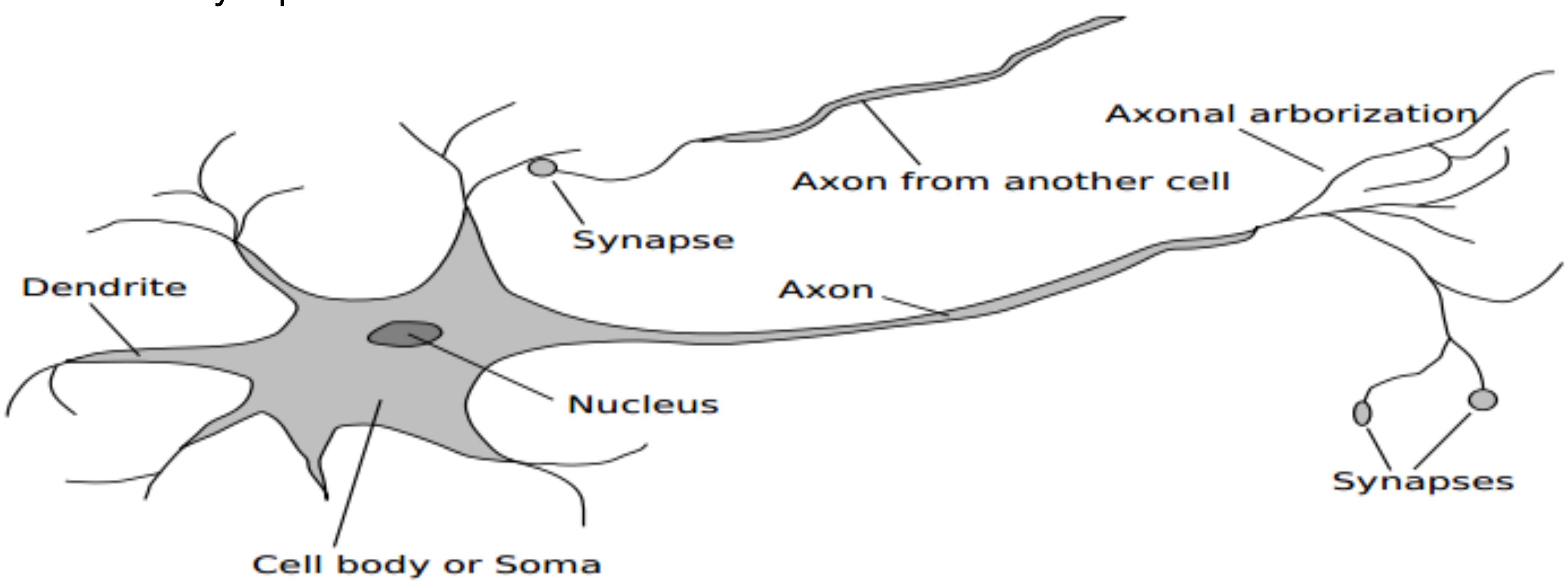
Artificial Neural Nets

Idea: The humans can often learn complex relationships very well.

Maybe we can simulate human learning?

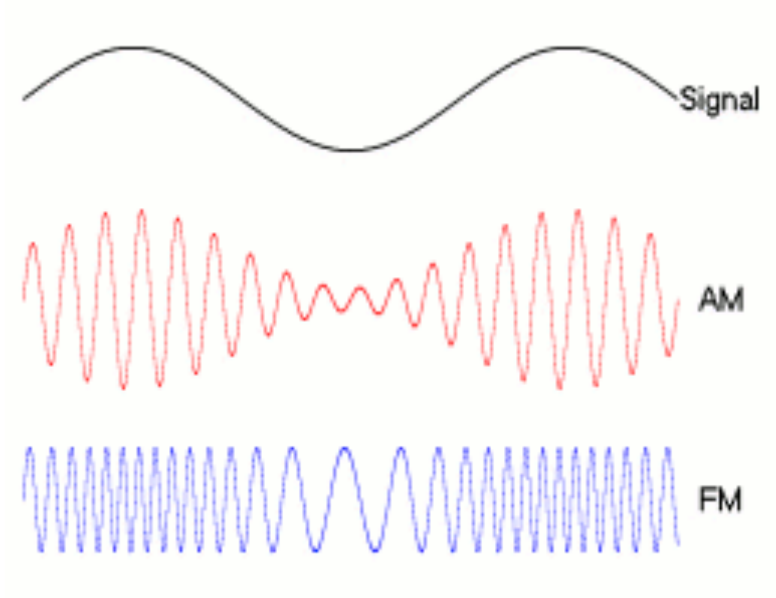
Human Brains

- A brain is a set of densely connected neurons.
- A neuron has several parts:
  - Dendrites: Receive inputs from other cells
  - Soma: Controls activity of the neuron
  - Axon: Sends output to other cells
  - Synapse: Links between neurons



Human Brains

- Neurons have two states
  - Firing, not firing
- All firings are the same



- Rate of firing communicates information (FM)
- Activation passed via chemical signals at the synapse between firing neuron's axon and receiving neuron's dendrite
- Learning causes changes in how efficiently signals transfer across specific synaptic junctions.

Artificial Brains?

- Artificial Neural Networks are based on very early models of the neuron.
- Better models exist today, but are usually used theoretical neuroscience, not machine learning

Artificial Brains?

- An artificial Neuron (McCulloch and Pitts 1943)

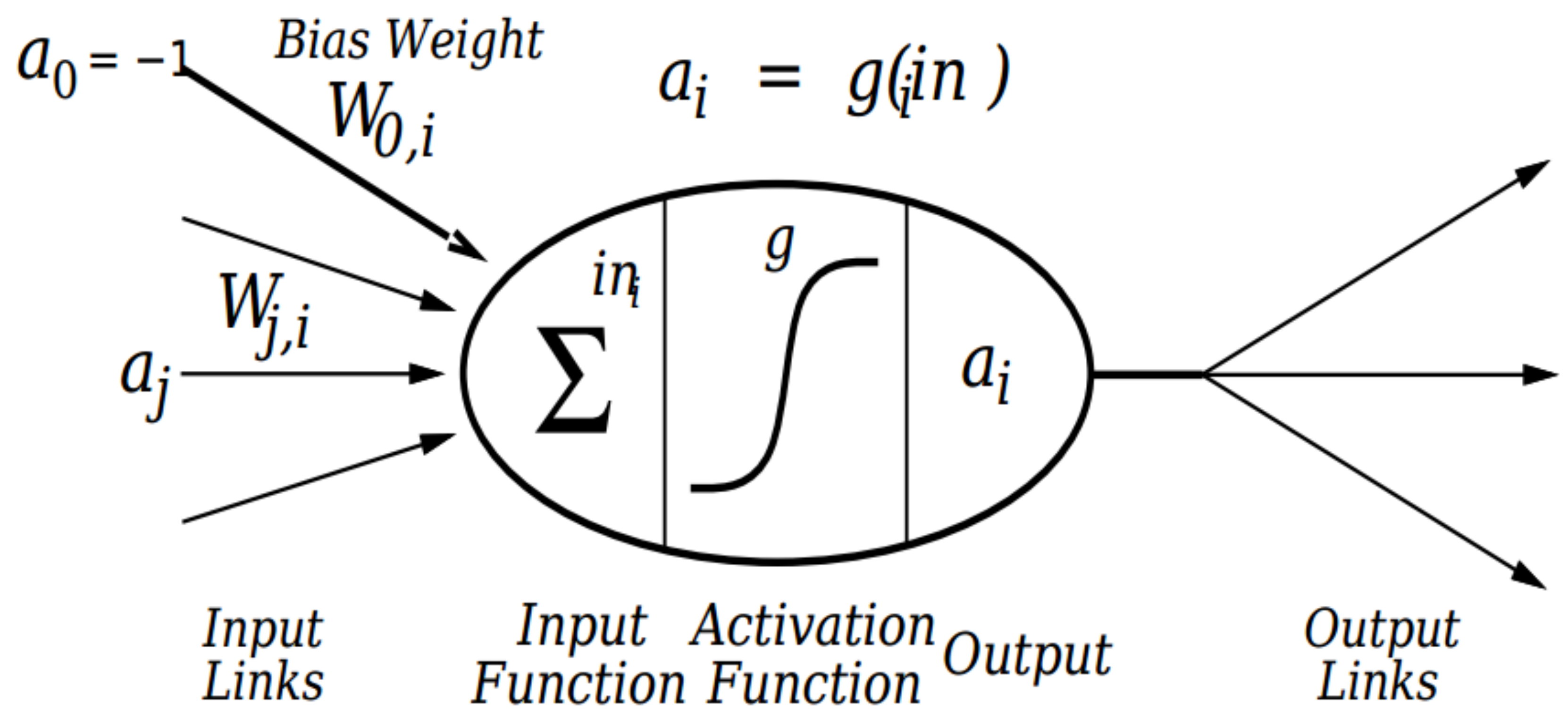
Link~ Synapse

Weight ~ Efficiency

Input Fun.~ Dendrite

Activation Fun.~ Soma

Output = Fire or not



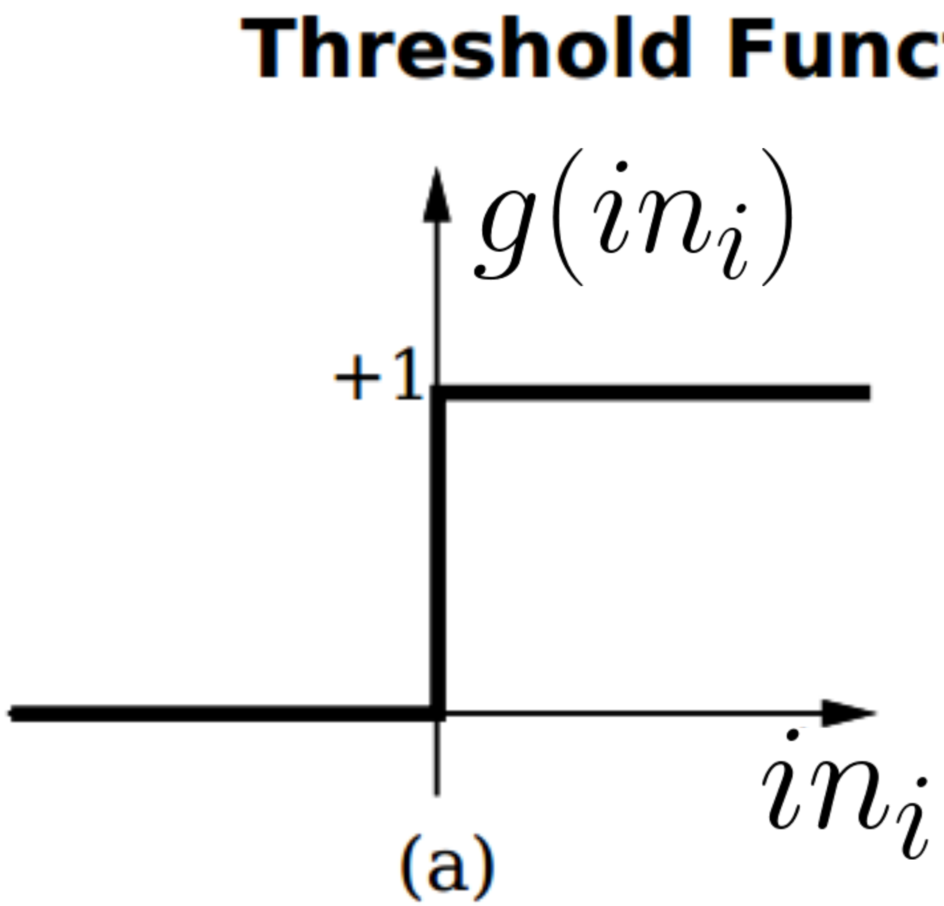
Artificial Neural Nets

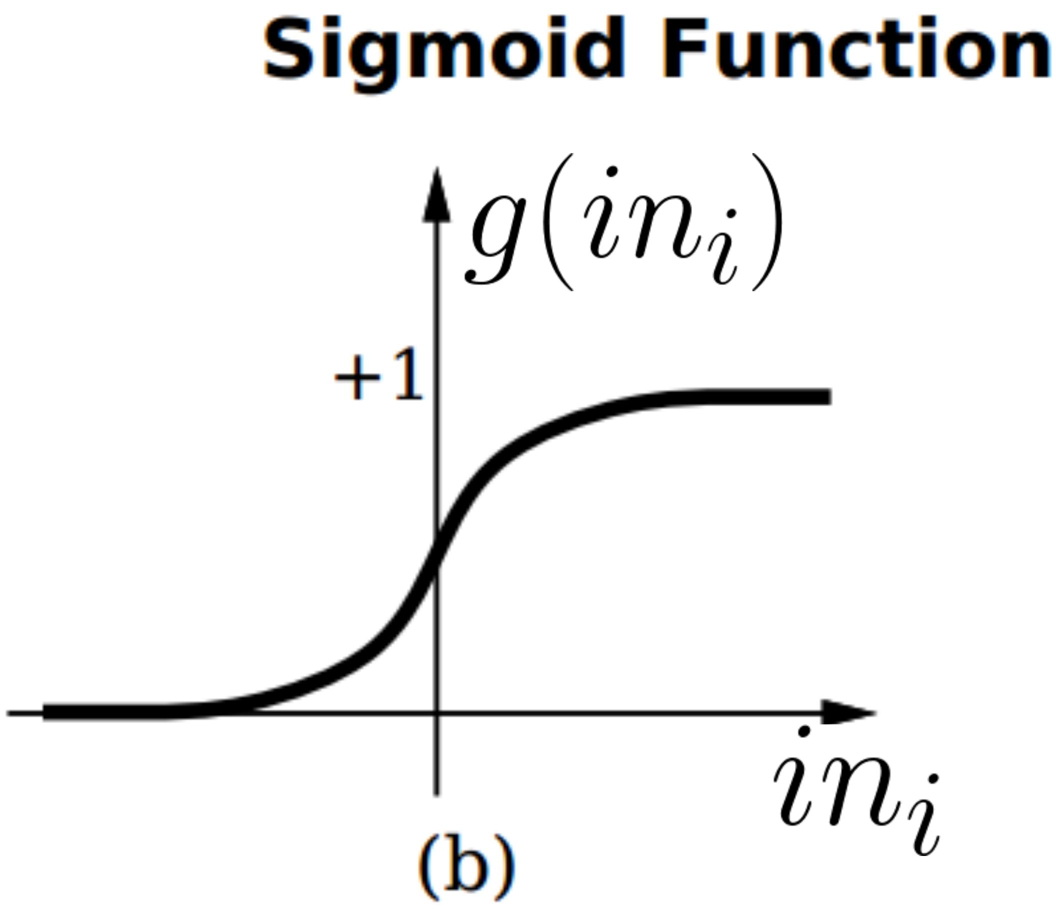
- Collection of simple artificial neurons.
- Weights $W_{i,j}$ denote strength of connection from i to j
- Input function: $in_i = \sum_j W_{i,j} \times a_j$
- Activation Function: $a_i = g(in_i)$

Activation Function

- Activation Function: $a_i = g(in_i)$
- Should be non-linear (otherwise, we just have a linear equation)
- Should mimic firing in real neurons
  - Active ($a_i \sim 1$) when the "right" neighbors fire the right amounts
  - Inactive ($a_i \sim 0$) when fed "wrong" inputs

Common Activation Functions



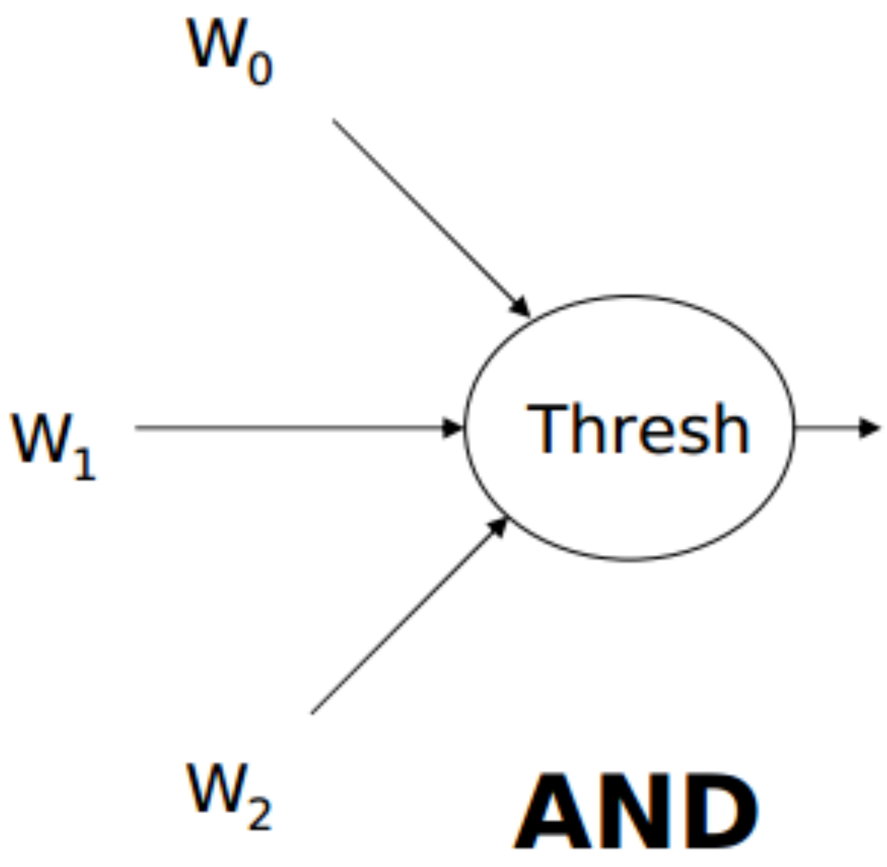


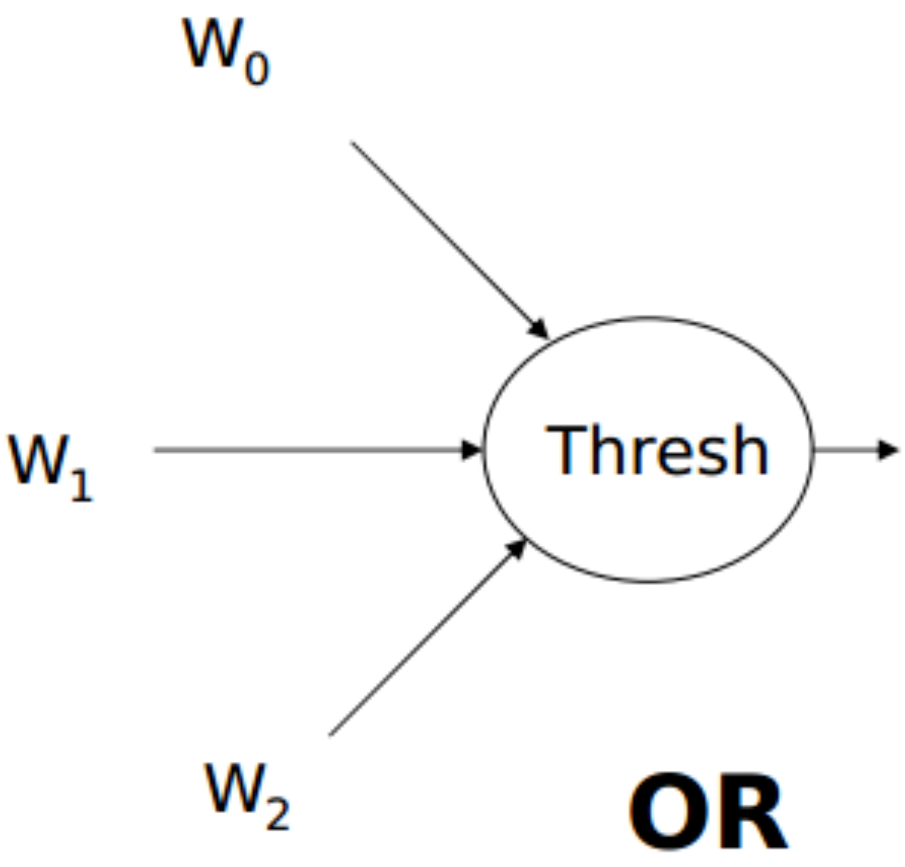
Weights determine threshold

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

Logic Gates

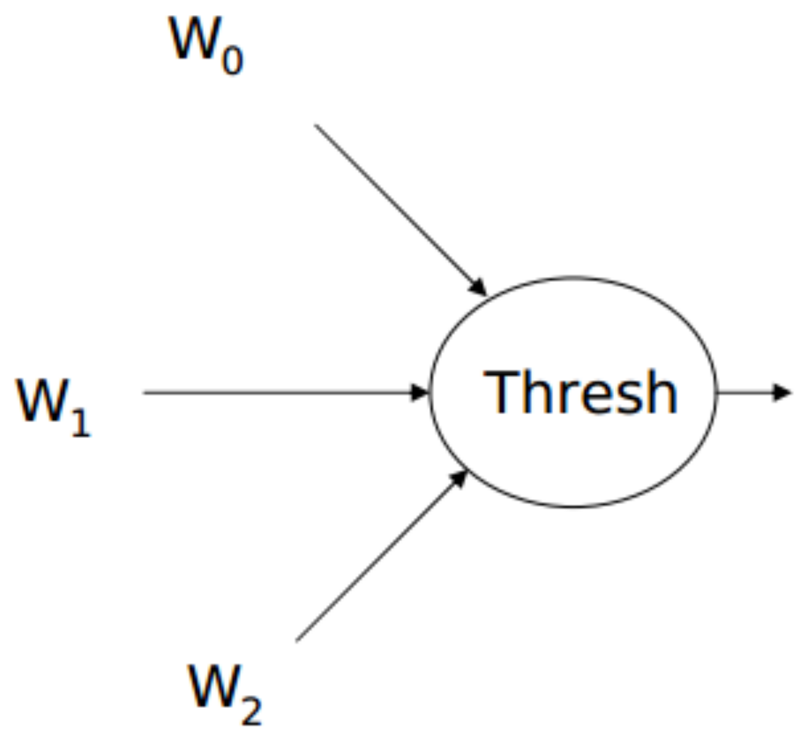
It is possible to construct a universal set of logic gates using the neurons described (McCulloch and Pitts 1943)





Logic Gates

It is possible to construct a universal set of logic gates using the neurons described (McCulloch and Pitts 1943)

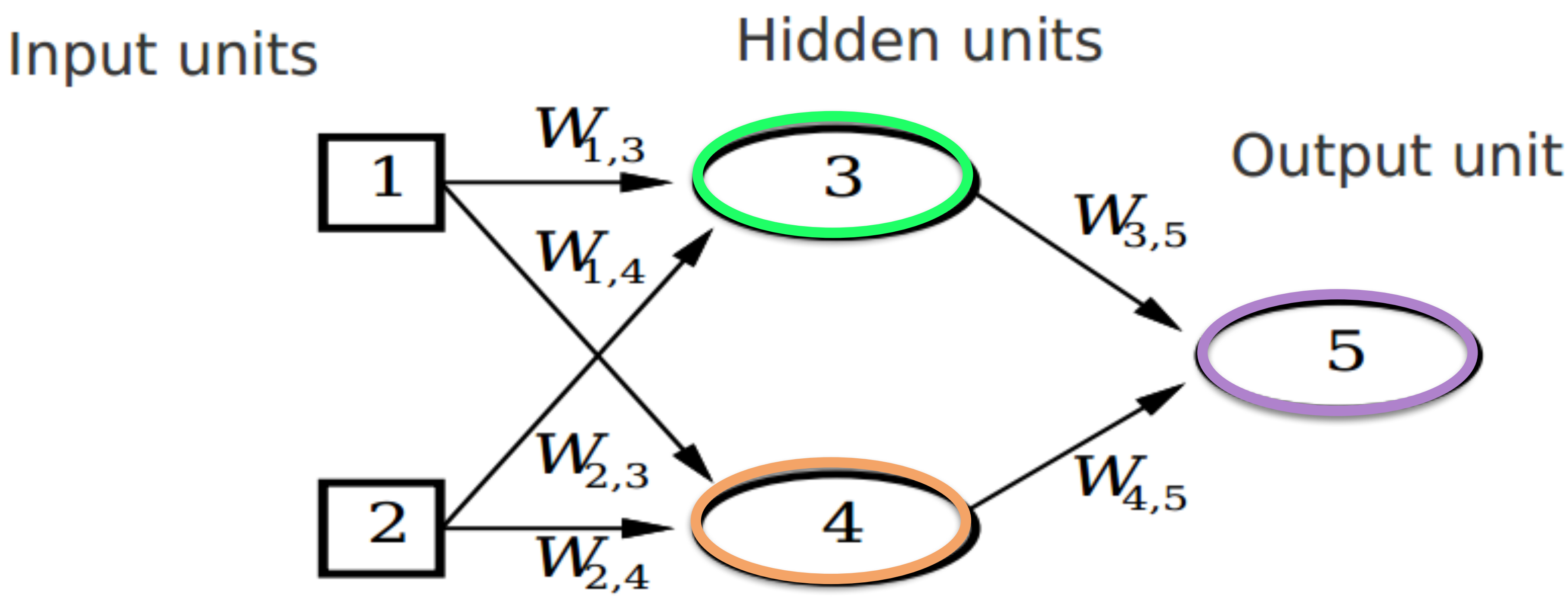


NOT

Network Structure

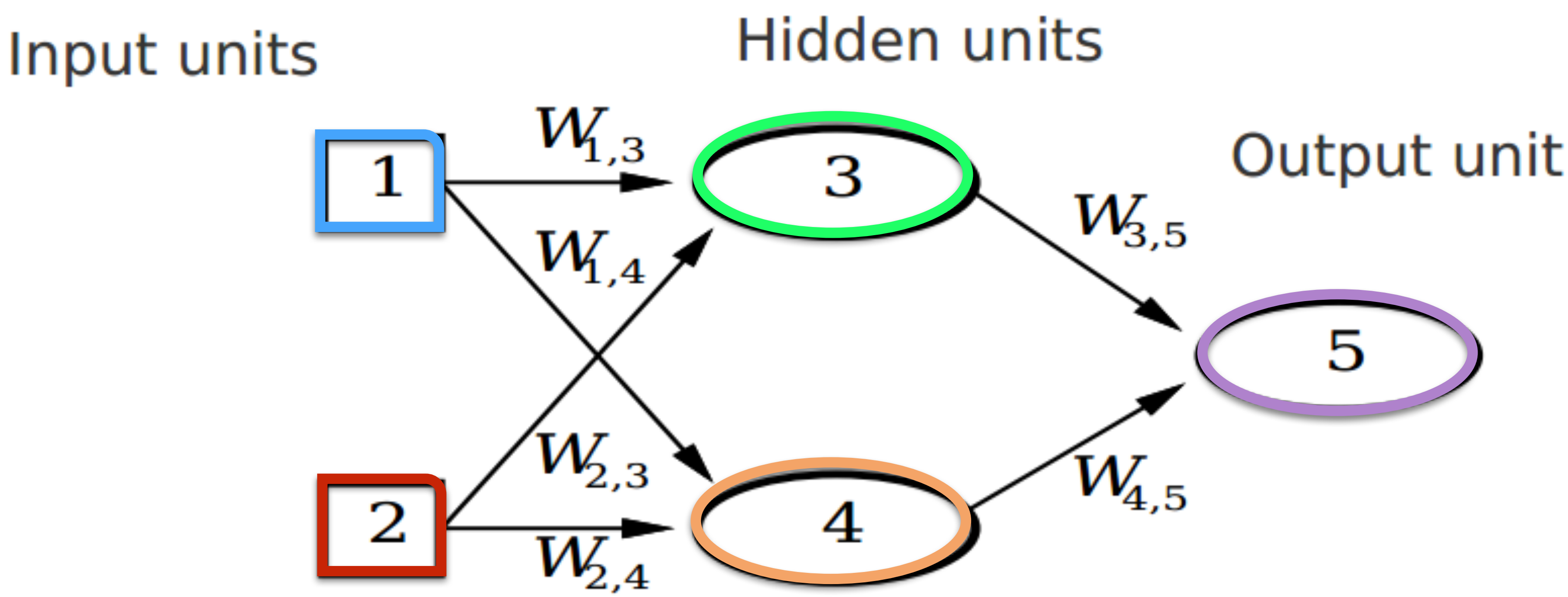
- Feed-forward ANN
  - Direct acyclic graph
  - No internal state: maps inputs to outputs.
- Recurrant ANN
  - Directed cyclic graph
  - Dynamical system with an internal state
  - Can remember information for future use

Example



$$a_5 = g(W_{3,5} \cdot a_5 + W_{4,5} \cdot a_4)$$

Example

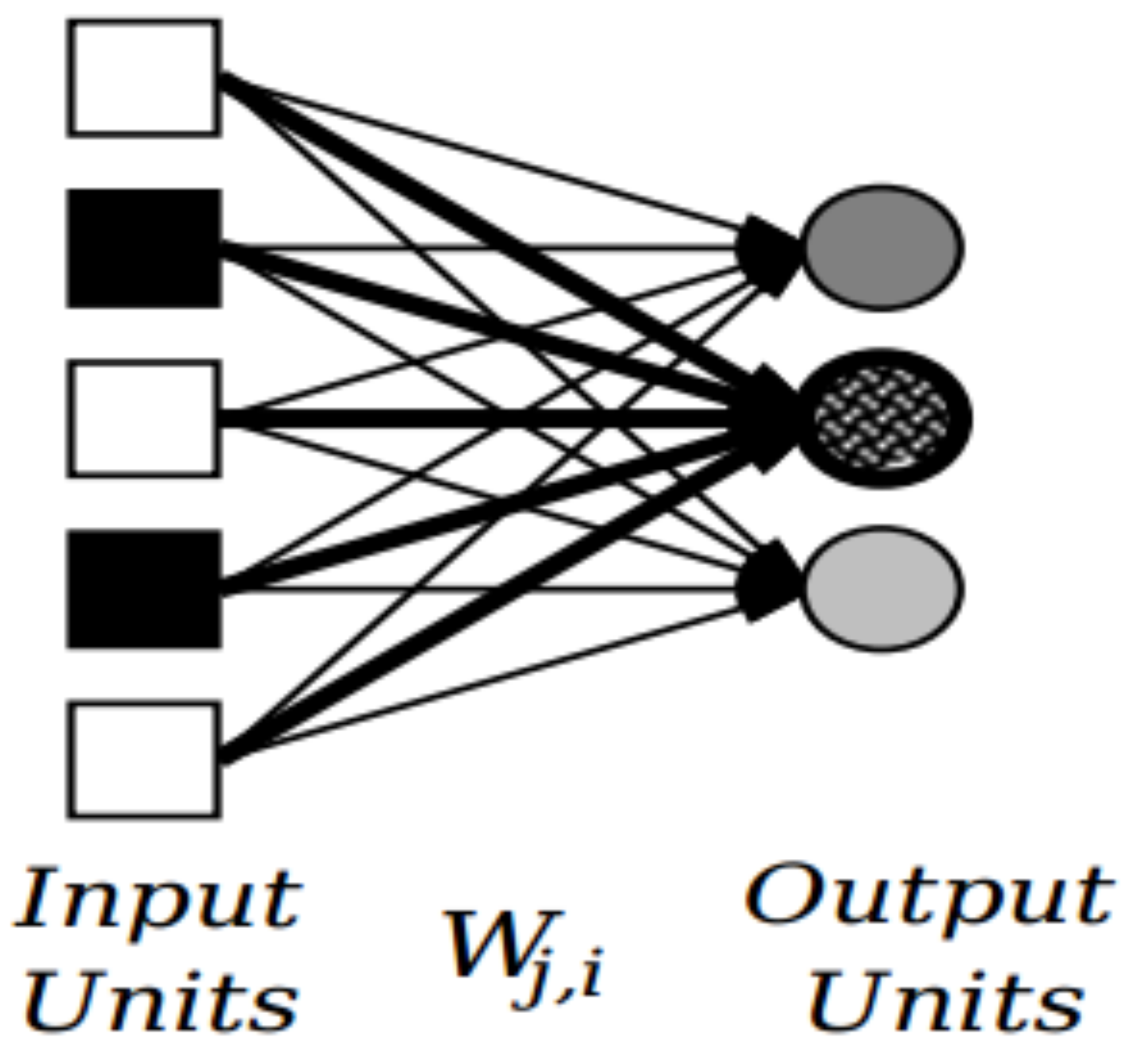


$$a_5 = g(W_{3,5} \cdot a_5 + W_{4,5} \cdot a_4)$$

$$a_5 = g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))$$

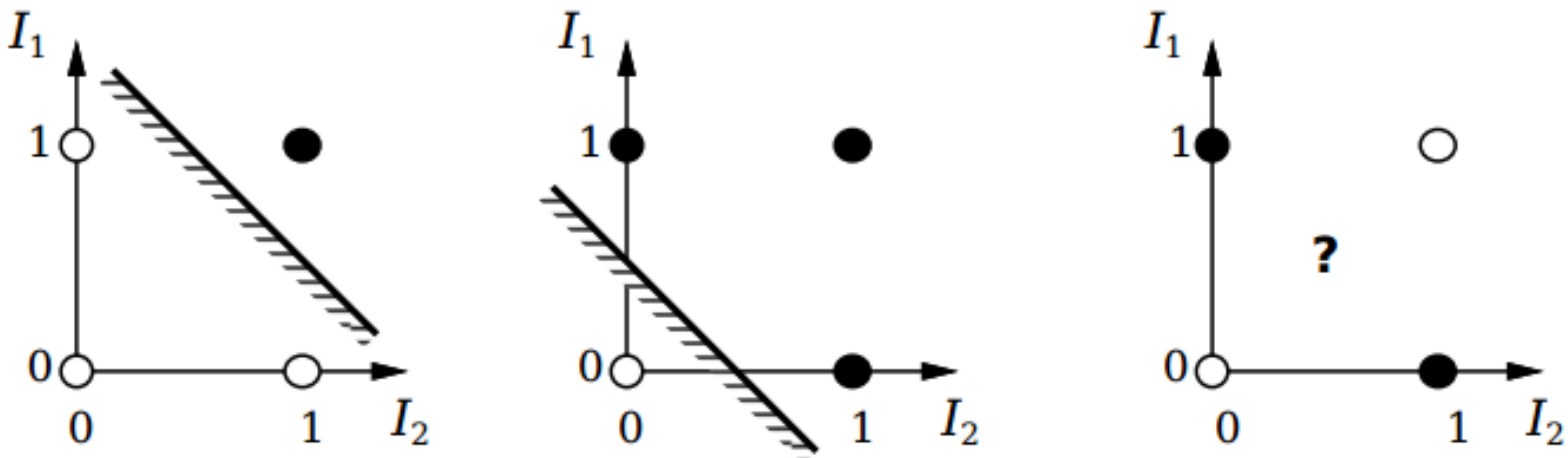
Perceptrons

Single layer feed-forward network



Perceptrons

Can learn only linear separators



(a) I_1 **and** I_2 (b) I_1 **or** I_2 (c) I_1 **xor** I_2

Training Perceptrons

Learning means adjusting the weights

- Goal: minimize loss of fidelity in our approximation of a function

How do we measure loss of fidelity?

- Often: Half the sum of squared errors of each data point

$$Err = \sum_i 0.5(y_i - h_W(x_i))^2$$

Learning Algorithm

- Repeat for "some time"
- For each example i:

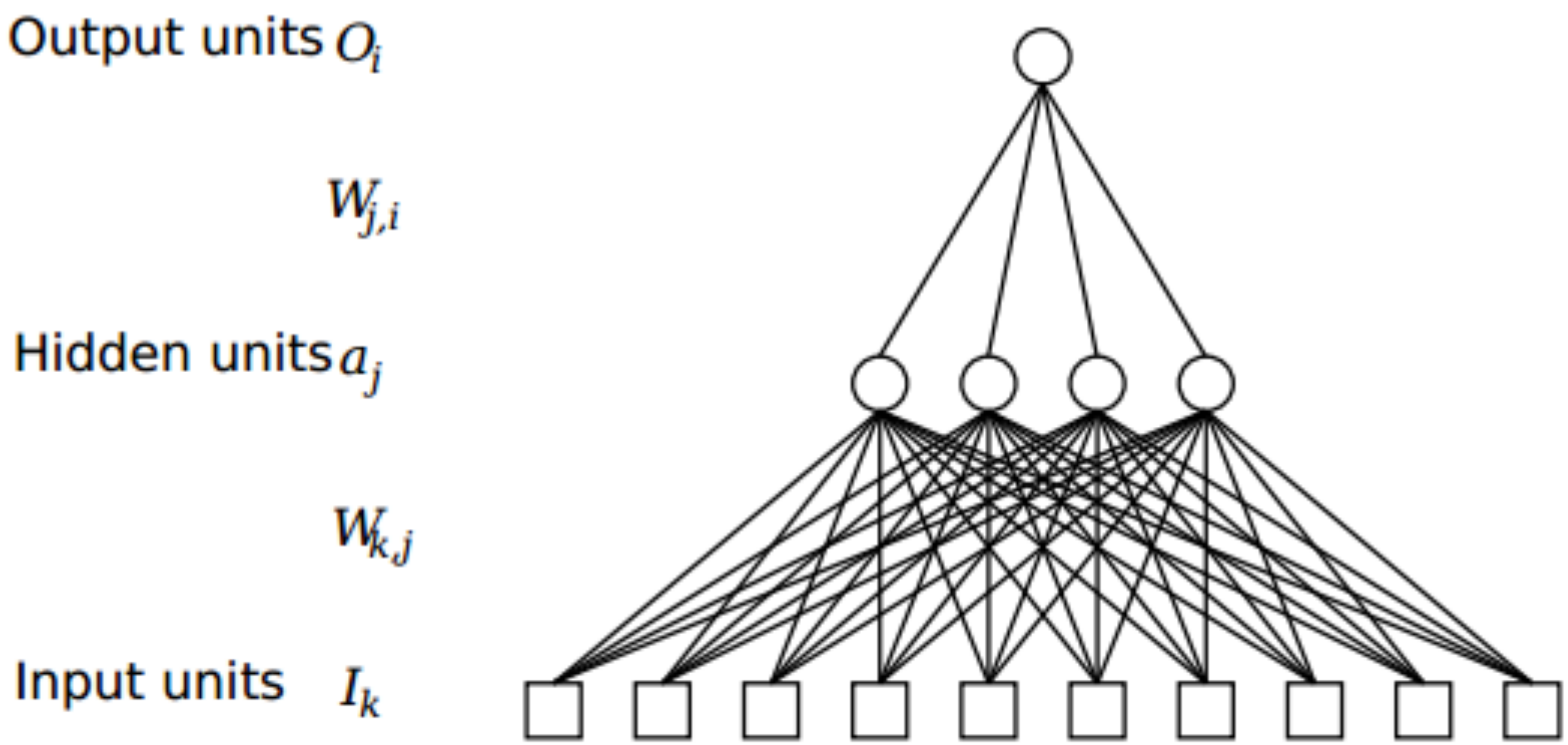
$$I \leftarrow \mathbf{w} \cdot \mathbf{x_i}$$
$$E \leftarrow y_i - g(I)$$
$$W_j \leftarrow W_j + \alpha(E \cdot g'(I) \cdot x_{i,j}) \forall j$$

Multilayer Networks

- Minsky's 1969 book *Perceptrons* showed perceptrons could not learn XOR.
- At the time, no one knew how to train deeper networks.
- Most ANN research abandoned.

Multilayer Networks

- *Any* continuous function can be learned by an ANN with just one hidden layer (if the layer is large enough).



Training Multilayer Nets

- For weights from hidden to output layer, just use Gradient Descent, as before.

$$\Delta_i = E \cdot g'(I)$$
$$W_{j,i} = W_{j,i} + \alpha \Delta_i a_j$$

- For weights from input to hidden layer, we have a problem: What is y?

$$Err = \sum_i 0.5(y_i - h_W(x_i))^2$$

Back Propagation

- Idea: Each hidden layer caused *some* of the error in the output layer.
- Amount of error caused should be proportionate to the connection strength.

$$\Delta_i = E \cdot g'(I)$$
$$W_{j,i} = W_{j,i} + \alpha \Delta_i a_j$$

$$\Delta_j = g'(I) \cdot \sum_i W_{j,i} \Delta_i$$

$$W_{k,j} = W_{k,j} + \alpha \Delta_j x_k$$

Back Propagation

- Repeat for "some time":
- Repeat for each example:
  - Compute Deltas and weight change for output layer, and update the weights .
  - Repeat until all hidden layers updated:
    - Compute Deltas and weight change for the deepest hidden layer not yet updated, and update it.

When to use ANNs

- When we have high dimensional or real-valued inputs, and/or noisy (e.g. sensor data)
- Vector outputs needed
- Form of target function is unknown (no model)
- Not import for humans to be able to *understand* the mapping

Drawbacks of ANNs

- Unclear how to interpret weights, especially in many-layered networks.
- How deep should the network be? How many neurons are needed?
- Tendency to overfit in practice (very poor predictions outside of the range of values it was trained on)