CS6482
Deep Reinforcement Learning

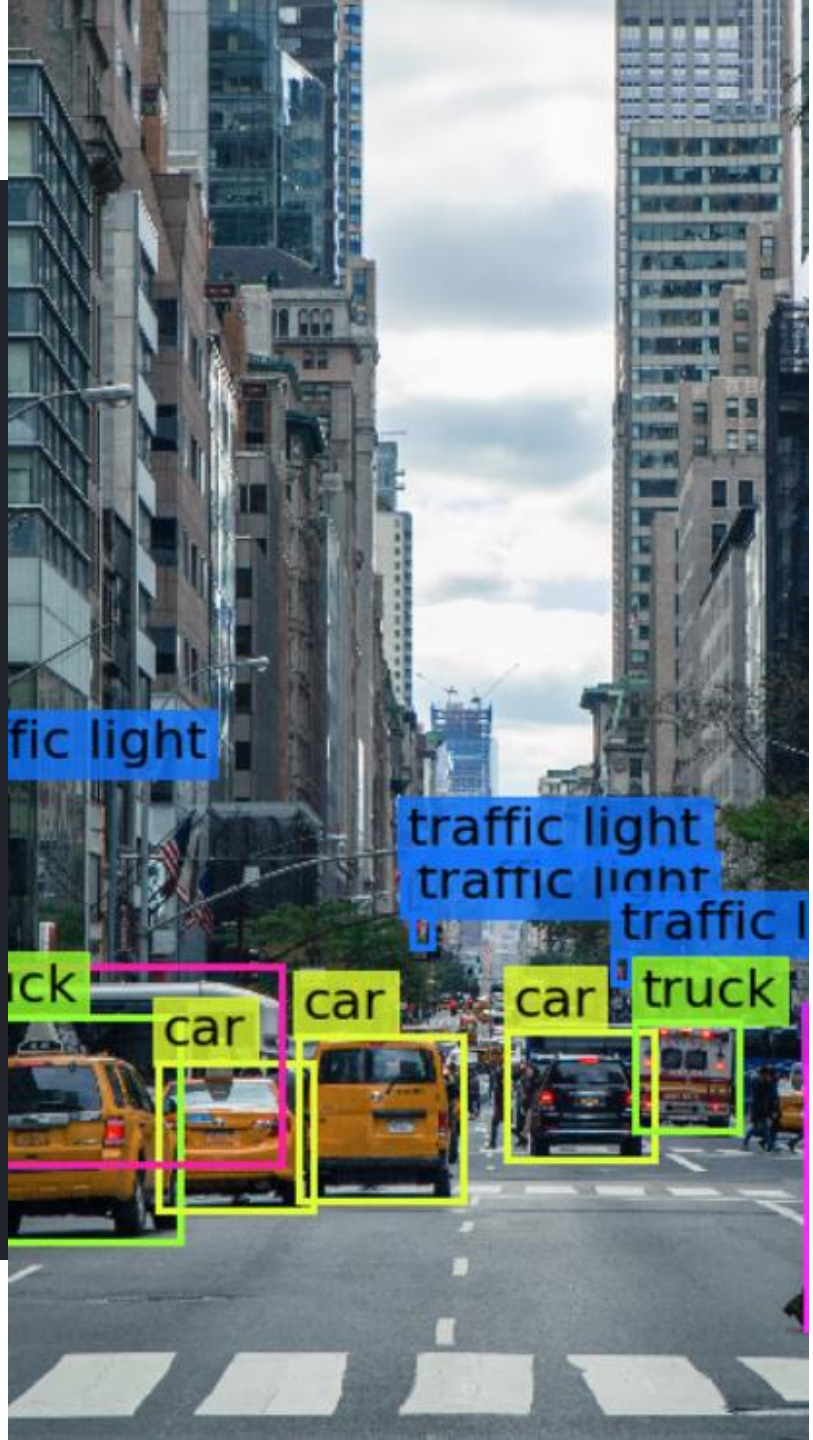
B: Perceptron, Gradient Descent, MLP, Backpropagation (BP), and MLP CODE from Nielsen .

J.J. Collins

Dept. of CSIS

University of Limerick

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH



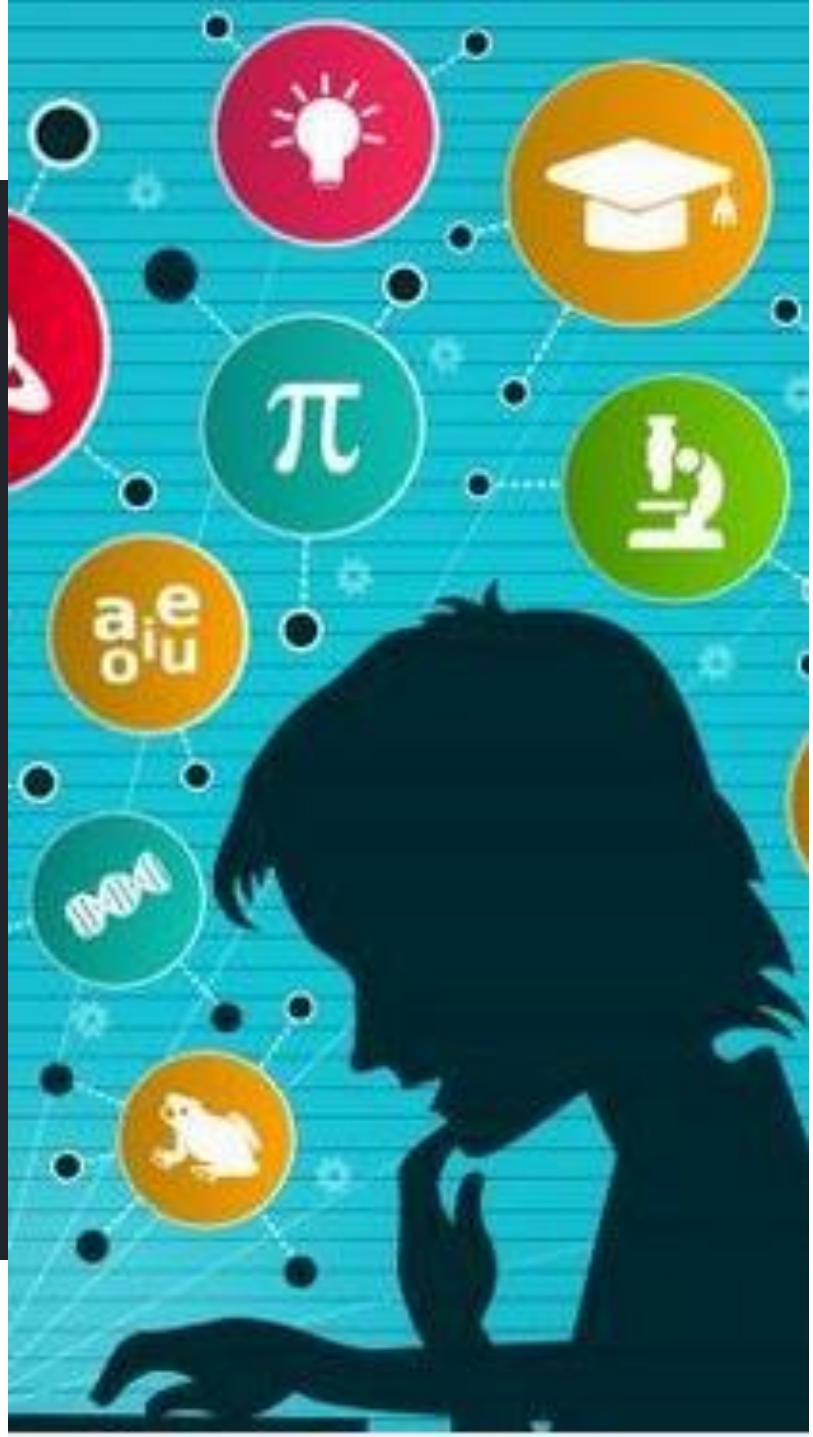
Objectives

- ❑ Part 1: Learning
- ❑ Part 1: The Perceptron and Gradient Descent
- ❑ Part 3: Gradient Descent for MLPs
- ❑ Part 4: Hyperparameters, Cross Fold Validation, Generalisation
- ❑ Part 5: MLP for MNIST

Part 1

LEARNING

UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH



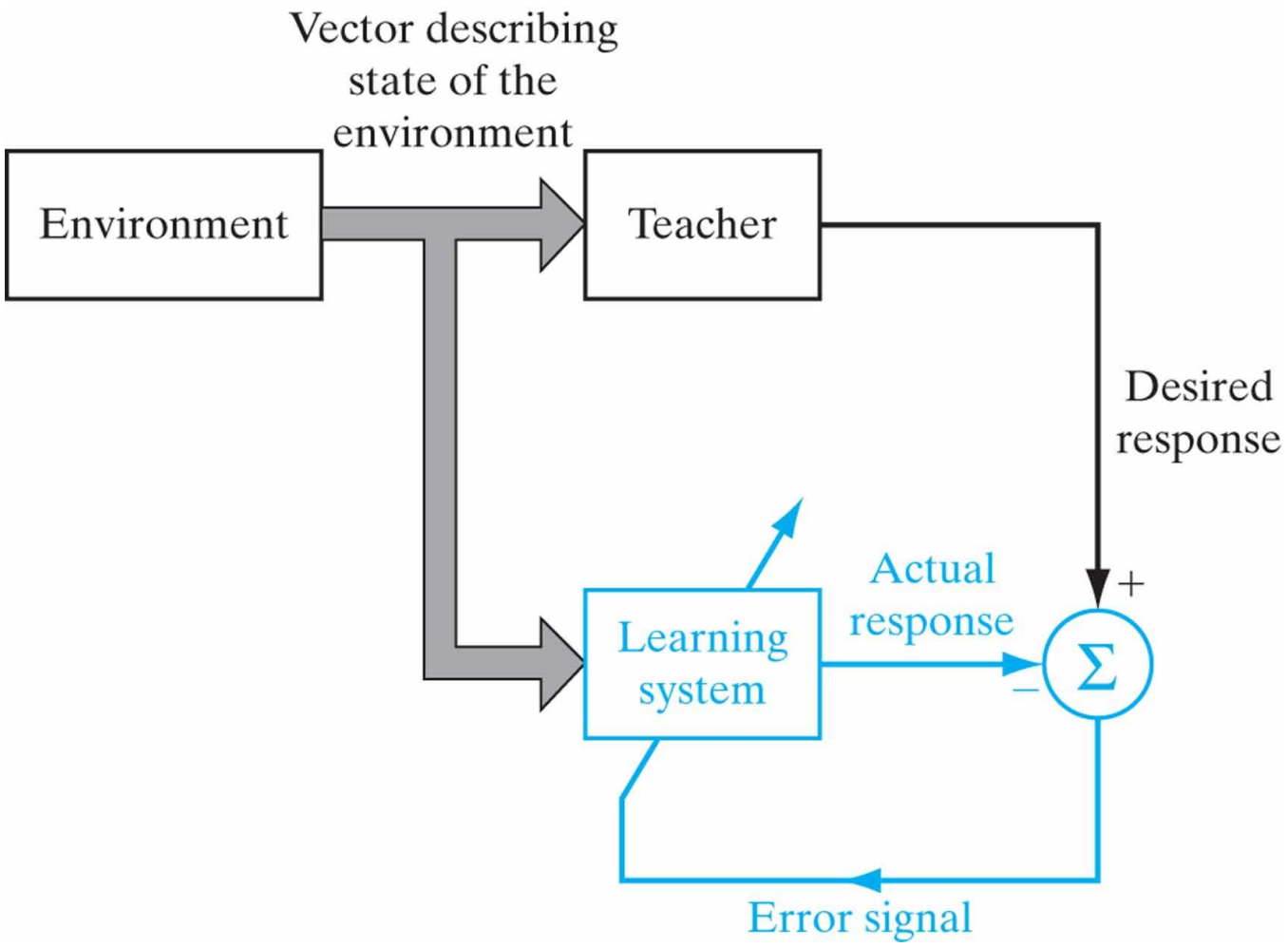
LEARNING

- A computer program is said to learn from experience E with respect to a class of tasks T and performance measure P, if its performance at tasks in T as measured by P improves with E

Tom Mitchell. Machine Learning. 1997.

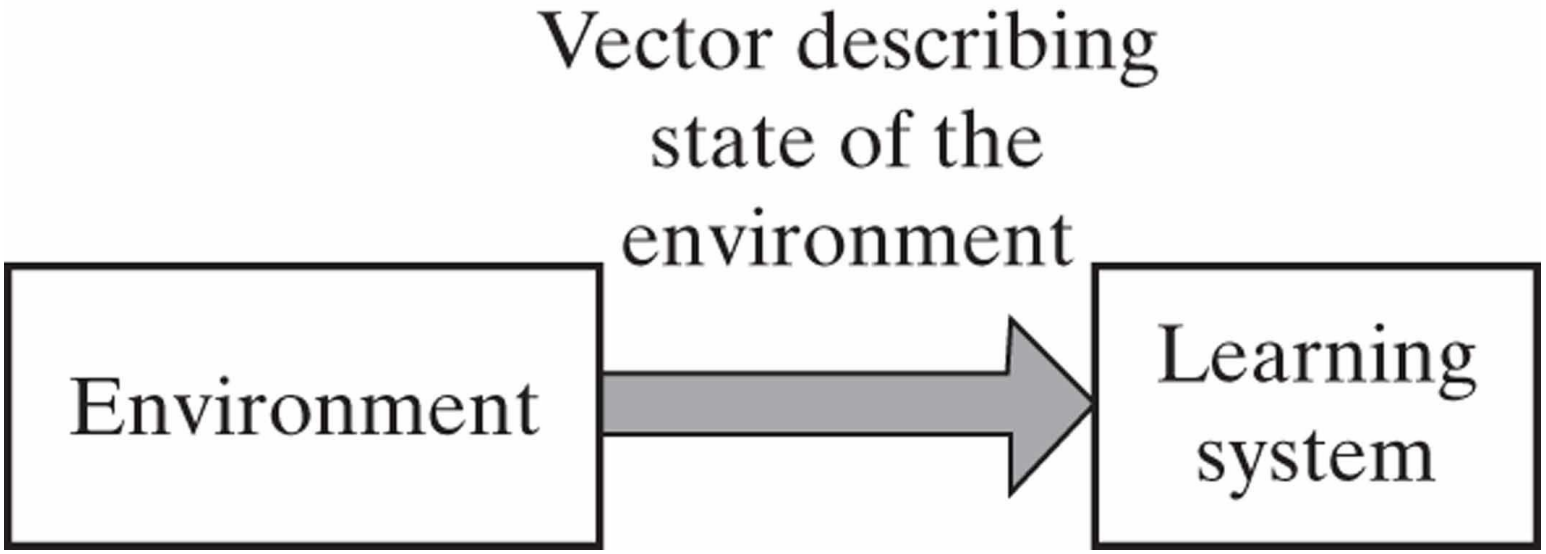
Learning

- Supervised Learning



Learning

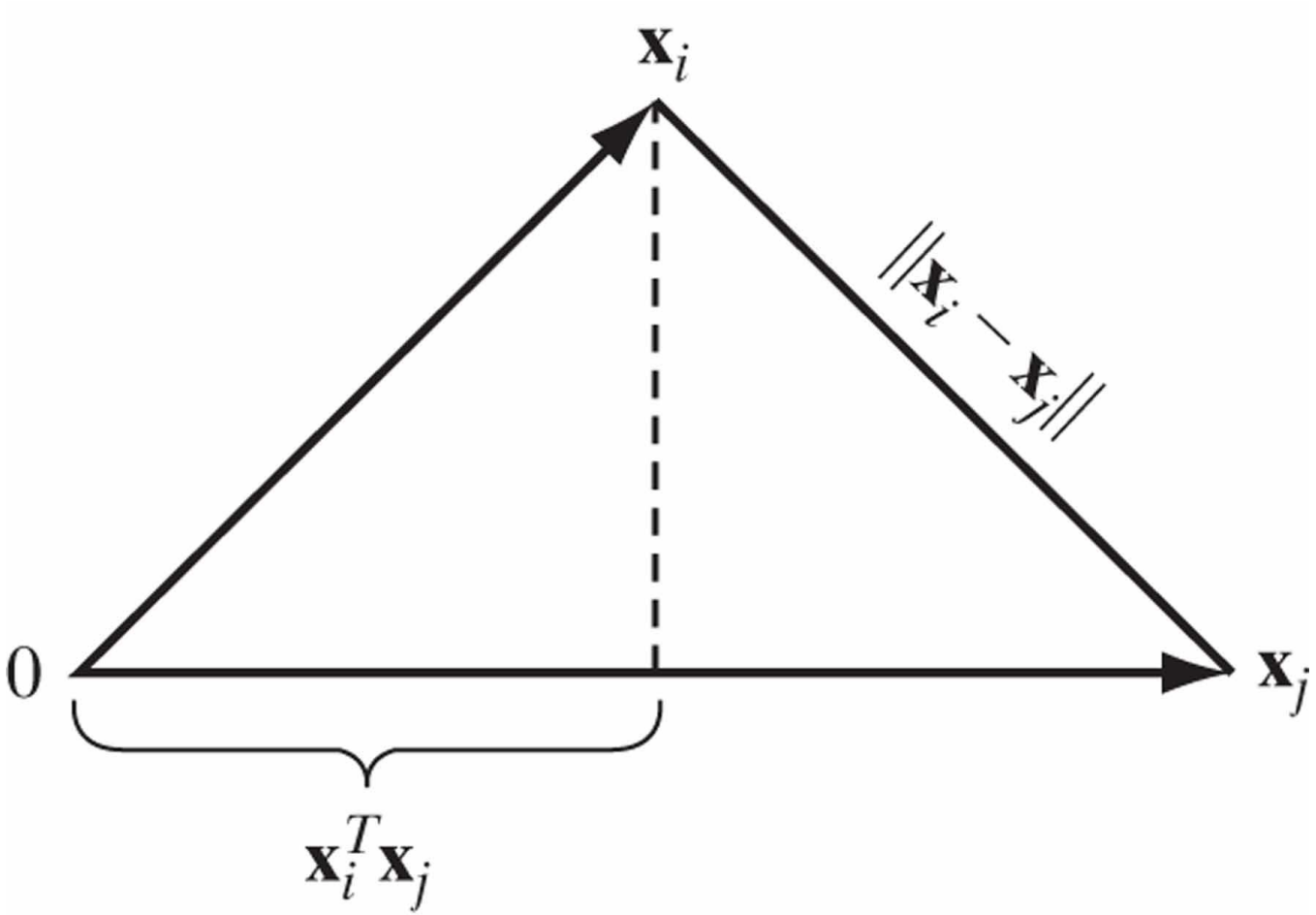
- Unsupervised Learning



- System searches for patterns in the data that are used to decompose it into disjoint sets.
- Example: A Self-Organising Map, K-Means Clustering, etc.

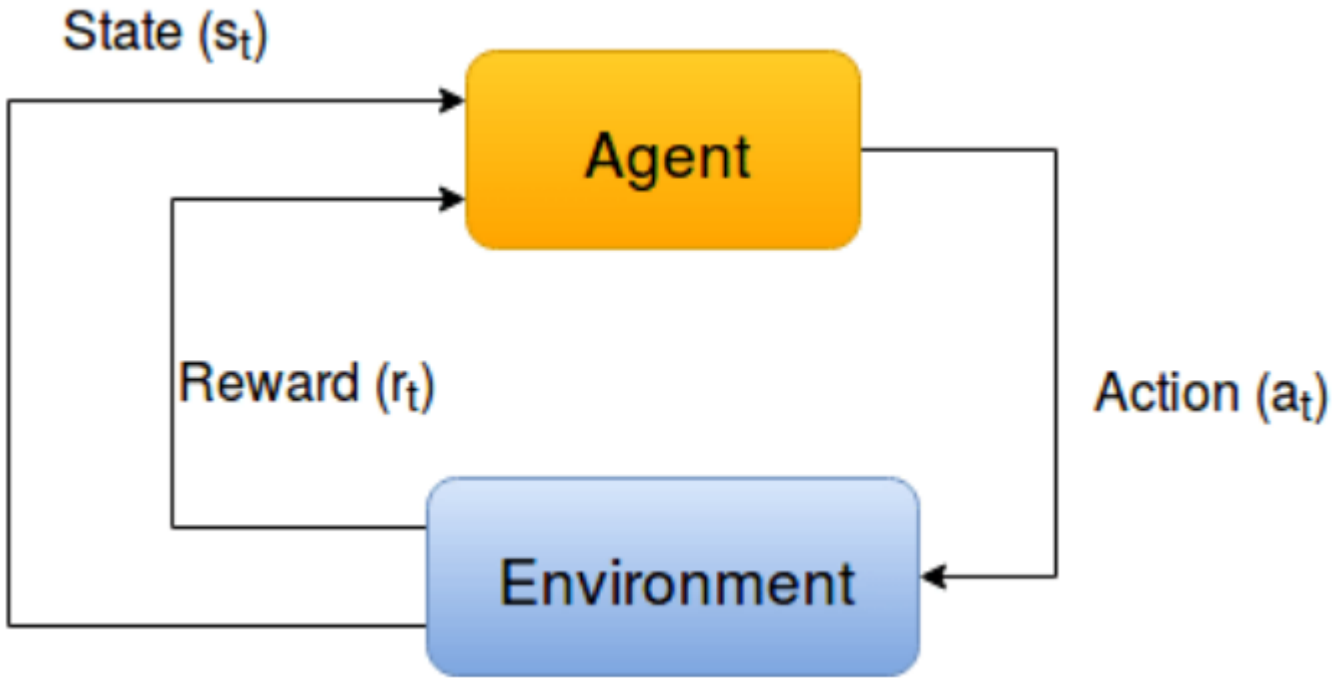
Learning

- Learning – similarity: inner produce v Euclidian distance



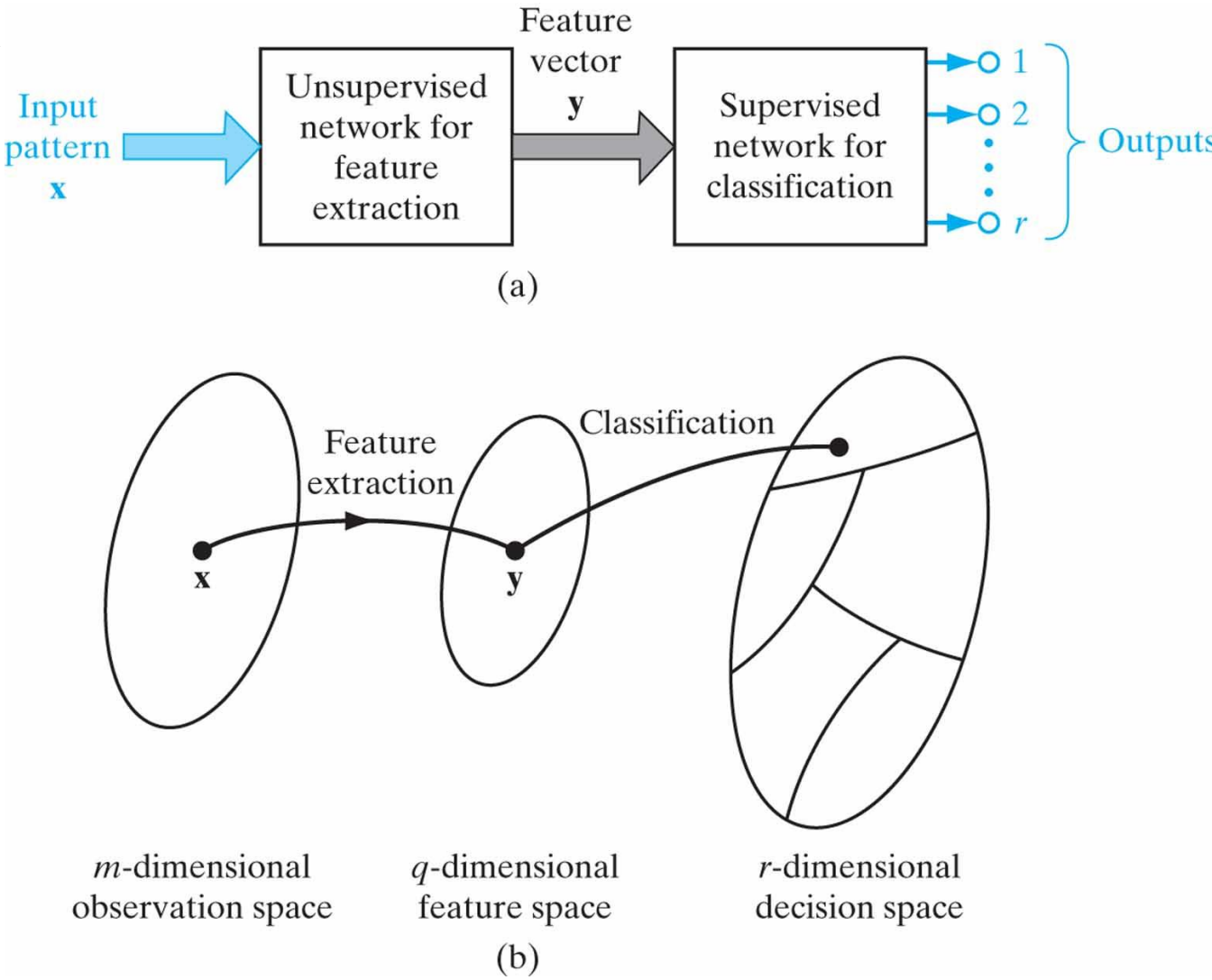
Learning

- Reinforcement Learning



Hybrid

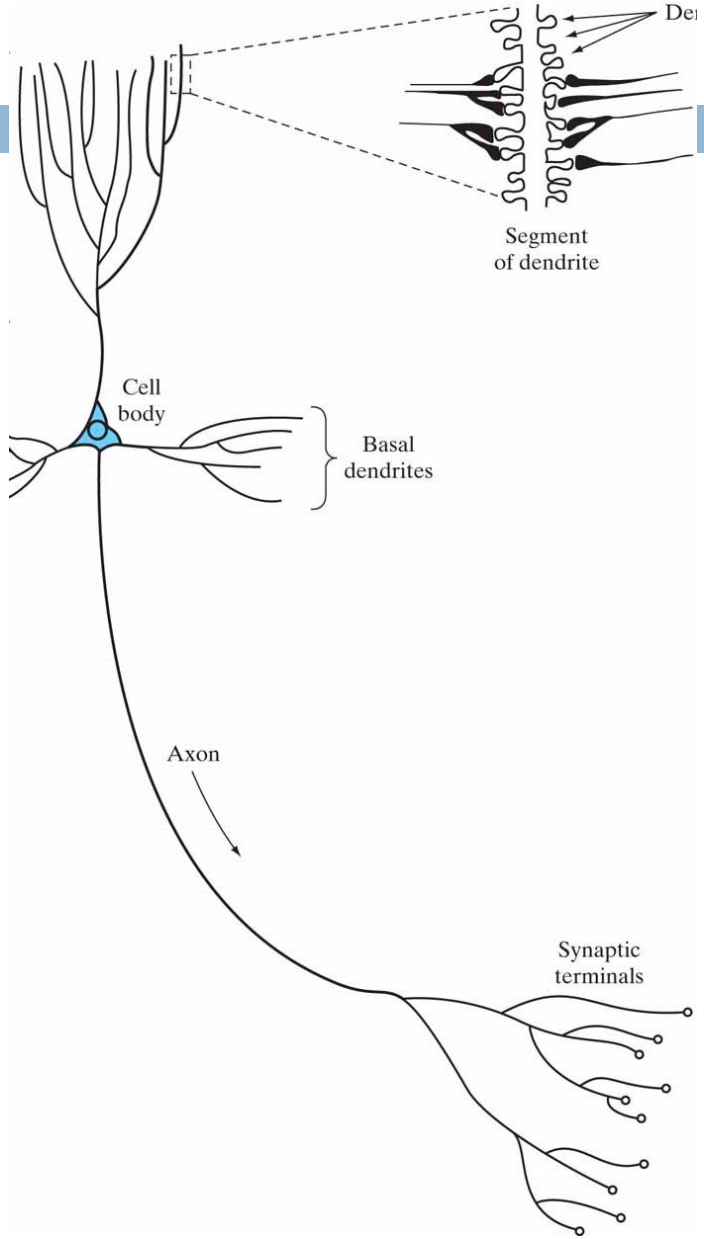
- AI practitioner usually tasked with feature extraction.
- An alternative - hybrid



Part 2

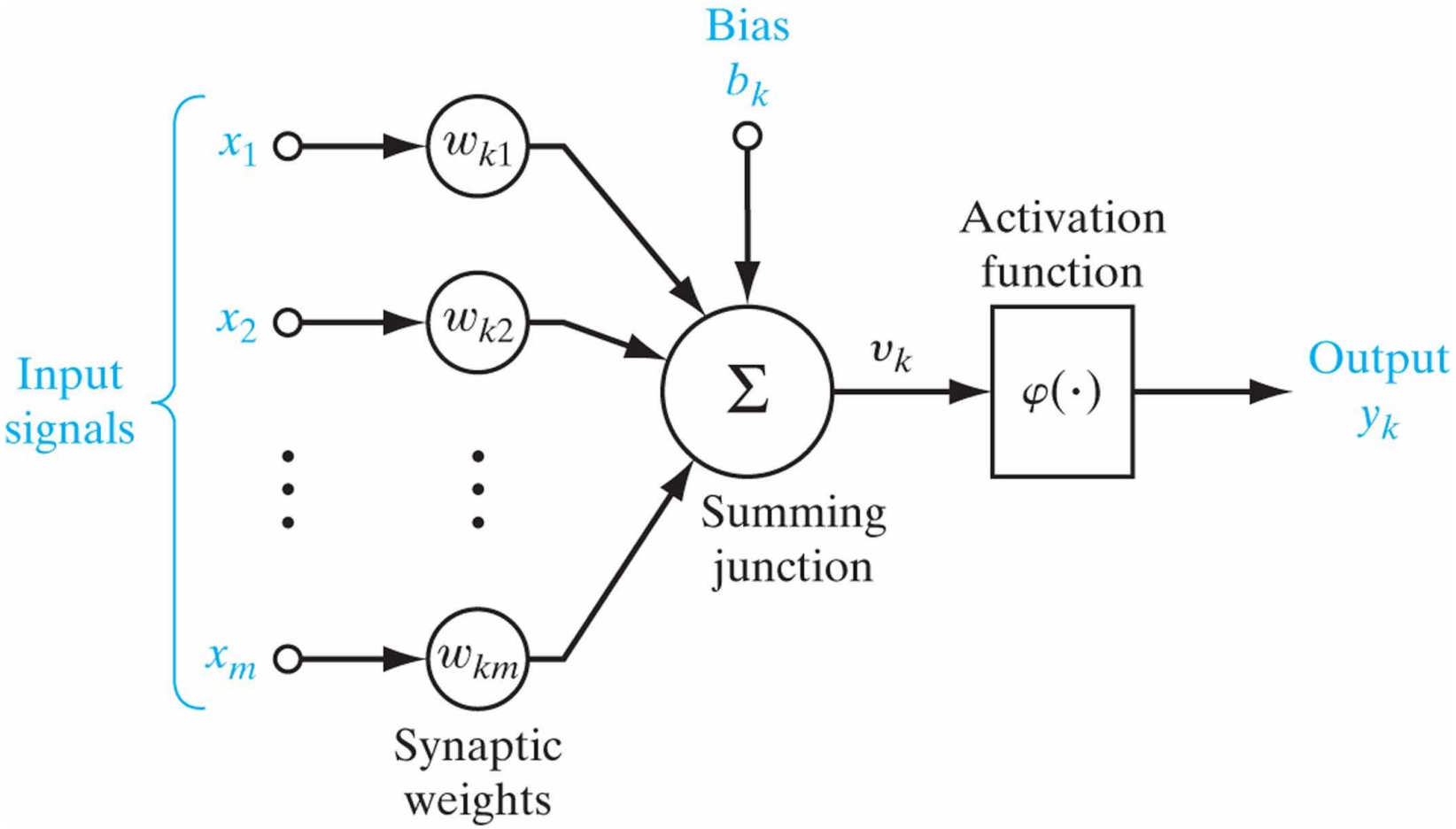
The Perceptron

Neurons



- Dendrites accept signals
- Cell body computes a response
- Response is transmitted to other neurons via synaptic terminals
- An example of a system with emergent properties
- The sum is greater than the parts
- Alan Turning identified this as a very likely ingredient to AI

A Computational Neuron

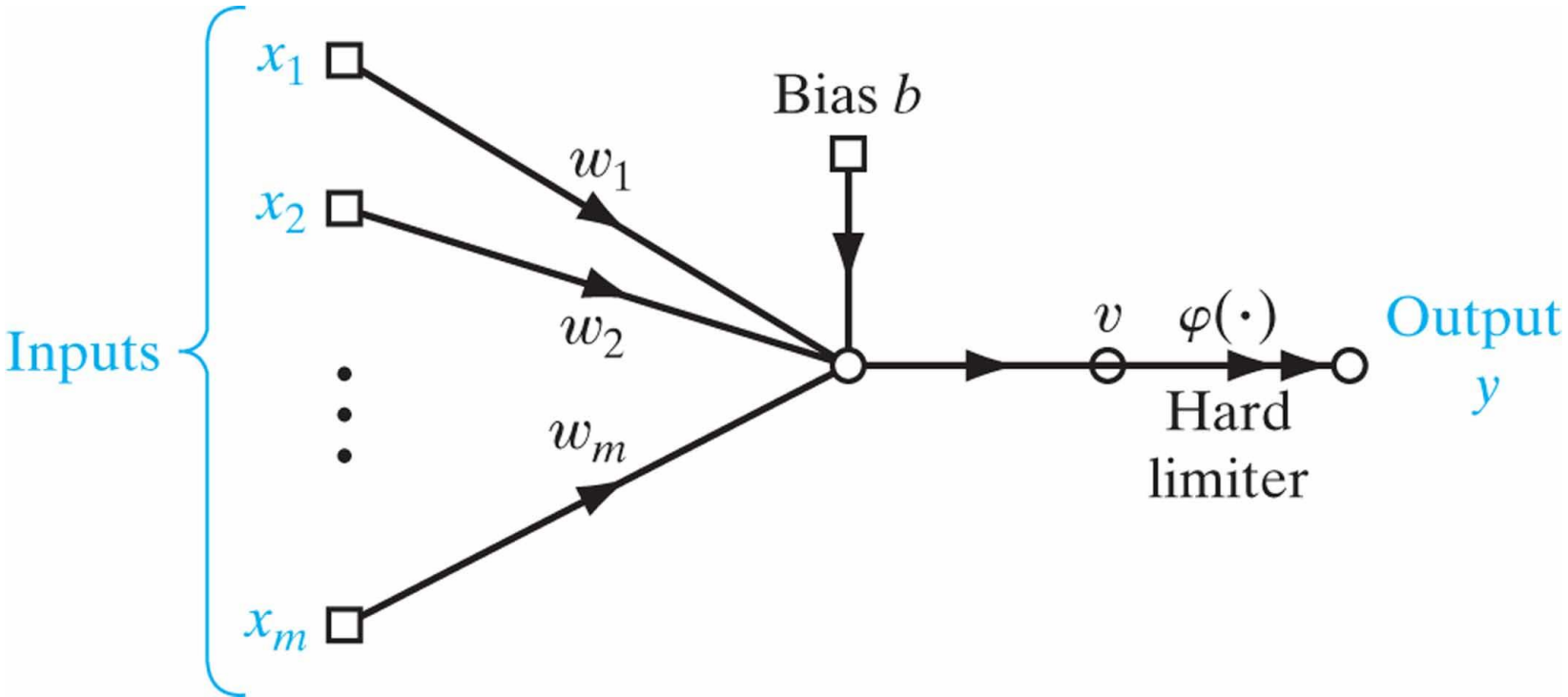


Rosenblatt's Perceptron

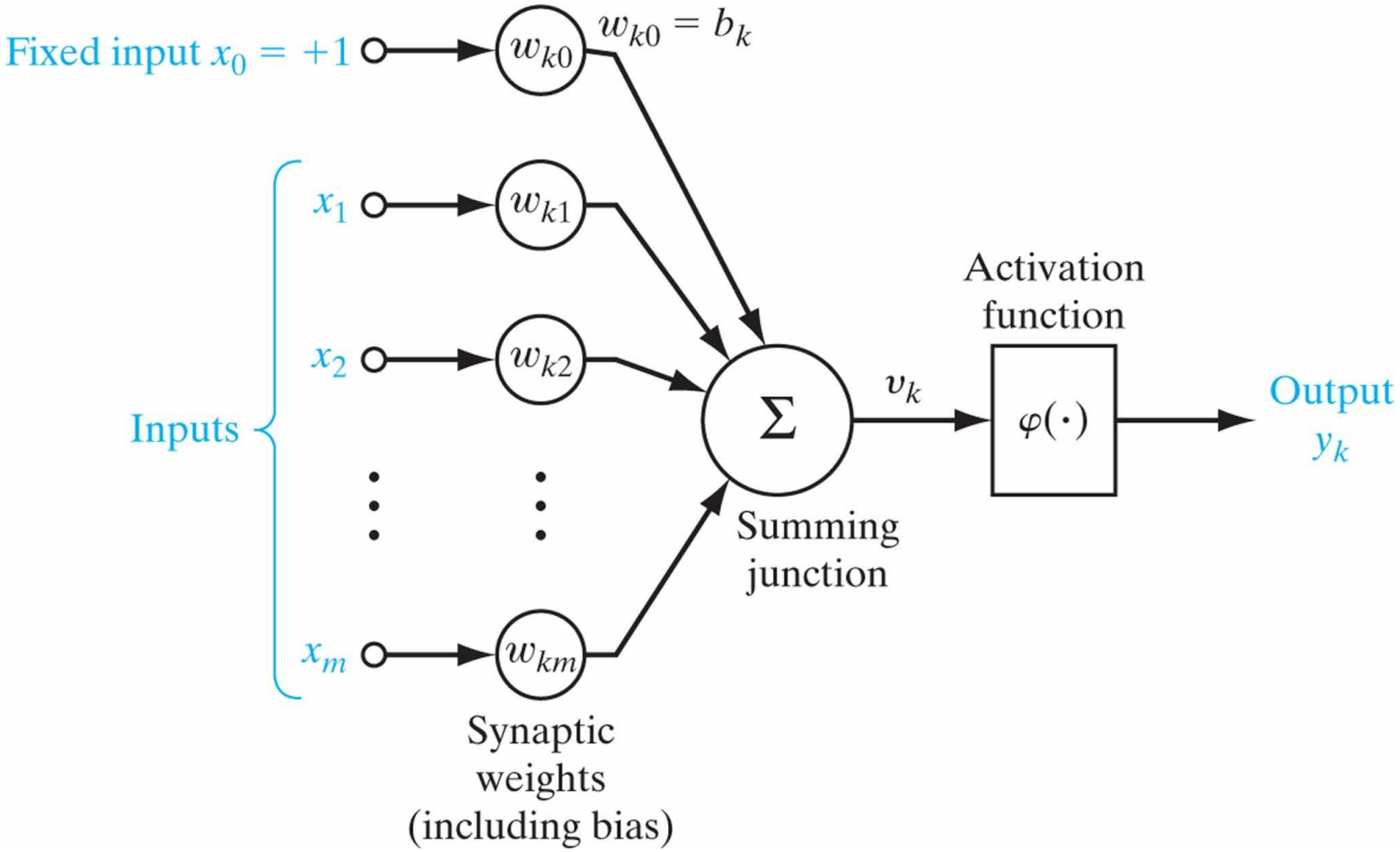
- Based on McCulloch and Pitts model of a neuron (1943)

  McCulloch, W.S., Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* **5**, 115–133 (1943)

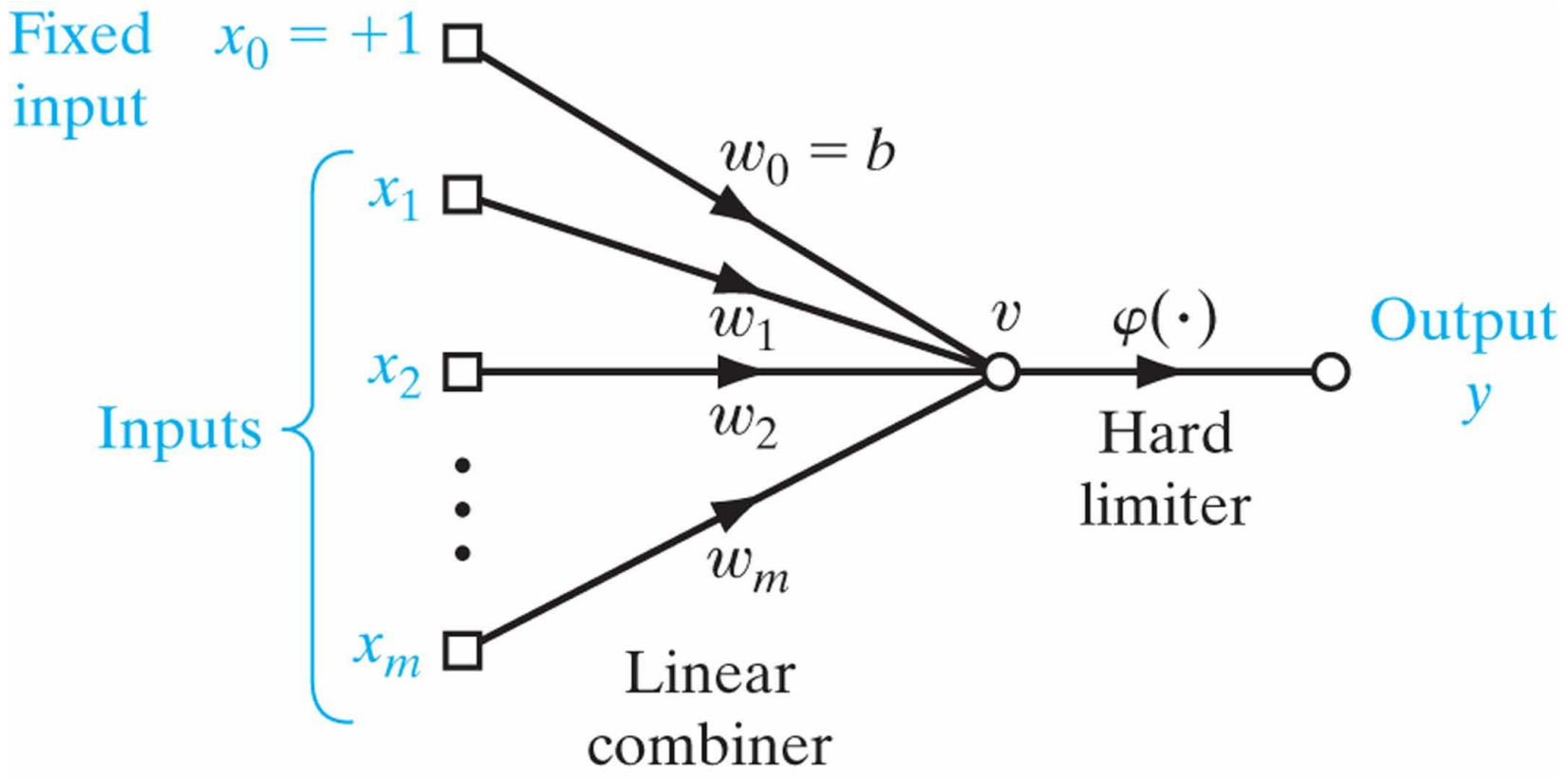
- And Hebbian theory of synaptic plasticity



A Computational Neuron

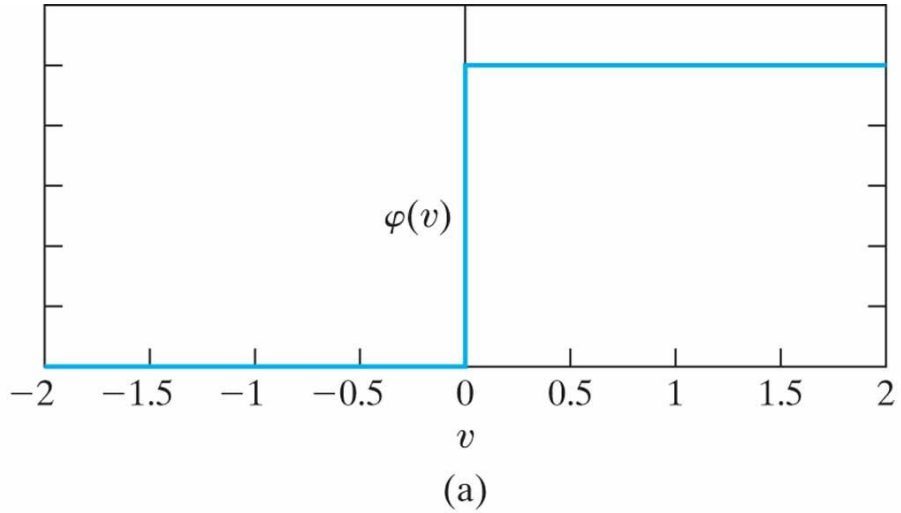


Rosenblatt's Perceptron



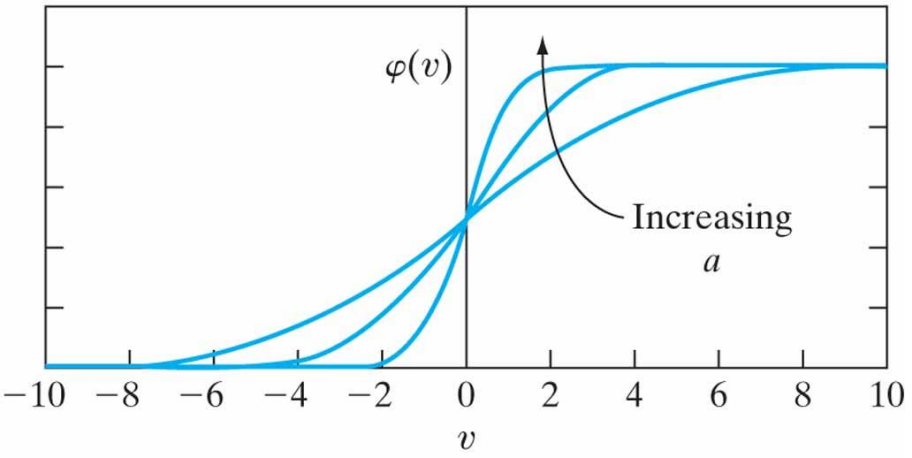
Activation Functions

- Threshold



(a)

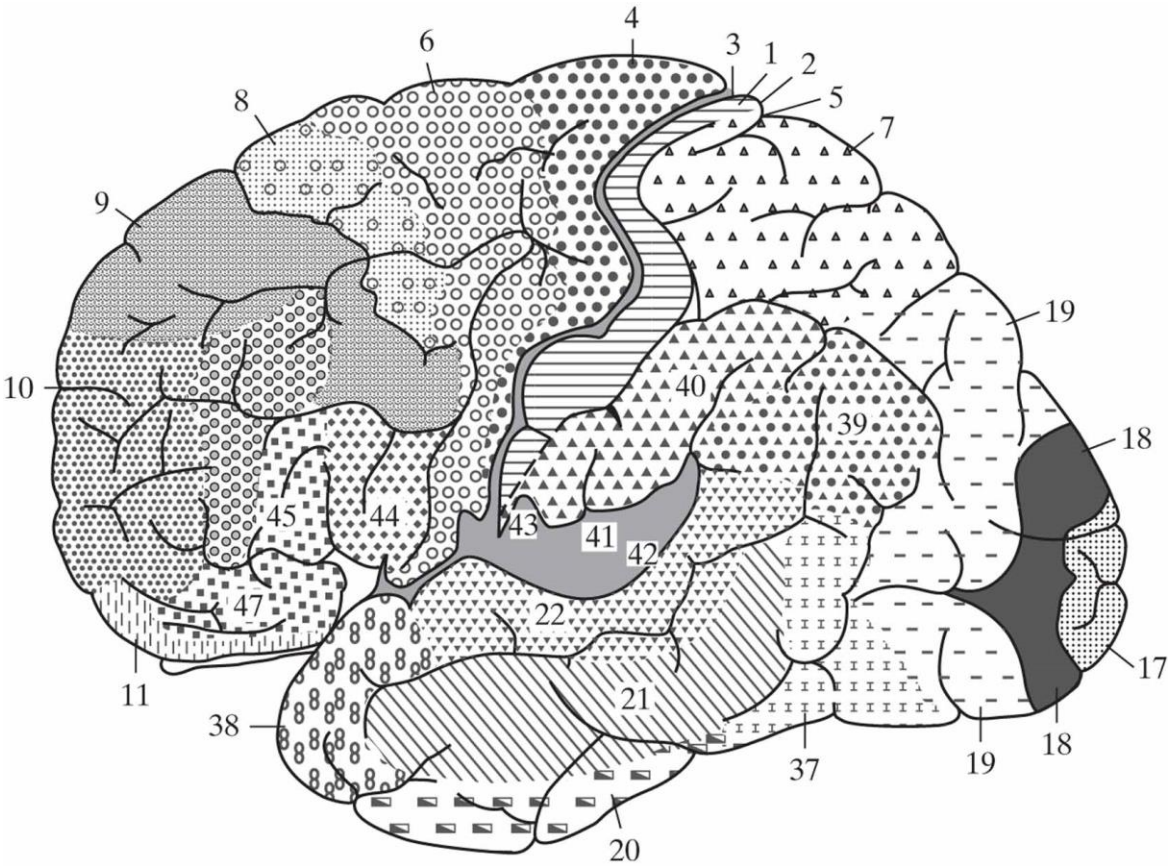
- Sigmoid with varying slope

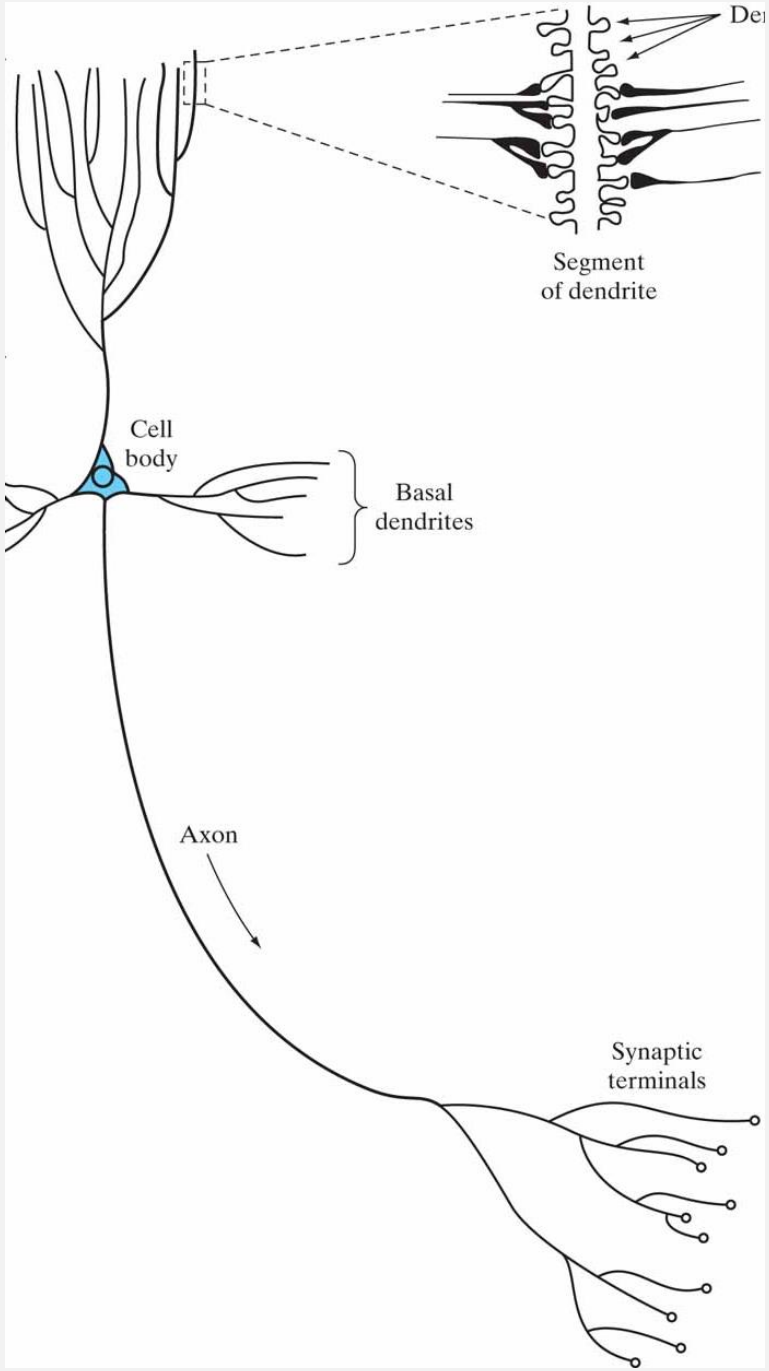


(b)

Mammalian Brain

Map of the cerebral cortex. Some of the key sensory areas show are: Motor cortex: motor strip, area 4; premotor area, area 6; frontal eye fields, area 8. Somatosensory cortex: areas 3, 1, and 2. Visual cortex: areas 17, 18, and 19. Auditory cortex: areas 41 and 42. *(From A. Brodal, 1981; with permission of Oxford University Press.)*





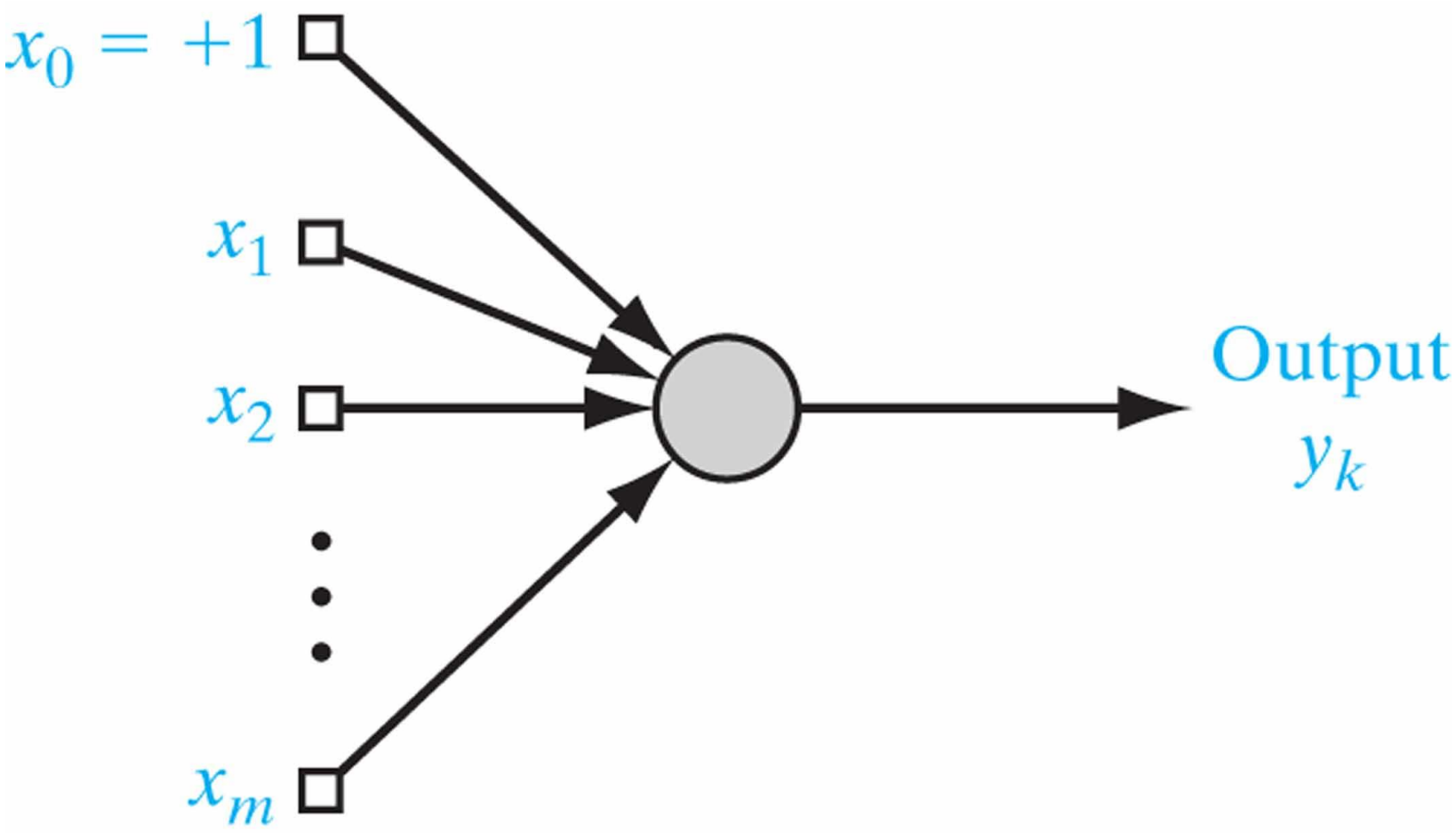
Mammalian Brain

- 86 billions neurons
- 100 trillion connections
- Neuron switching time ~ 0.001 second
- Scene recognition time ~ 0.1 second
- 100 inference steps doesn't seem like enough
- → much parallel computation

Why Neural Computing?

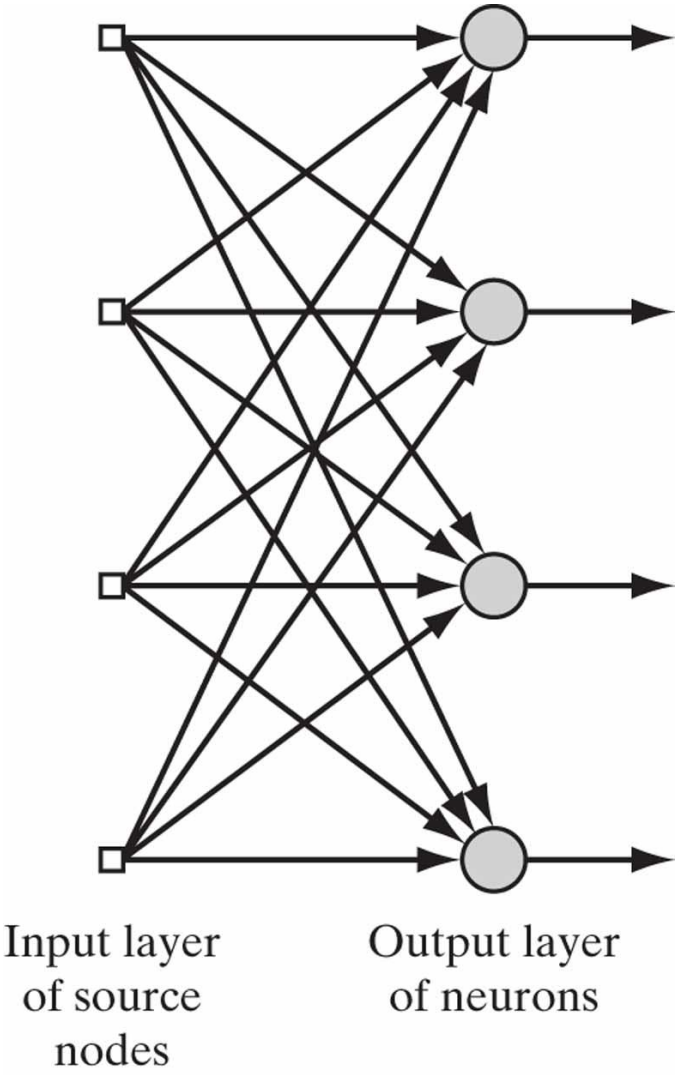
1. Nonlinearity
2. Input-Output Mapping
3. Adaptivity
4. Evidential Response
5. Contextual Information
6. Fault Tolerance
7. VLSI Implementability
8. Uniformity of Analysis and Design
9. Neurobiological Analogy

Drawing a Neuron



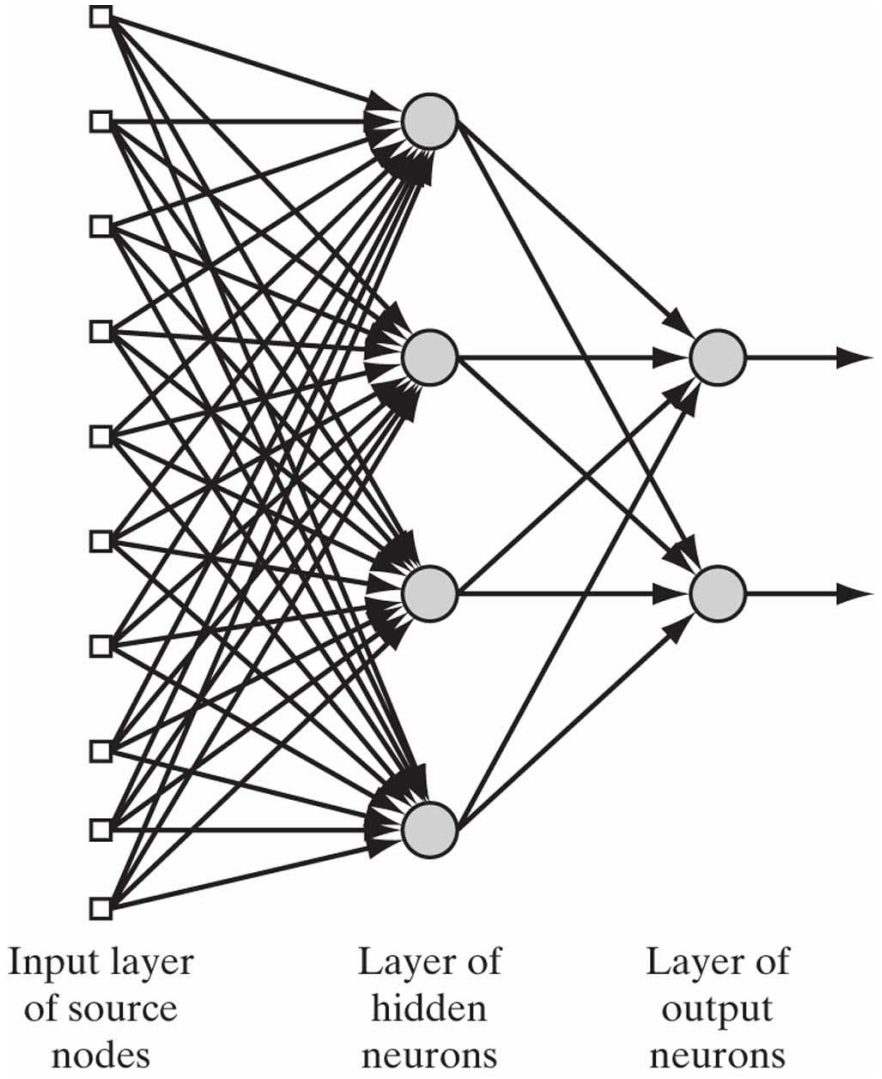
Drawing Neural Architectures

- Feedforward network with one layer of neurons



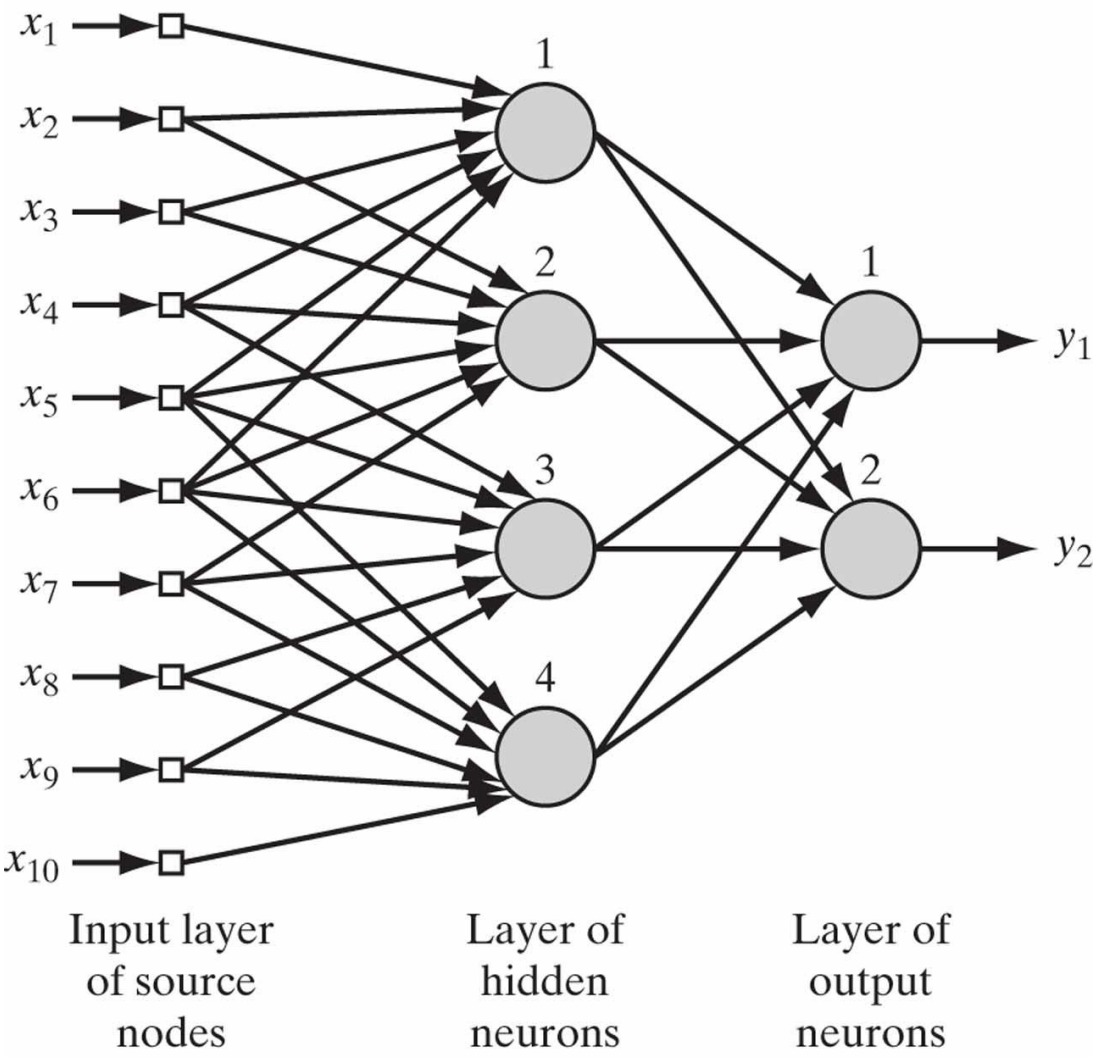
Drawing Neural Architectures

- Fully connected feedforward network with one layer of hidden neurons and one layer of output neurons

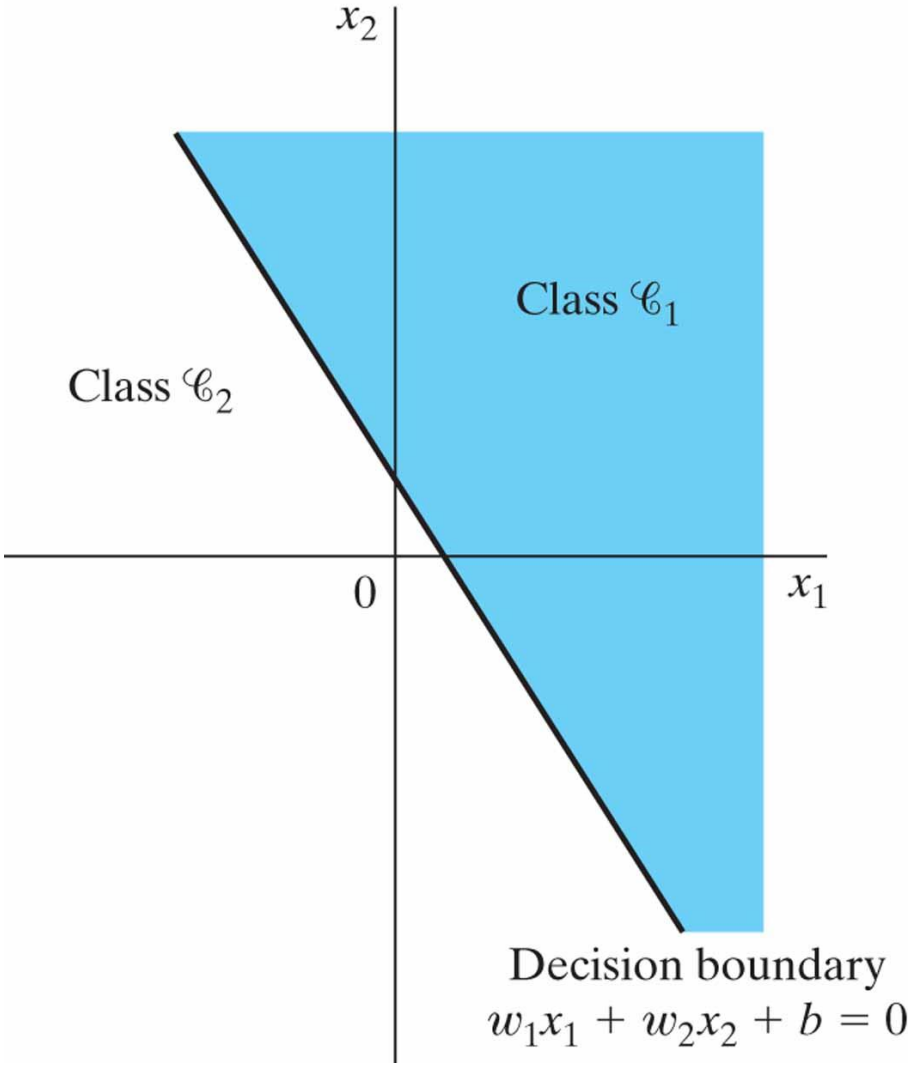


Drawing Neural Architectures

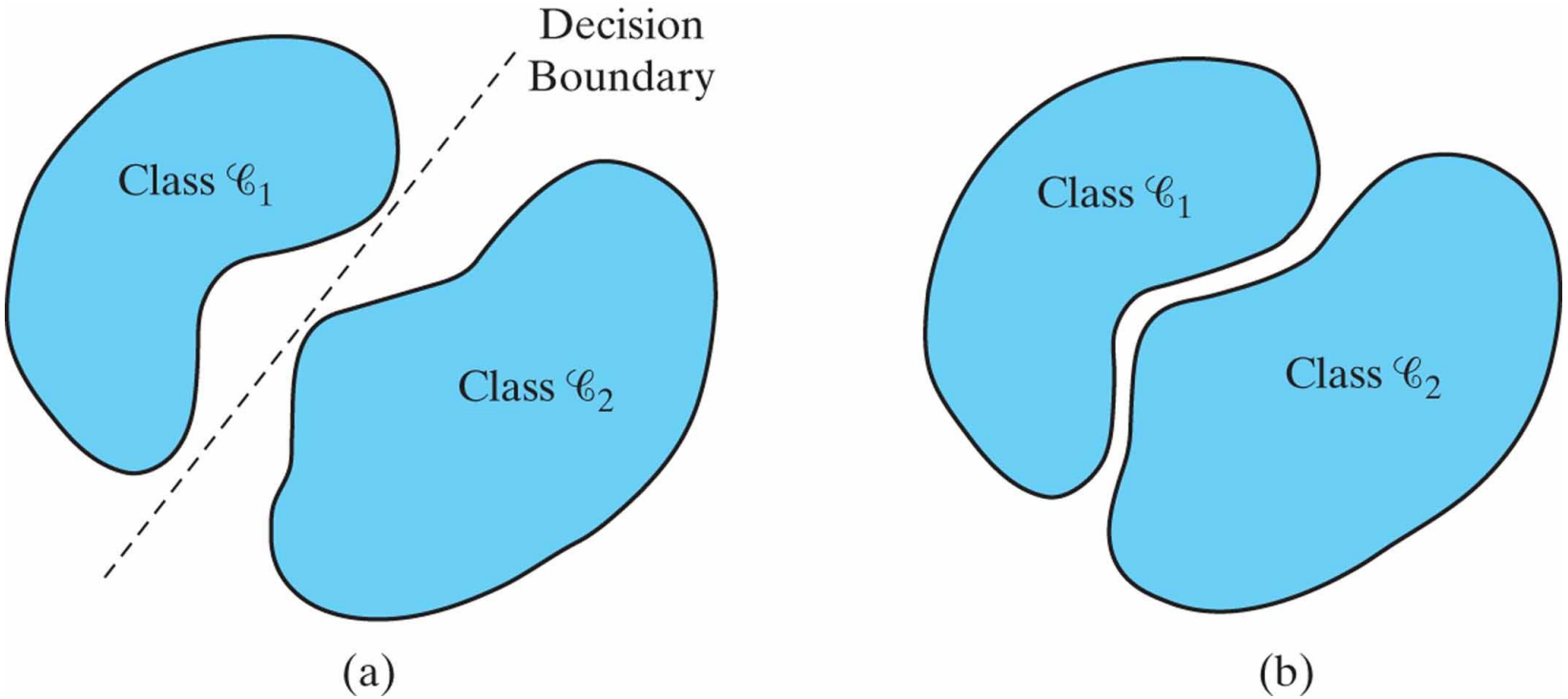
- Feedforward network with receptive fields that has one layer of hidden neurons and one layer of output neurons



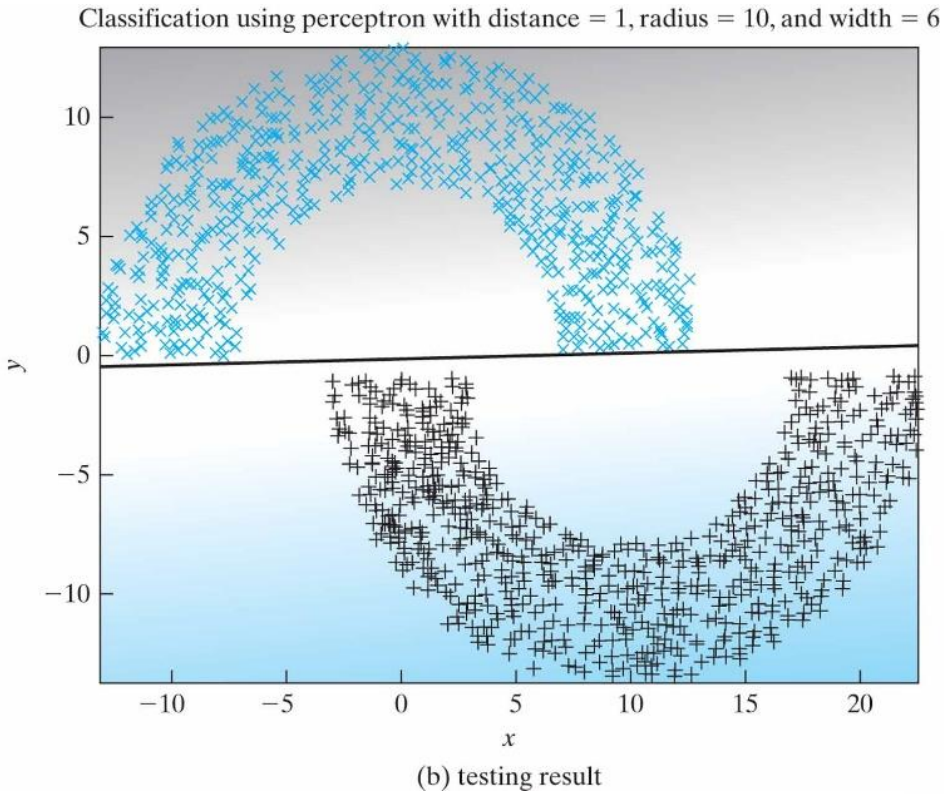
- Hyperplane = decision boundary / decision surface
- Is linear for a perceptron



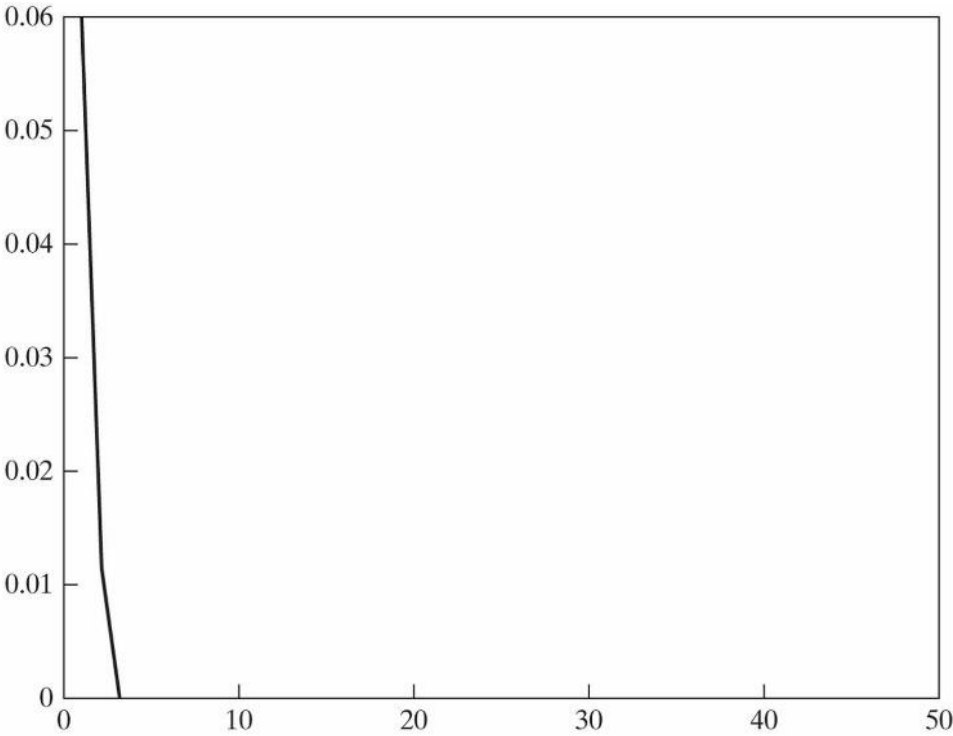
Linear Separable



The Double Moon Classification Problem

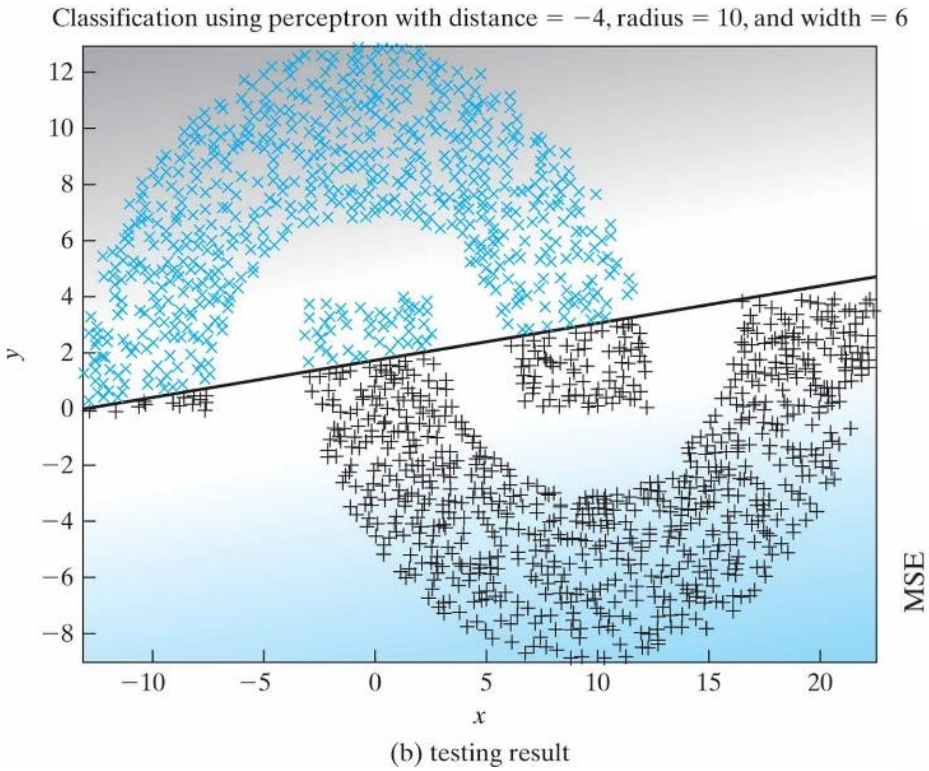


(b) testing result

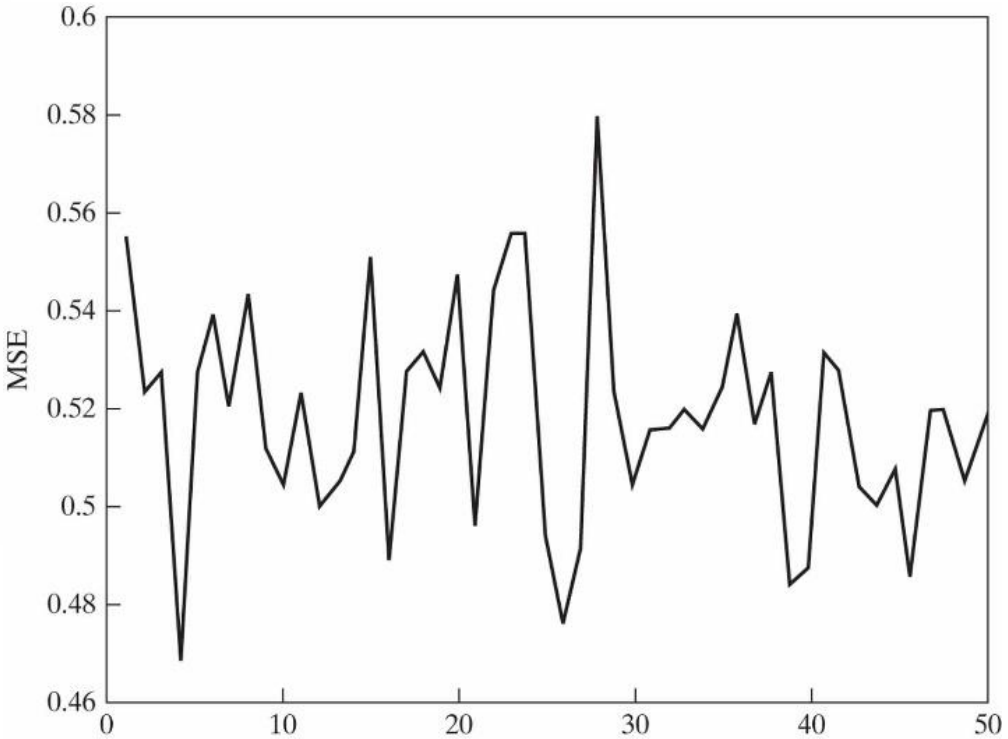


(a) learning curve

The Double Moon Classification Problem



(b) testing result



(a) learning curve

Perceptron Training Algorithm

TABLE 1.1 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

$\mathbf{x}(n)$ = $(m + 1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
$\mathbf{w}(n)$ = $(m + 1)$-by-1 weight vector
$= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
b = bias
$y(n)$ = actual response (quantized)
$d(n)$ = desired response
η = learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time-step $n = 1, 2, \ldots$.
2. *Activation.* At time-step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.
3. *Computation of Actual Response.* Compute the actual response of the perceptron as

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

where sgn(·) is the signum function.

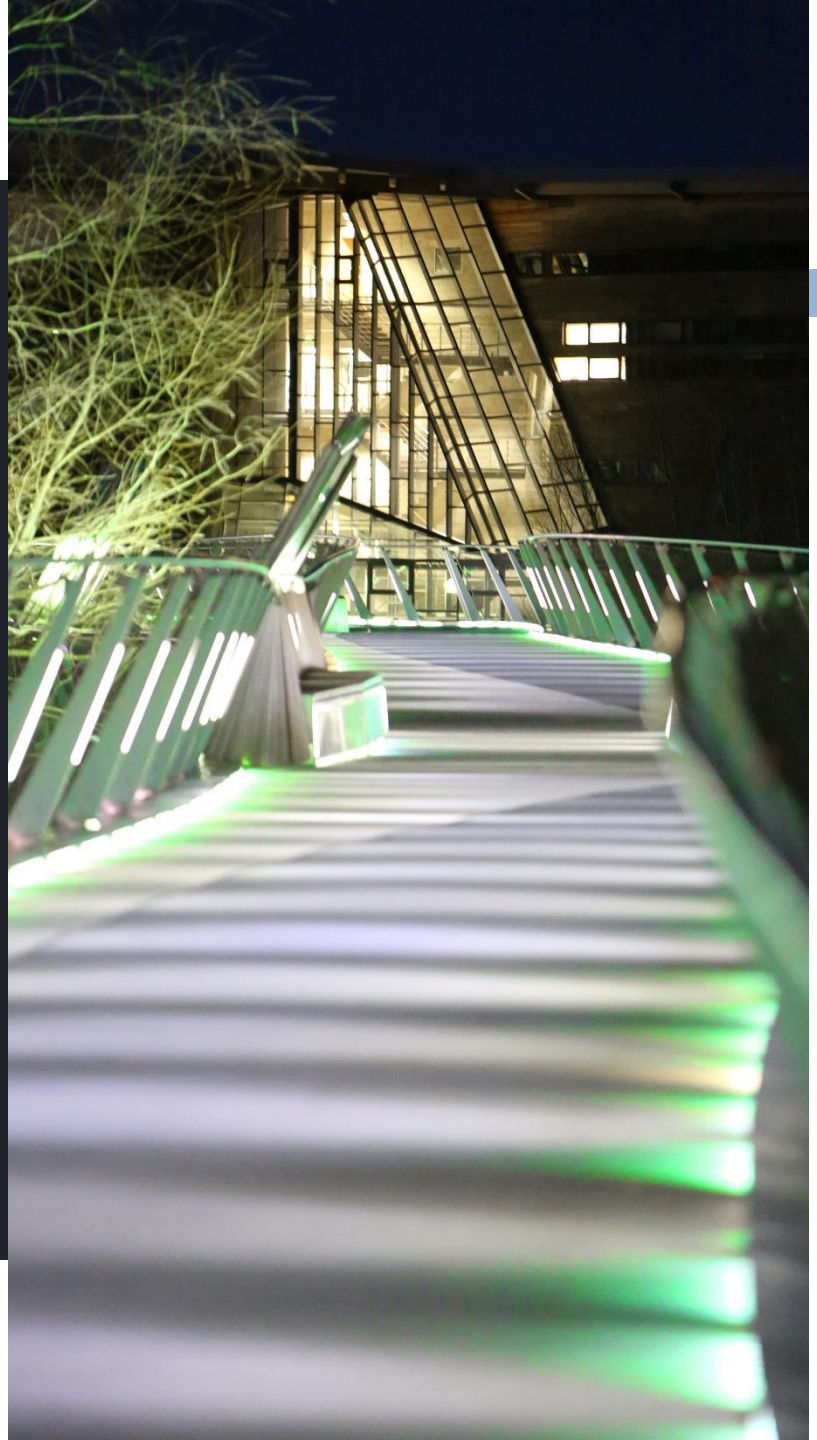
4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron to obtain

$$\mathbf{w}(n + 1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$

where

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$

5. *Continuation.* Increment time step n by one and go back to step 2.



Exercise

Create a data set consisting of 2 linearly separable classes.

Train a single perceptron to classify samples

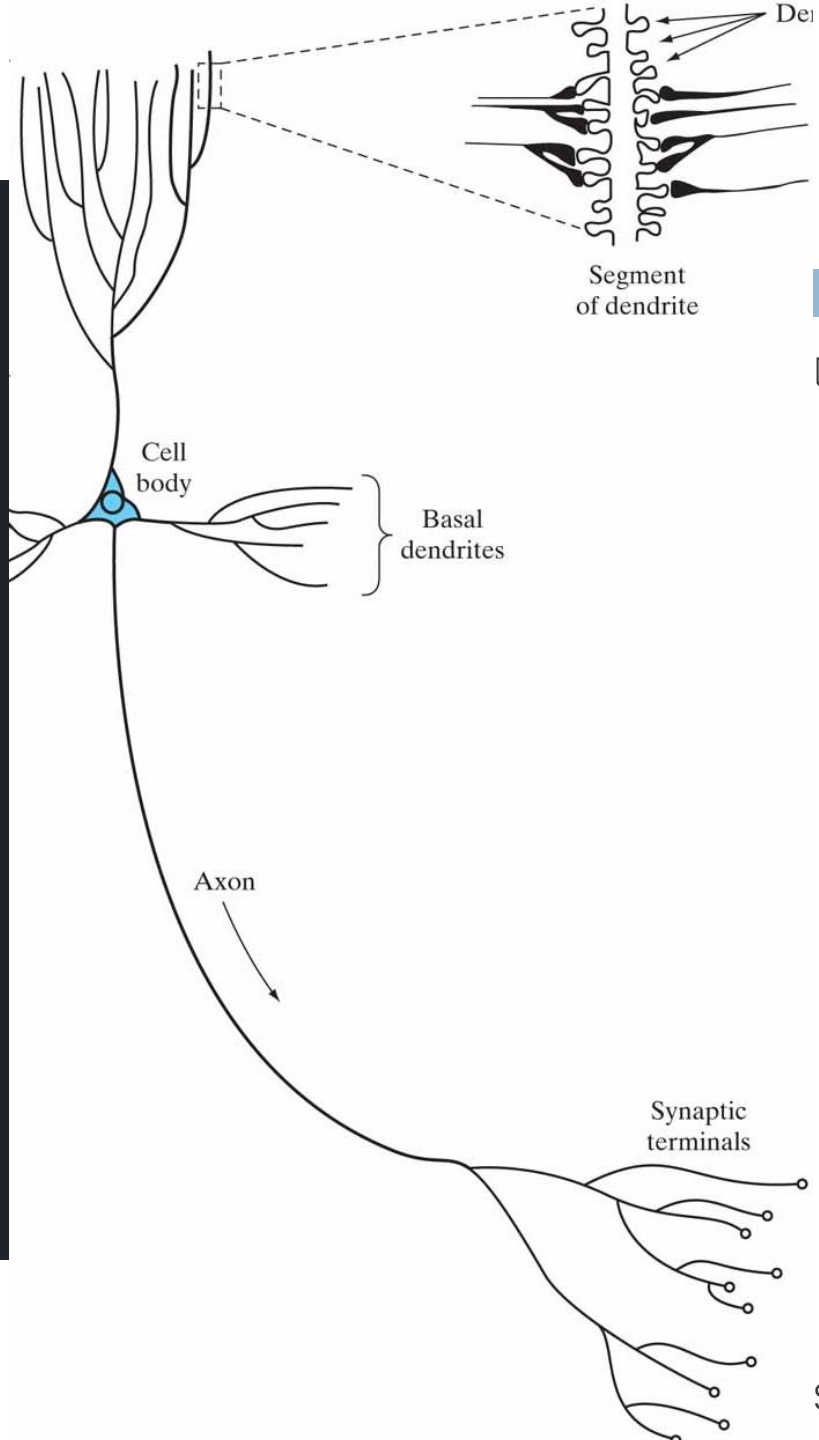
Graph performance

Keep the implementation as simple as possible – KISS principles.

CS6482

ANNs?

- Many neuron-like switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically



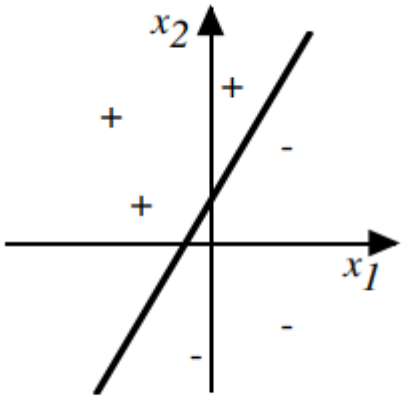
Learning

- A neural Network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two aspects:
  - Knowledge is acquired by the networks from its environment through a learning process
  - Interneuron connection strengths known as synaptic weights are used to store the acquired knowledge.

Simon Haykin. Neural Networks and Machine

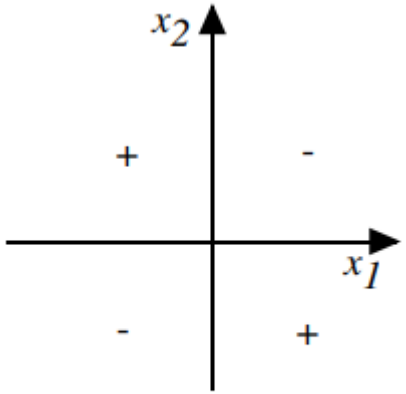
Decision Surface of a Perceptron

- Represents some useful functions



(a)

- But some functions are not linearly separable
- XOR



(b)

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

- Can prove it will converge if
  - Training data is linearly separable
  - And η is sufficiently small

Gradient Descent – The Delta Rule

- For a simple **linear** unit
  - Output o = $w_0 + w_1x_1 + w_2x_2 + \ldots\ldots + w_nx_n$
- Objective
  - Learn the weights w_i's that minimise the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

  - Where D is the set of training examples

Gradient Descent – The Delta Rule

- Gradient

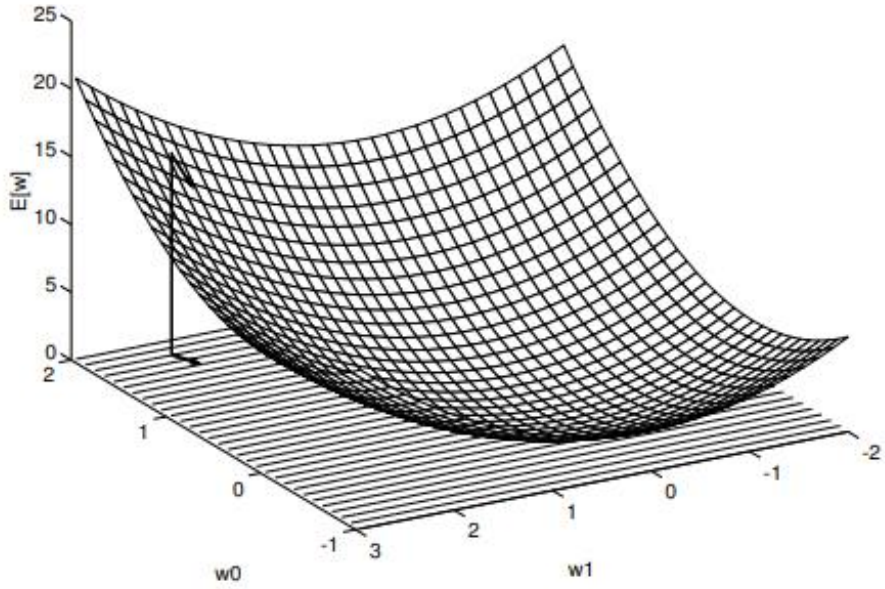
$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

- Training Rule

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

- i.e

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$



Gradient Descent – The Delta Rule

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient Descent Algorithm

- Inputs: Labelled Training Examples, η
  - Each training example is a pair of the form $\langle \vec{x}, t \rangle$
  - Where $\vec{x}$ is a vector of input value
  - η set to a low value e.g. 0.05
- Initialise each w_i to a small random value
- Until termination condition satisfied, Do:
  - Initialise each Δw_i to 0
  - For each $\langle \vec{x}, t \rangle$ Do:
    - Input $\vec{x}$ to the unit and compute the output o
  - For each linear unit weight w_i Do:

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - Update each linear unit weight w_i

$$w_i \leftarrow w_i + \Delta w_i$$

Gradient Descent – The Delta Rule

- Perceptron training rule guaranteed to succeed if
  - Training examples are linearly separable
  - Sufficiently small learning rate η
- Linear unit training rule uses gradient descent
  - Guaranteed to converge to hypothesis with minimum squared error
  - Given sufficiently small learning rate η
  - Even when training data contains noise
  - Even when training data not separable by H

Incremental (Stochastic) Gradient Descent

Batch Mode Gradient Descent

- Do Until Terminating Condition Satisfied
  - Computer the gradient $\nabla E_d[\vec{w}]$
  - Update $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

Incremental Mode Gradient Descent

- Do Until Terminating Condition Satisfied
  - For each training example d in D
    - Compute the gradient $\nabla E_d[\vec{w}]$
    - Update $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

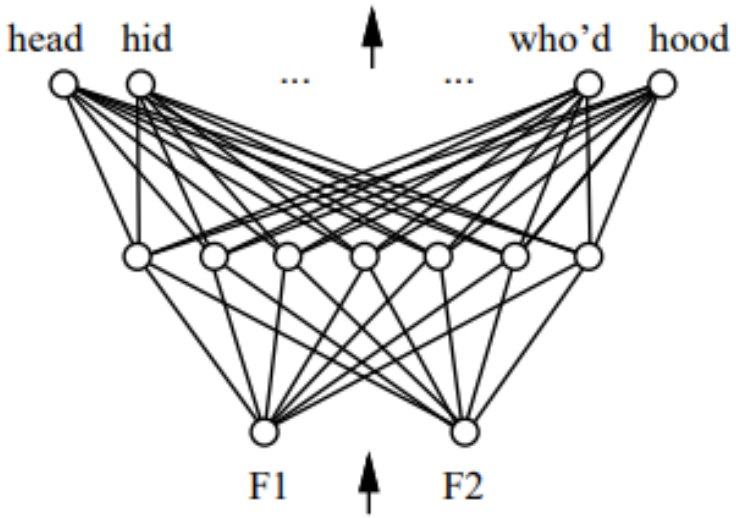
$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

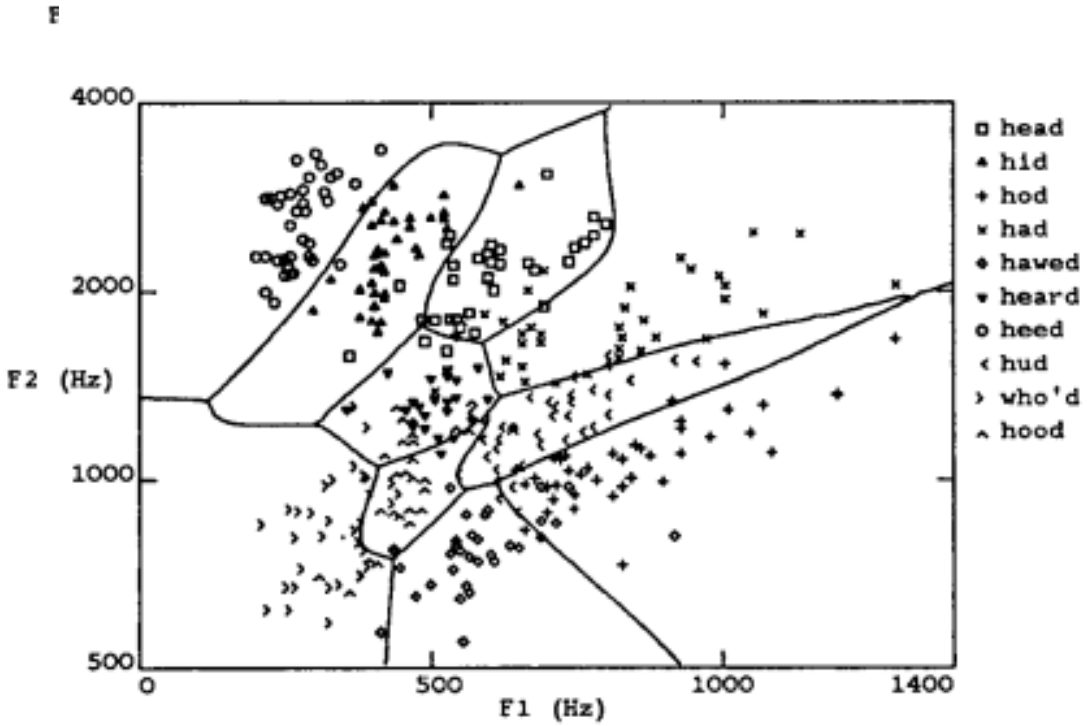
Part 3

The MultiLayer Perceptron (MLP) and Backpropagation (BP)

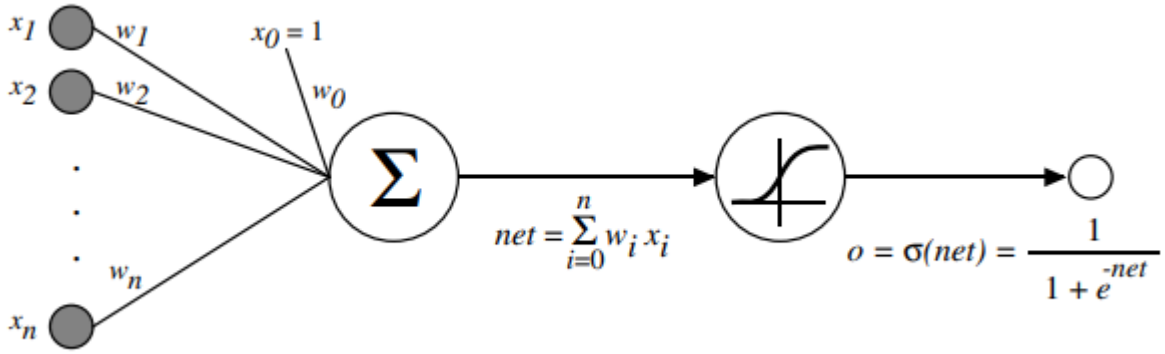
Multi-Layer Perceptron (MLP)

- Example: Speech Recognition





MLP



- The activation function $\sigma(x)$ uses a sigmoid.
  - It is continuous (differentiable) and non-linear!
- Use gradient descent to change the weights
- Property $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- We can derive rules to update weights of neurons in the hidden layer and output layer
- Known as Back Propagation (BP)

Backpropagation (BP) Algorithm

Initialise all weights w_i's to small random value

- Repeat until terminating condition satisfied
  - For each training example do:
    - Forward Pass: input the training example to the network and compute the outputs o_k's
    - For each output unit k $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$
    - For each hidden unit h $\delta_h \leftarrow o_h(1-o_h)\sum_{k\in outputs} w_{h,k}\delta_k$
    - Update $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$
    - where $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

Gradient Descent for MLP

- Chain Rule:
- $\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$
- Output neuron:
- $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta(t_j - o_j) o_j (1 - o_j)$
- Hidden Unit j
- $\delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$
- $\Delta w_{ji} = \eta \delta_j x_{ji}$
- For full derivation, see Tom Mitchell, chapter 4

Backpropagation

- Gradient Descent over entire network of weights (vector) → vector processing libraries and hardware units
- Can get caught in local minima
- Used with a weight momentum

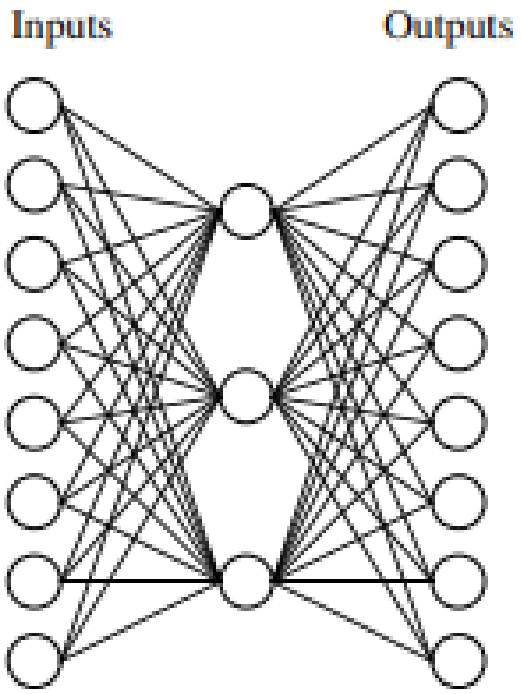
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimises error over training examples
- Will it generalise?
- Training can take thousands of iterations
- Using network after training is very fast.

Example

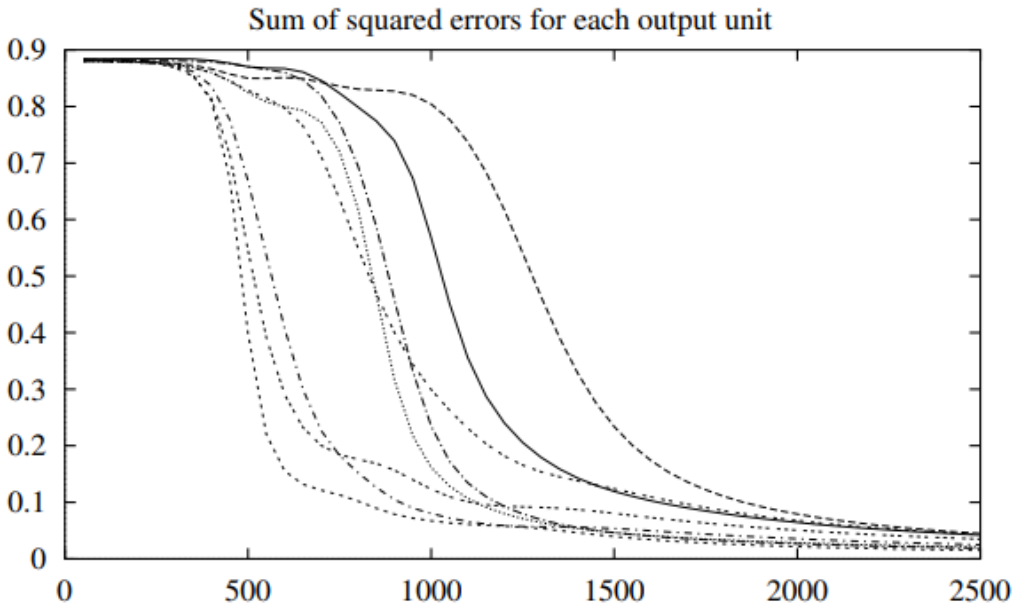
- Learn an input-output mapping

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001



Example

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

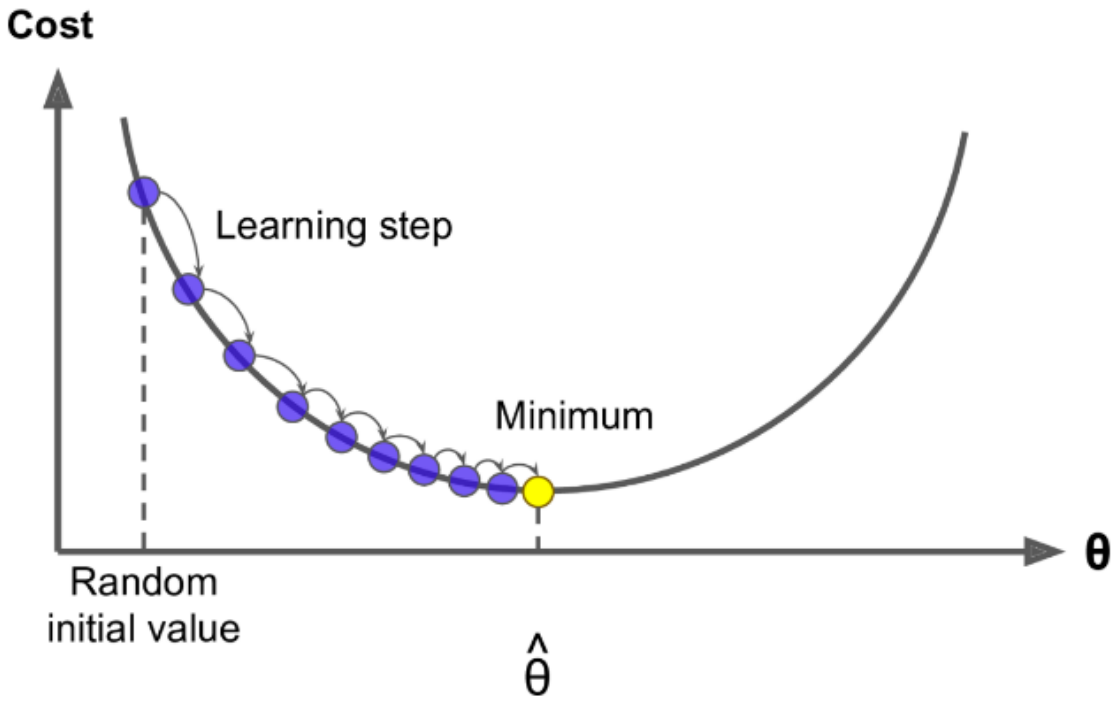


Part 4

Hyperparameters, Generalisation, Cross Fold Validation.

Learning Rate

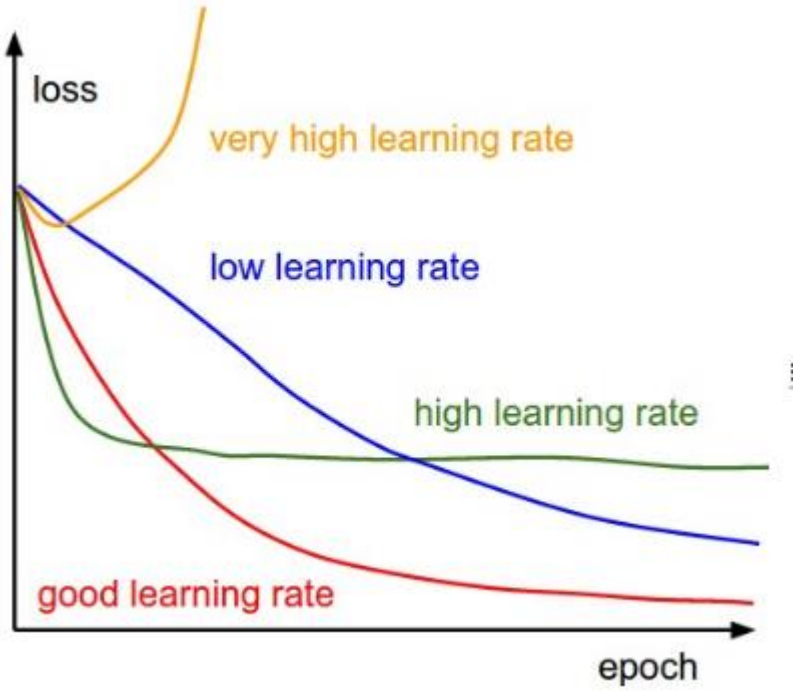
- The learning rate determines how much to change the weights in response to the estimated error.
- The estimated error known as Loss Function



https://www.i2tutorials.com/what-is-learning-rate-and-how-can-it-effect-accuracy-and-performance-in-neural-networks/

Learning Rate

- High learning rates perform well at first descending the error surface
- But frequently fail to find acceptable minima



https://cs231n.github.io/neural-networks-3/#loss

Momentum

- When partial derivative has same signs on consecutive iterations, $\Delta w_{j,i}$ grows in magnitude and weight is adjusted by a large amount
- When partial derivative has opposite signs on consecutive iterations, $\Delta w_{j,i}$ shrinks in magnitude and weight is adjusted by a small amount
- Stabilising effect
  - Moving average of our gradients
  - Skiing analogy
    - Skier going downhill builds up momentum
    - Momentum will propel skier out of a local depression

Observations on BP

- Gradient descent to some minima
  - Perhaps not the global minimum
  - Stochastic gradient descent
  - Train multiple nets with differing initial weights

- Weights referred to as the MODEL or parameters
- Hyper-parameters → number of layers, nodes per layer, connectivity, learning rate, momentum, loss function, optimiser, etc.
- Loss Function- Mean Squared Error (MSE), used to calculate gradient (slope).

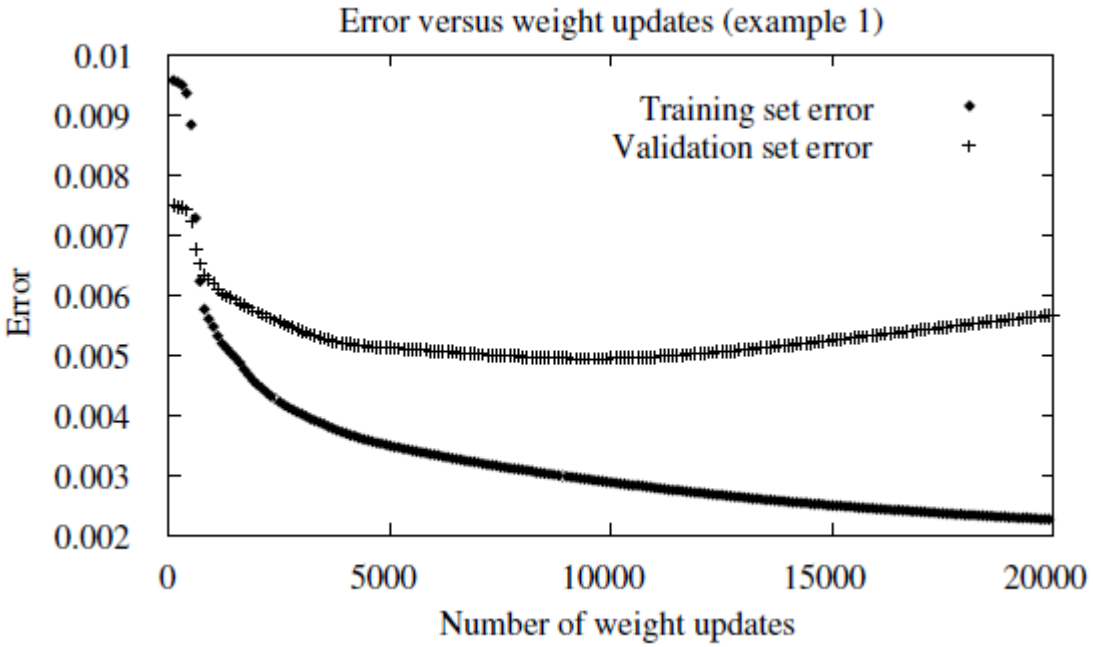
Approximation Capability

- Boolean Functions
  - Every Boolean Function can be represented by a network with a single hidden layer
  - But might require exponential hidden units
- Continuous Functions
  - Every bounded continuous function can be approximated with a small error by a network with one hidden layer [Cybenko 1989, Hornik et al., 1989].
  - Any function can be approximated to arbitrary accuracy by a network with 2 hidden layers [Cybenko 1988]

- An Artificial Neural Network (ANN) is a function approximator

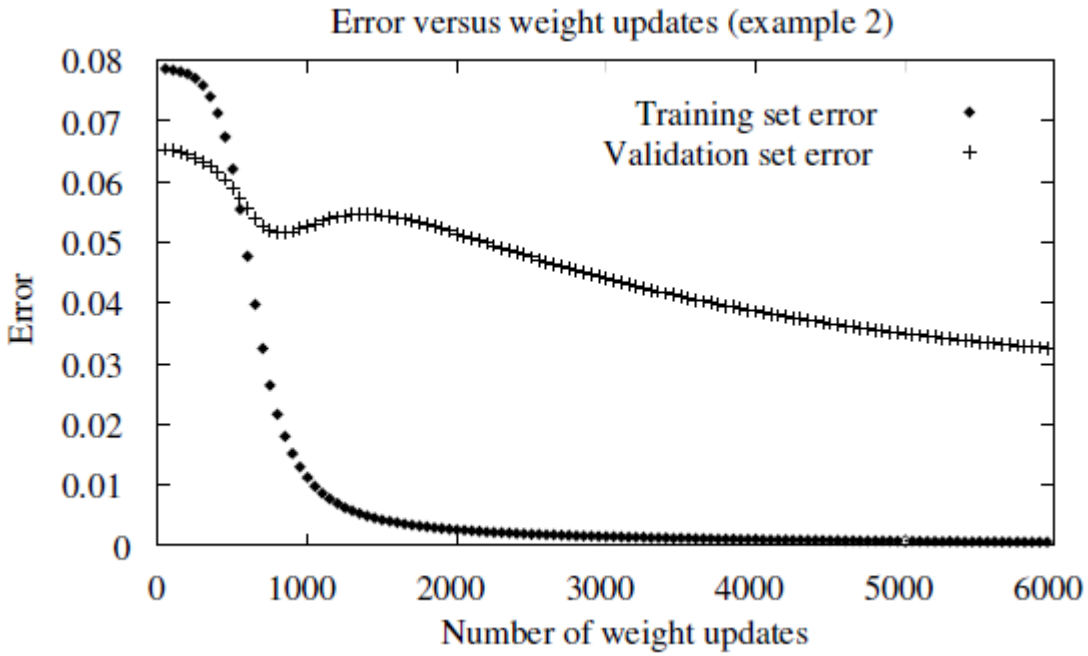
Overfitting

- Overfitting defined as poor Generalisation
  - Performs poorly on previously unseen data i.e. test set
  - ANN memorises training data
  - Training error depicted in bottom curve in fig below



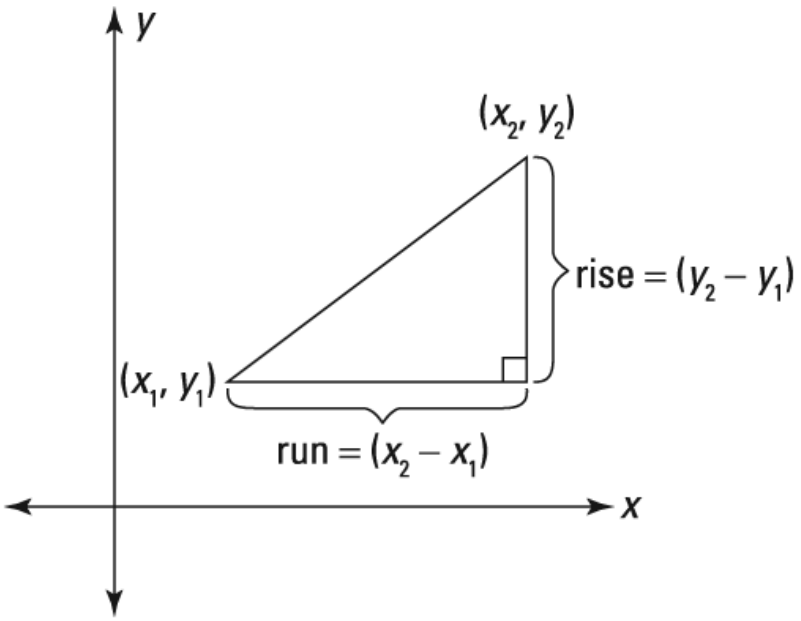
Cross Validation

- Cross Validation
  - Split data into training set and test set.
    - 3 fold cross validation set, 5 fold, etc.



Slope

- Slope = gradient = direction of the line
- $m = \frac{y2 - y1}{x2 - x1} = \frac{rise}{run}$
- $m = \frac{dy}{dx}$
- Function: y= x^2 $then$ $\frac{dy}{dx} = 2x$
- Differentiation
- Differentiate the error surface to find the slope/gradient
- Use the slope to descent to a minima.
- A partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant
  - Function = Error
  - Variables = weight



Slope is the ratio of the rise to the run.

Part 5

Michael Nielsen. Neural Networks and Deep Learning. Determination Press. 2015.

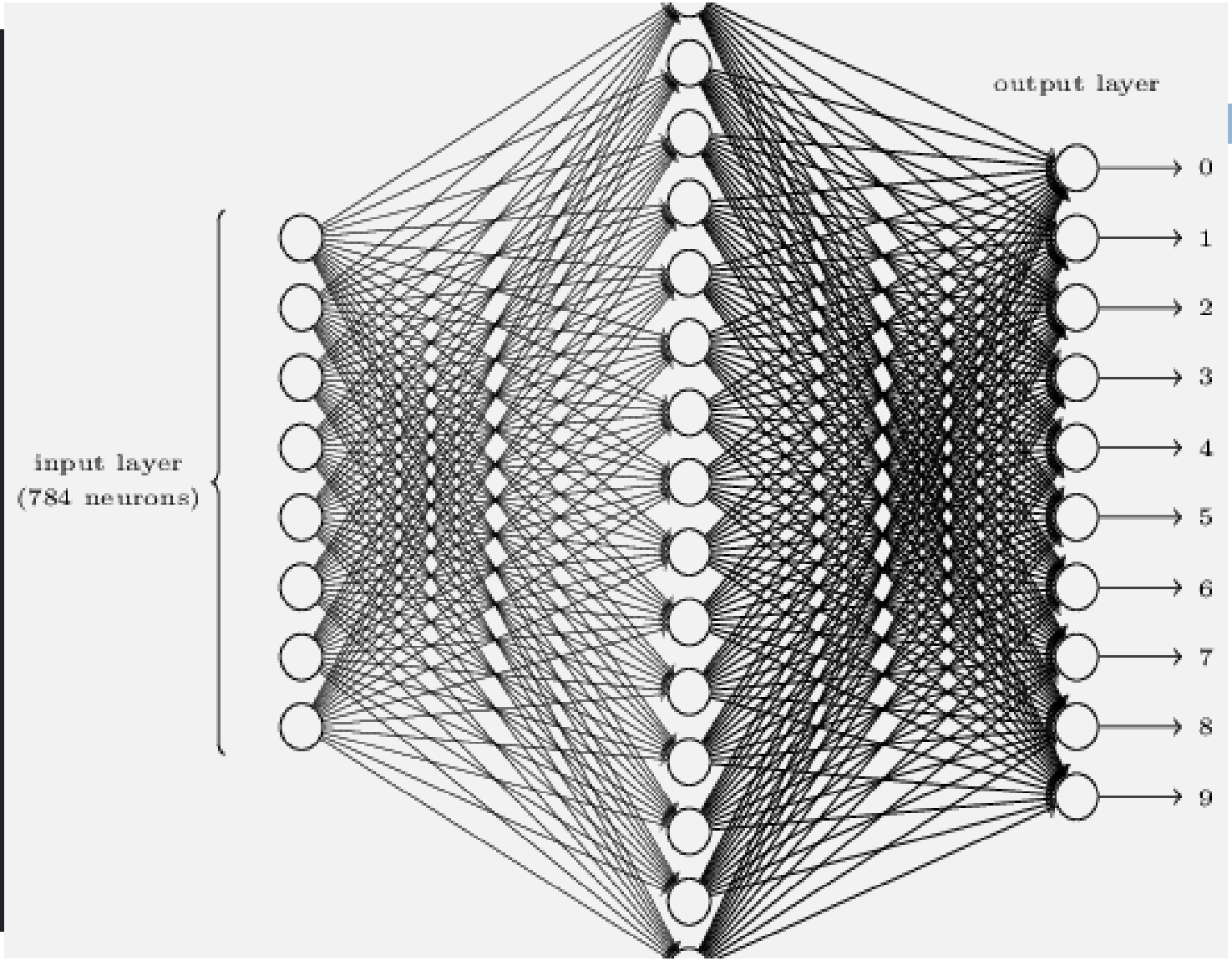
http://neuralnetworksanddeeplearning.com/

Nielsen's MLP Code for MNIST

http://yann.lecun.com/exdb/mnist

- The code splits the 60,000 MNIST data into two parts
- 50,000 images as training data set (to be used as before to do updates during training)
- 10,000 images as *validation* set; not used in training of the neural network
  - Used in figuring out/testing how to set certain *hyper-parameters* of the neural network such as learning rate, etc.
  - These parameters are 'manually' selected by developer and not directly selected by the learning algorithm

Nielsen's MLP Code for MNIST: the Network



A 3-layered traditional Multi-Layered Perceptron (MLP)

Nielsen's MLP Code for MNIST

- We use notation **x** to denote a training input (image)
- Regard each as a 28 by 28 = 748 dimension (column) *vector*
- Each component of the vector represents
  - the grey value for a single pixel in the image

A few images from MNIST

2. Nielsen's MLP Code for MNIST

- We use $\mathbf{y} = y(\mathbf{x})$ denoting the corresponding desired output where $\mathbf{y}$ is a 10-dimensional vector
- If training image $\mathbf{x}$ depicts digit '7, then $\mathbf{y} = y(\mathbf{x}) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ is the desired output

Nielsen's MLP Code for MNIST

NumPy package

- The NumPy package has an important data structure: *array* (internal class *ndarray*)
- Used to represent: 1-D array (vector), 2-D *matrix*, 3-D or higher-D array (tensor)
- In the book: sometimes "matrix" or "matrices" loosely refer to NumPy *array* generally and not just for 2-D
- Vectorising a function
- When a function e.g. sigmoid is called with an input argument that is a NumPy *array* (vector),
- NumPy automatically applies the function sigmoid *element-wise*, i.e. in *vectorised* form.

$$a = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \sigma(z) = \sigma\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right) = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \end{bmatrix}$$

Nielsen's MLP Code for MNIST

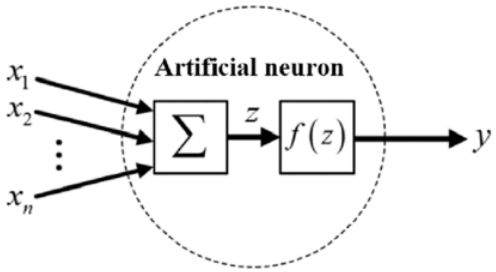
- Let z = $\sum_k w_k x_k + b$
- z= $\mathbf{w} \cdot \mathbf{x} + b$
- **w** is the weight vector consisting of components

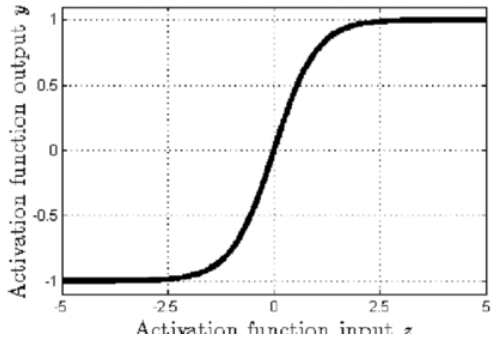
$\{w_1, w_2, \ldots, \}$

- **x** is the input vector consisting of components
- $\{x_1, x_2, \ldots, \}$
- Inner product = $\mathbf{w} \cdot \mathbf{x} = \sum_k w_k x_k$
- In matrix form dot product
- $\mathbf{w}^T \cdot \mathbf{x} = \sum_k w_k x_k$

Nielsen's MLP Code for MNIST

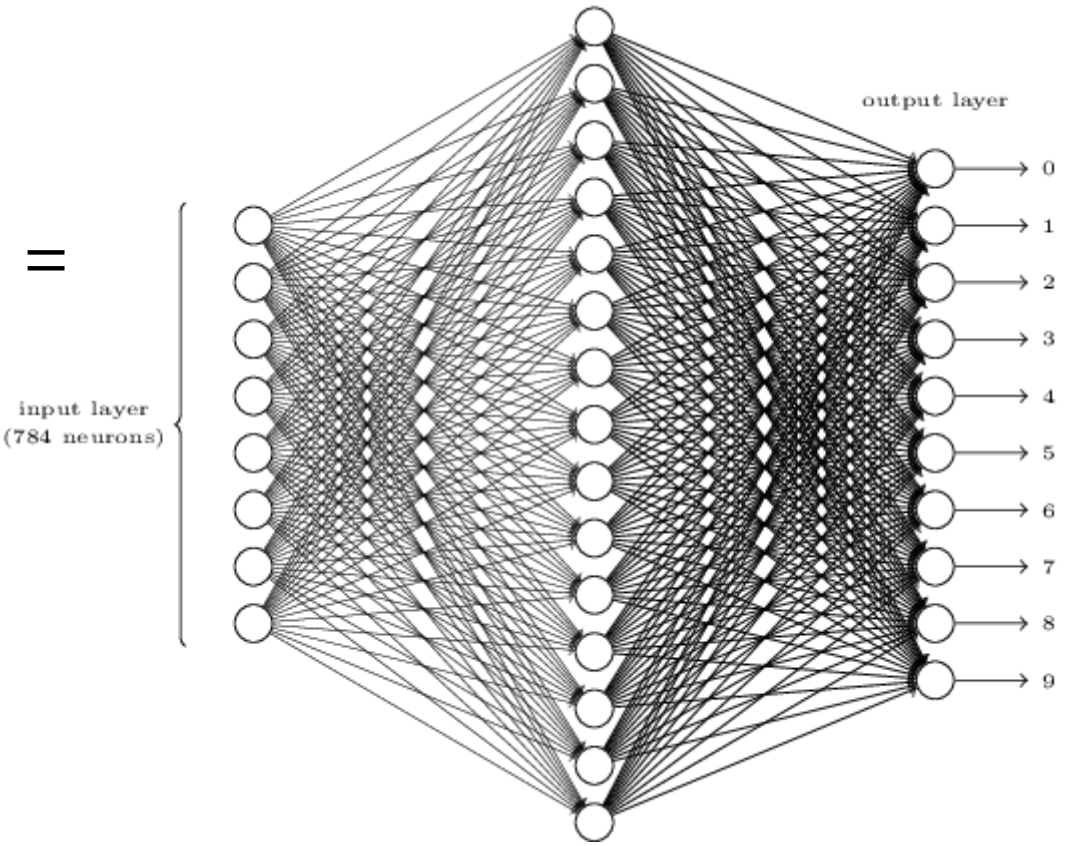
- Output given by sigmoid activation function
- It is a function of z which is $\mathbf{w} \cdot \mathbf{x} + b$
- $\sigma(z)$ or $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$
- Also called logistic function
- Output $a = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$
- $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+exp\left(-\sum_k w_k x_k - b\right)}$





2. Nielsen's MLP Code for MNIST

- Input Layer: Layer 0.
  - Not really neurons, for notational convenience
  - MNIST input = 28 x 28 = 784 inputs (neuros) in Layer 0
- Hidden Layer: Layer 1
  - 30 sigmoid neurons
- Output Layer: Layer 2
  - 10 neurons



Nielsen's MLP Code for MNIST

- The Network Class

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

- Use `Network` class to represent a neural network; it has `sizes` as an instance attribute
  - `sizes` is a *list* that contains the *number* of neurons in the respective layers
  - After creating a **`net`** `Network` object with: *784* neurons in 1st layer, *30* in 2nd, *10* in 3rd then ***`sizes`*** is `[784, 30, 10]`

```
# network.py example
import network
net = network.Network([784, 30, 10])
```

Nielsen's MLP Code for MNIST

- The Network Class

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

- The `biases` and `weights` are instance attributes stored as *lists* of NumPy matrices
- The list `sizes[1:]` is `[30, 10]` The list `sizes[:-1]` is `[784, 30]`
  The list `zip(sizes[:-1], sizes[1:])` is `[(784, 30), (30, 10)]`
- `biases` is list of two matrices: [30 by 1 matrix, 10 by 1 matrix]
  `weights` is list of two matrices: [30 by 784 matrix, 10 by 30 matrix]
- Denote: $w^{l=3rd\ layer}$ = `weights[1]` a 10 by 30 matrix

Nielsen's MLP Code for MNIST

- The Network Class

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

- `biases` and `weights` are initialized randomly to give our algorithm a starting point; none needed for neurons in first layer because it is an input layer
  - The NumPy `np.random.randn(y,x)` function creates a NumPy *matrix* of *shape [y, x]* and fills it with random numbers (Gaussian distributions with mean of 0 and sd of 1)
  - The Python `zip(list1, list2)` function returns a *list* of *tuples*
    - Each tuple obtains elements from the respective elements in `list1` and `list2`

Nielsen's MLP Code for MNIST

- The Network Class

Reminder: $w^{l=3rd\ layer}$ = `weights[1]` a 10 by 30 matrix

- As we'll see, it stores the weights associated with the neurons in the third layer
- We can write the *output* **activation** of third layer: $a^{l=3} = \sigma\left(w^{l=3}\, a^{l=2} + b^{l=3}\right)$

$$a^{l=3} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,k}^{l=3} & \cdots & \cdots & w_{1\,30}^{l=3} \\ w_{j\,1}^{l=3} & w_{j\,k}^{l=3} & \cdots & \cdots & w_{j\,30}^{l=3} \\ \vdots & & & & \\ w_{10\,1}^{l=3} & w_{10\,k}^{l=3} & \cdots & \cdots & w_{10\,30}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_k^{l=2} \\ \vdots \\ \vdots \\ a_{30}^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \\ b_j^{l=3} \\ \vdots \\ b_{10}^{l=3} \end{bmatrix}\right)$$

- $w_{jk}^{l=3} = weights[1]_{jk}$ is one of the weights of the j^{th} neuron in the *third* layer, in particular, the weight for the output from the k^{th} neuron in the *second* layer

Nielsen's MLP Code for MNIST

- The Network Class

$$a^{l=3} = \sigma\left(w^{l=3}\, a^{l=2} + b^{l=3}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,k}^{l=3} & \cdots & \cdots & w_{1\,30}^{l=3} \\ w_{j\,1}^{l=3} & w_{j\,k}^{l=3} & \cdots & \cdots & w_{j\,30}^{l=3} \\ \vdots & & & & \\ w_{10\,1}^{l=3} & w_{10\,k}^{l=3} & \cdots & \cdots & w_{10\,30}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_k^{l=2} \\ \vdots \\ \vdots \\ a_{30}^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \\ b_j^{l=3} \\ \vdots \\ b_{10}^{l=3} \end{bmatrix}\right)$$

- Denote: $b^{l=3}$ = `biases[1]` a 10 by 1 matrix
  - $b_j^{l=3} = biases[1]_j$ is the bias for the j^{th} neuron in the third layer

Nielsen's MLP Code for MNIST

The Network Class

- For neurons in the second (hidden layer)
- $w_{jk}^{l=2} = weights[0]_{jk}$
  - Weight for the jth neuron in the hidden layer applied to the output of the kth neuron in the input layer
- $b_j^{l=2} = bias[0]_j$
  - Bias of the jth neuron in the second layer

Nielsen's MLP Code for MNIST

The Network Class

- $z^l = w^l a^{l-1} + b^l$
- $z^{l+1} = w^{l+1} a^l + b^{l+1}$
- Define the sigmoid

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

- Define the derivative of the sigmoid for gradient descent
- $\sigma'(z) = \sigma(z)\,(1-(\sigma(z))$

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

Nielsen's MLP Code for MNIST

The Network Class

- Add a feedforward method

```
def feedforward(self, a):
    """Return the output of the network if "a" is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

- This method is only used by `self.evaluate(test_data)` for accuracy after each epoch.
- Warning from the book: the input a is an (n,1) NumPy ndarray, not a (n,) vector.
  - Using an (n,) vector will generate strange outputs.

Nielsen's MLP Code for MNIST

The Network Class

In each epoch

- Randomly shuffles training data
- Partitions it into mini-batches of `mini_batch_size`
- Apply Gradient Descent to each mini-batch using
- `self.update_mini_batch(mini_batch,eta)`
- Which updates the weights and biases using the training data in mini_batch

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful fo
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

Nielsen's MLP Code for MNIST

The Network Class

- Uses Stochastic Gradient Descent (SGD)
- The training data is a list of tuples (x,y) → inputs and desired outputs for each
- Params: epochs, minibatches, and learning rate eta

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful fo
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

Nielsen's MLP Code for MNIST

The Network Class

- nabla = ∇ = partial derivative
- `delta_nabla_b, delta_nabla_w = self.backprop(x, y)` runs the backprop phase
  - Work through this in your own time
- Returns a tuple: a list of matrices holding the sum for
- $\frac{\delta C_x}{\delta b_j^l}$ and $\frac{\delta C_x}{\delta w_{jk}^l}$ and
- And then updates weights and biases x by learning rate η (*eta*)

```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

Nielsen's MLP Code for MNIST

- Run test.py
- And then fine tune the parameters
- And then Network2.py

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.
training_data = list(training_data)

# - network.py example:
import network

net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Exercises

Who is credited with the invention of BP?

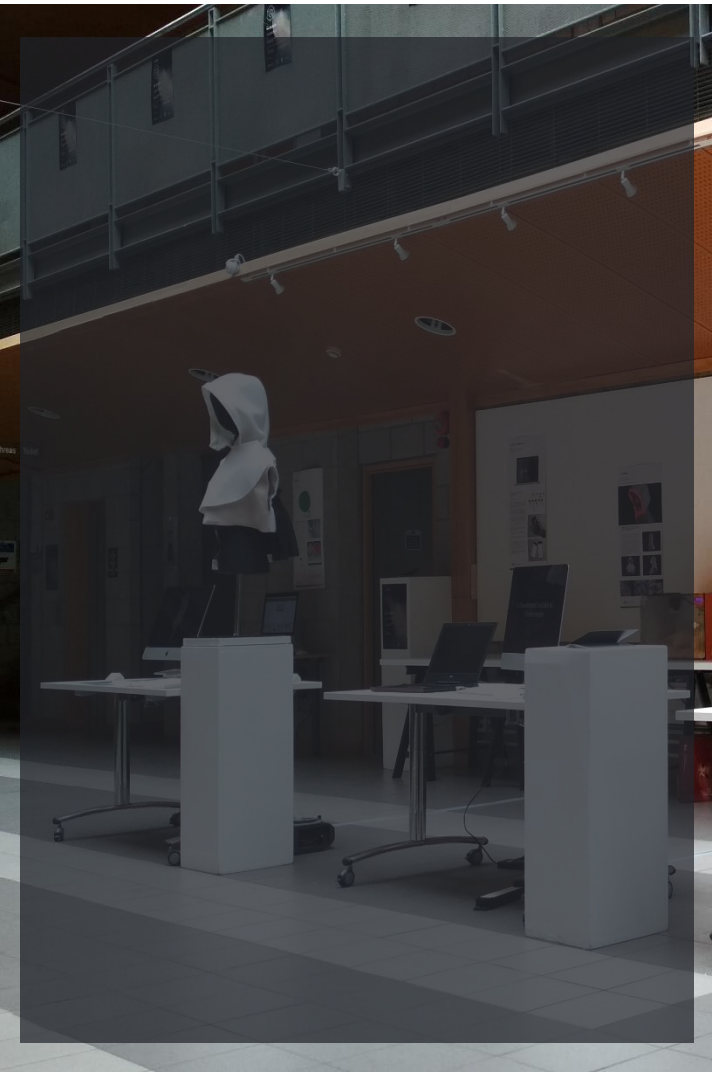
Understand derivation of BP, see Tom Mitchell. Machine Learning. McGraw Hill. 1997.

Read Nielsen's online text Deep Learning

Run Nielsen's MLP code for MNIST

Investigate impact of varying parammeters and hyperparameters

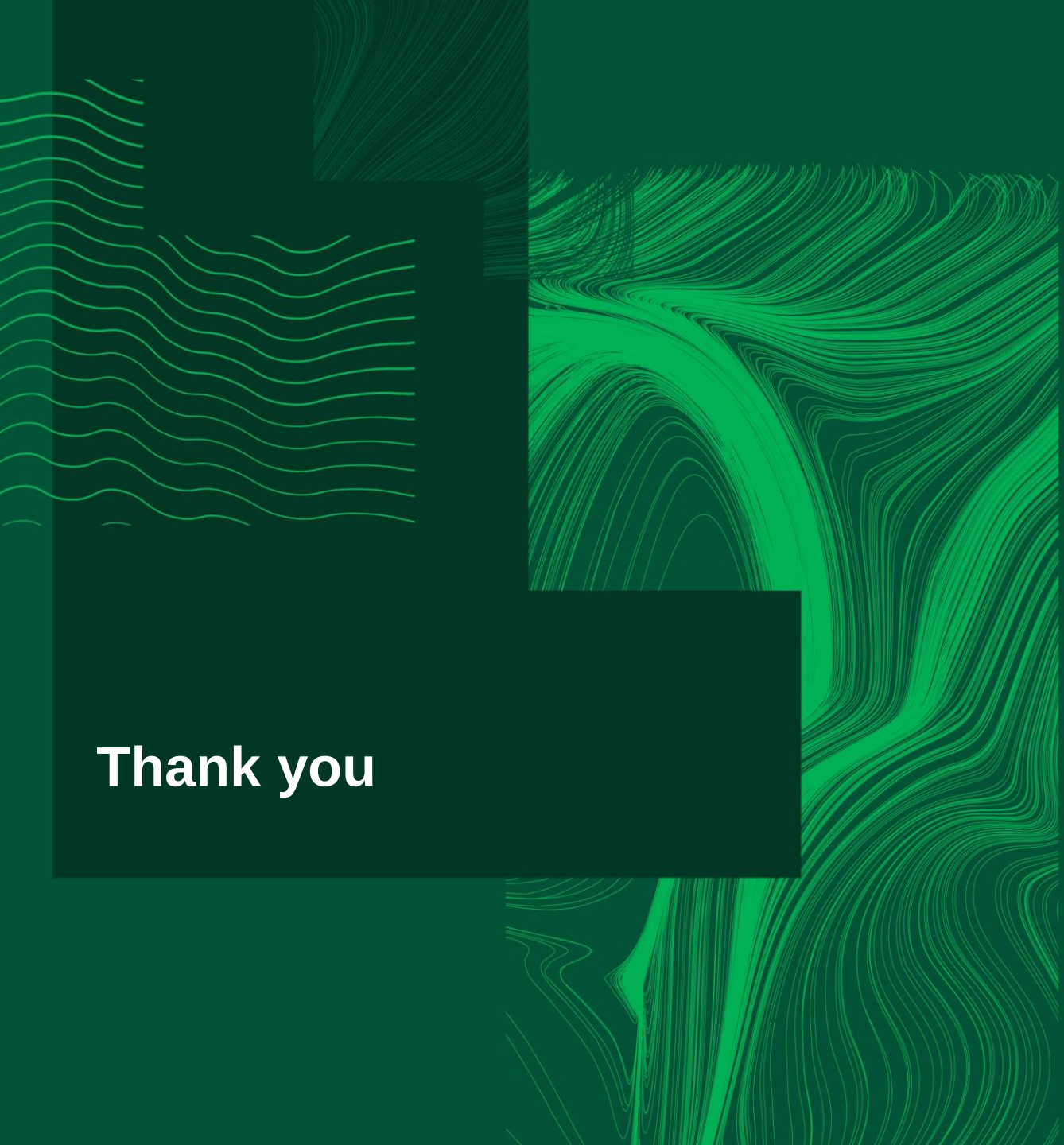
Implement an MLP to compute XOR



Reading for Week 3

- Chapters 1 and 2 in Geron – available online through UL Library catalogue here.
- Chapters 1- 4 in Nielsen
  - http://neuralnetworksanddeeplearning.com/
- 1990 Paper by LeCun et al. on Character Recognition using MLP
  - https://ieeexplore.ieee.org/abstract/document/119325

Thank you



UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

University of Limerick,
Limerick, V94 T9PX,
Ireland.

Ollscoil Luimnigh,
Luimneach,
V94 T9PX, Éire.

+353 (0) 61 202020

ul.ie

3 LeCun et al.

- 1990 Paper by LeCun et al. on Character Recognition using MLP
  - https://ieeexplore.ieee.org/abstract/document/119325