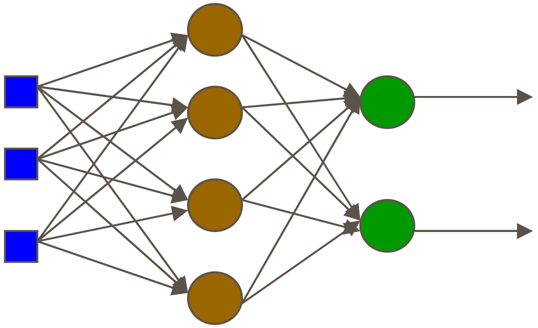
Neural Networks

What Are Neural Networks

- Simple computational elements forming a large network
  - Emphasis on learning (pattern recognition)
  - Local computation (neurons)

- Definition of NNs is vague
  - Often (but not always) inspired by biological brain



History

Roots of work on NN are in:

- **Neurobiological studies** (more than one century ago):
  - How do nerves behave when stimulated by different magnitudes of electric current?
  - Is there a minimal threshold needed for nerves to be activated?
  - Given that no single nerve cell is long enough, how do different nerve cells communicate with each other?
- **Psychological studies**:
  - How do animals learn, forget, recognize and perform other types of tasks?
- **Psycho-physical** experiments helped to understand how individual neurons and groups of neurons work.
- **McCulloch and Pitts** introduced the first mathematical model of single neuron, widely applied in subsequent work.

NN Topics

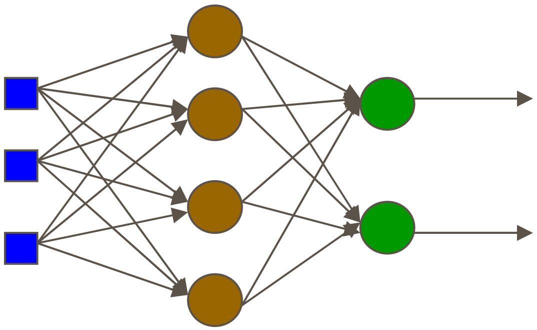
Supervised

- Data: Labeled examples
  - Pairs (input , desired output)
- Tasks:
  - Classification
  - Regression
- NN models:
  - Perceptron
  - Adaline
  - Feed-forward NN
  - Radial Basis Functions
  - Support Vector Machines

Unsupervised

- Data: Unlabeled examples
  - Different realizations of the input
- Tasks:
  - Clustering
  - Content addressable memory
- NN models:
  - Self-organizing Maps (SOM)
  - Hopfield networks

NNs: Goal and Design



A NN is specified by:

- An architecture: a set of neurons and links connecting neurons. Each link has a weight,
- A neuron model: the information processing unit of the NN,
- A learning algorithm: used for training the NN by modifying the weights in order to solve the particular learning task correctly on the training examples.

The aim is to obtain a NN that generalizes well

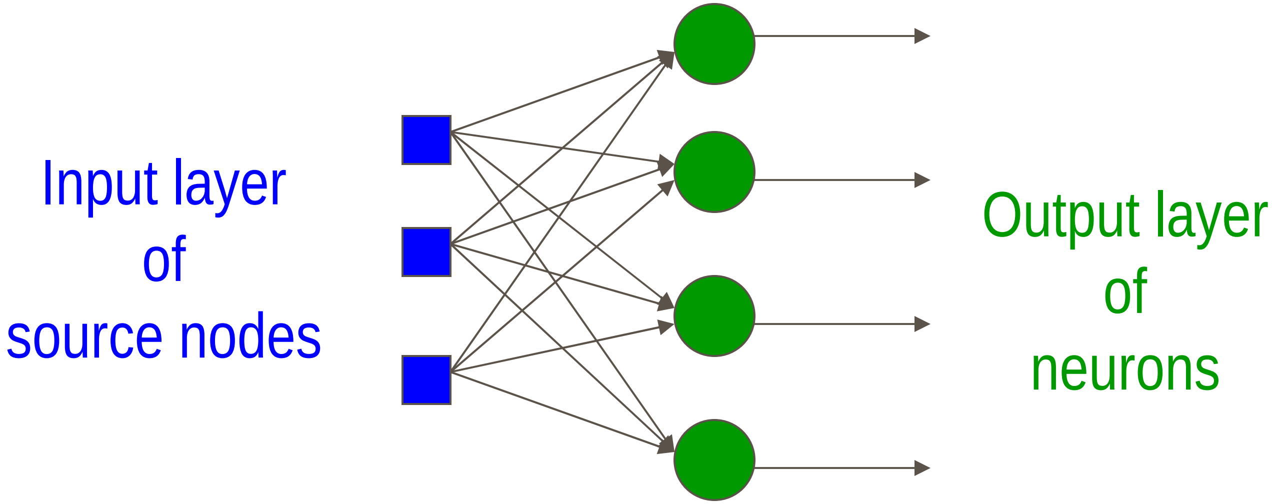
Network Architectures

- Three different classes of network architectures
  - single-layer feed-forward
  - multi-layer feed-forward
  - recurrent

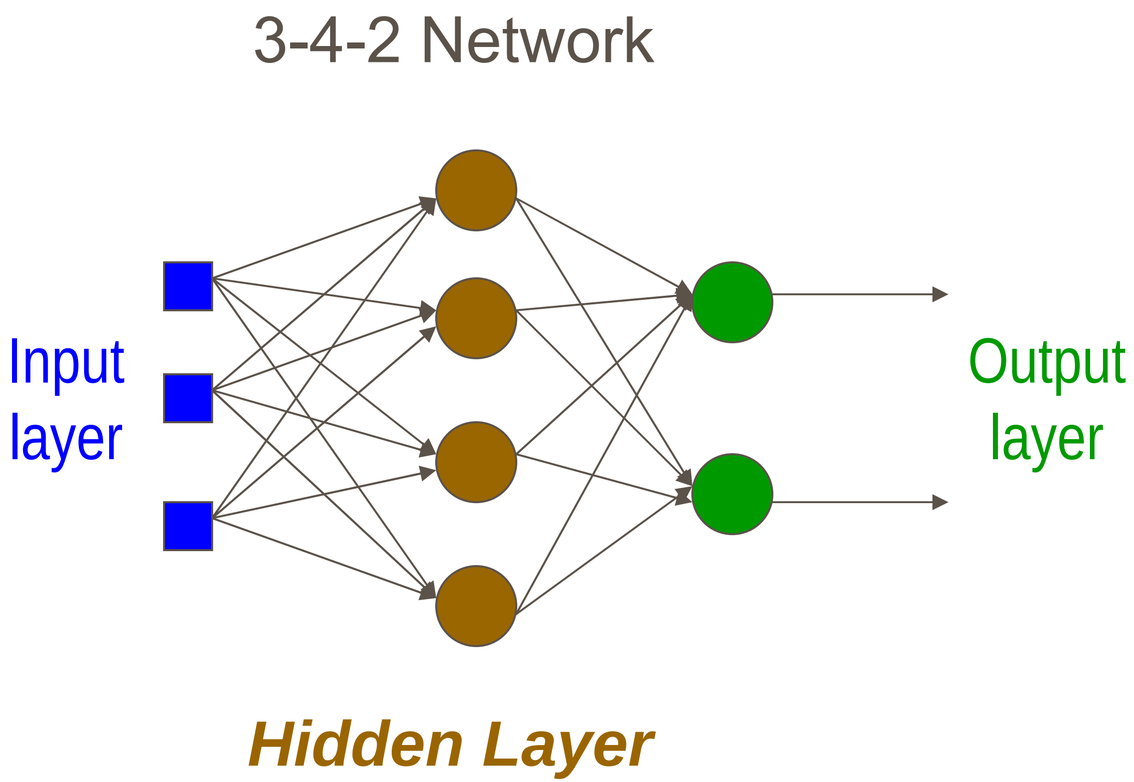
  neurons are organized in acyclic layers

- The architecture of a neural network is linked with the learning algorithm used to train

Single Layer Feed-Forward NN

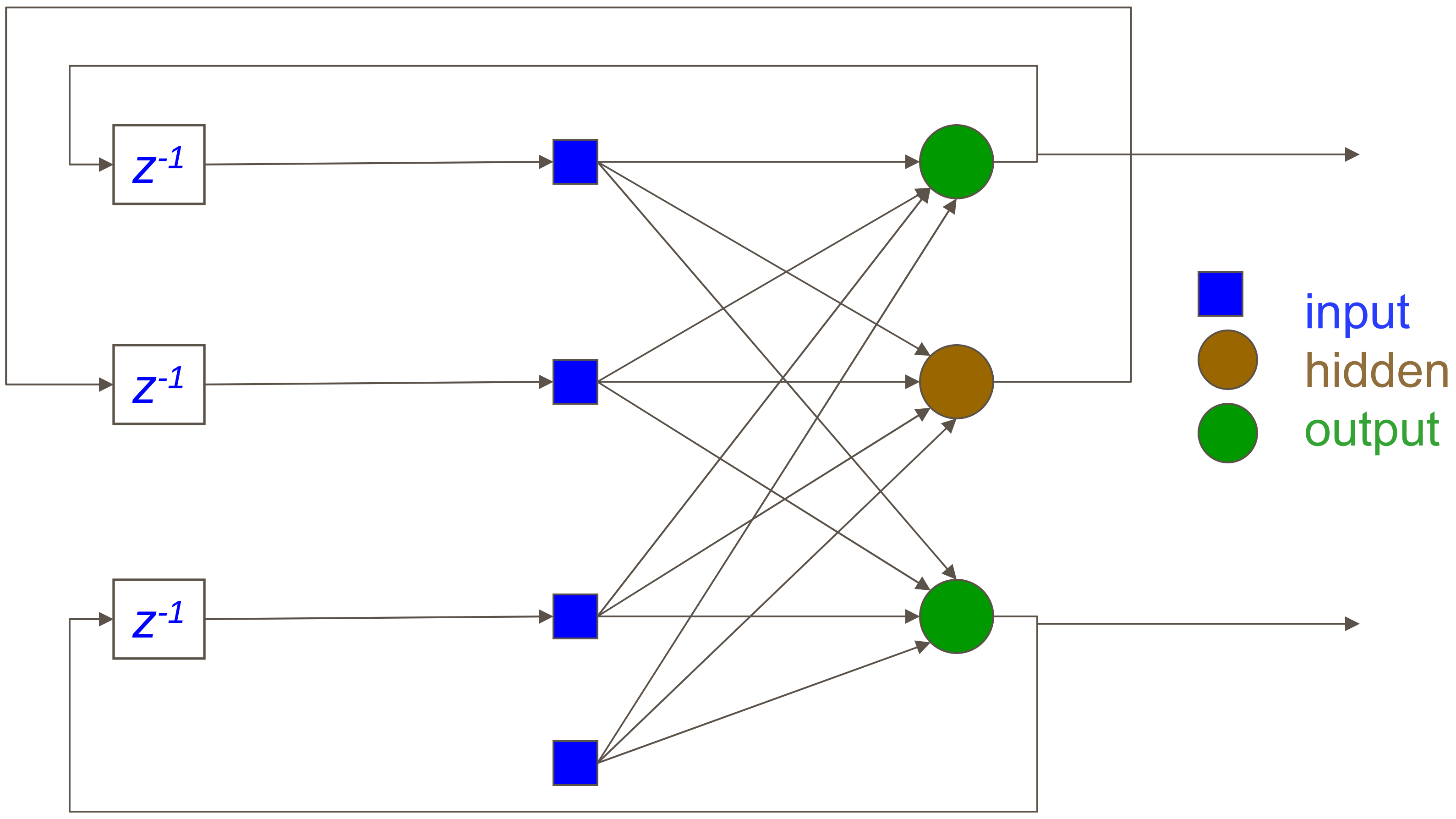


Multi Layer Feed-Forward

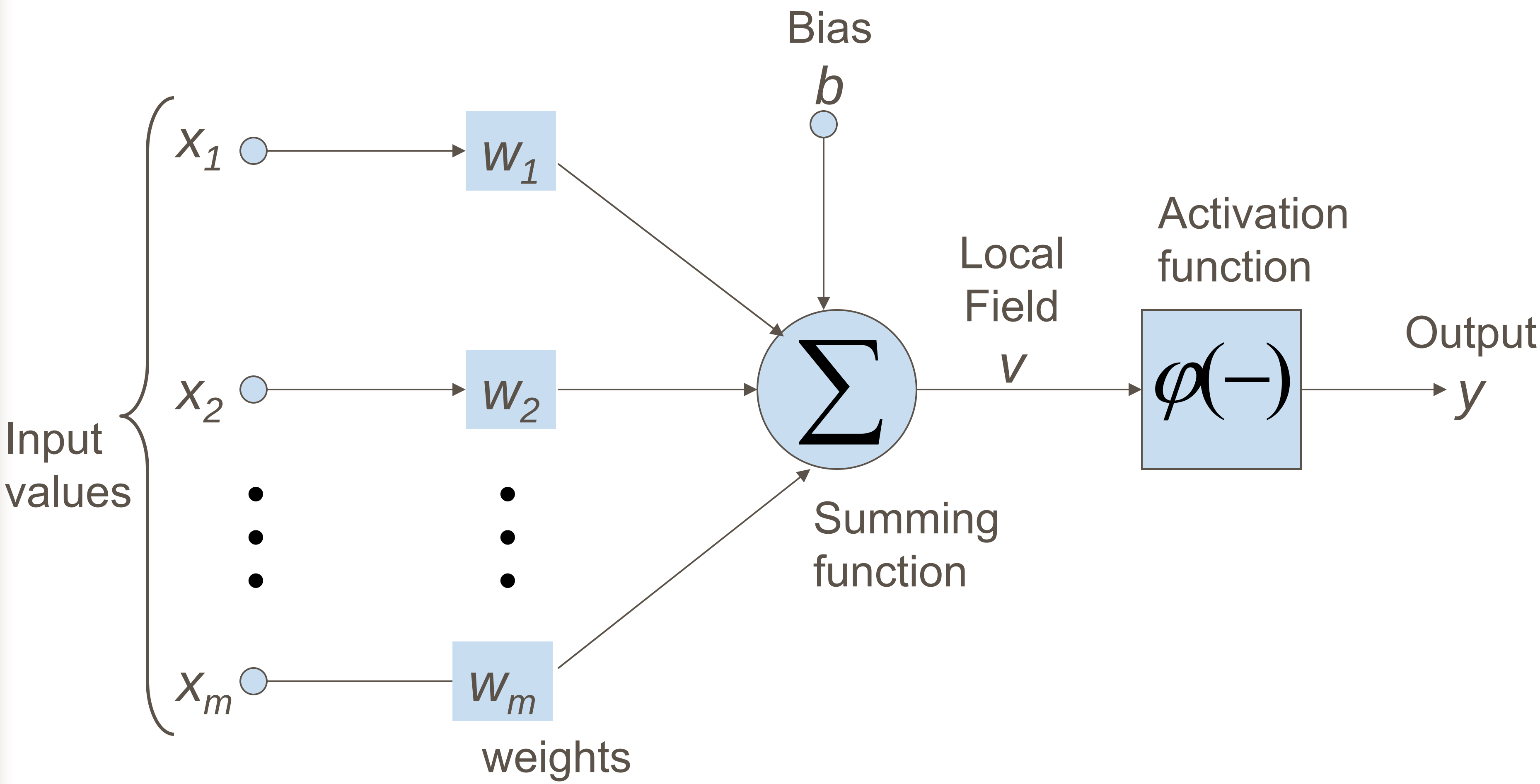


Recurrent Network

Recurrent Network with hidden neuron: unit delay operator z^{-1} is used to model a dynamic system



The Neuron


Bias
b
x1
x2
xm
w1
w2
wm
Input values
weights
Local Field
v
Summing function
Activation function
φ(−)
Output
y

The Neuron

- The neuron is the basic information processing unit of a NN.
- It consists of:
  1. A set of links, describing the neuron inputs, with weights $w_1, w_2, \ldots, w_m$
  2. An adder function (linear combiner) for computing the weighted sum of the inputs (real numbers):

$$v = \sum_{j=1}^{m} w_j x_j$$

  3. Activation function (squashing function) φ for limiting the amplitude of the neuron output.

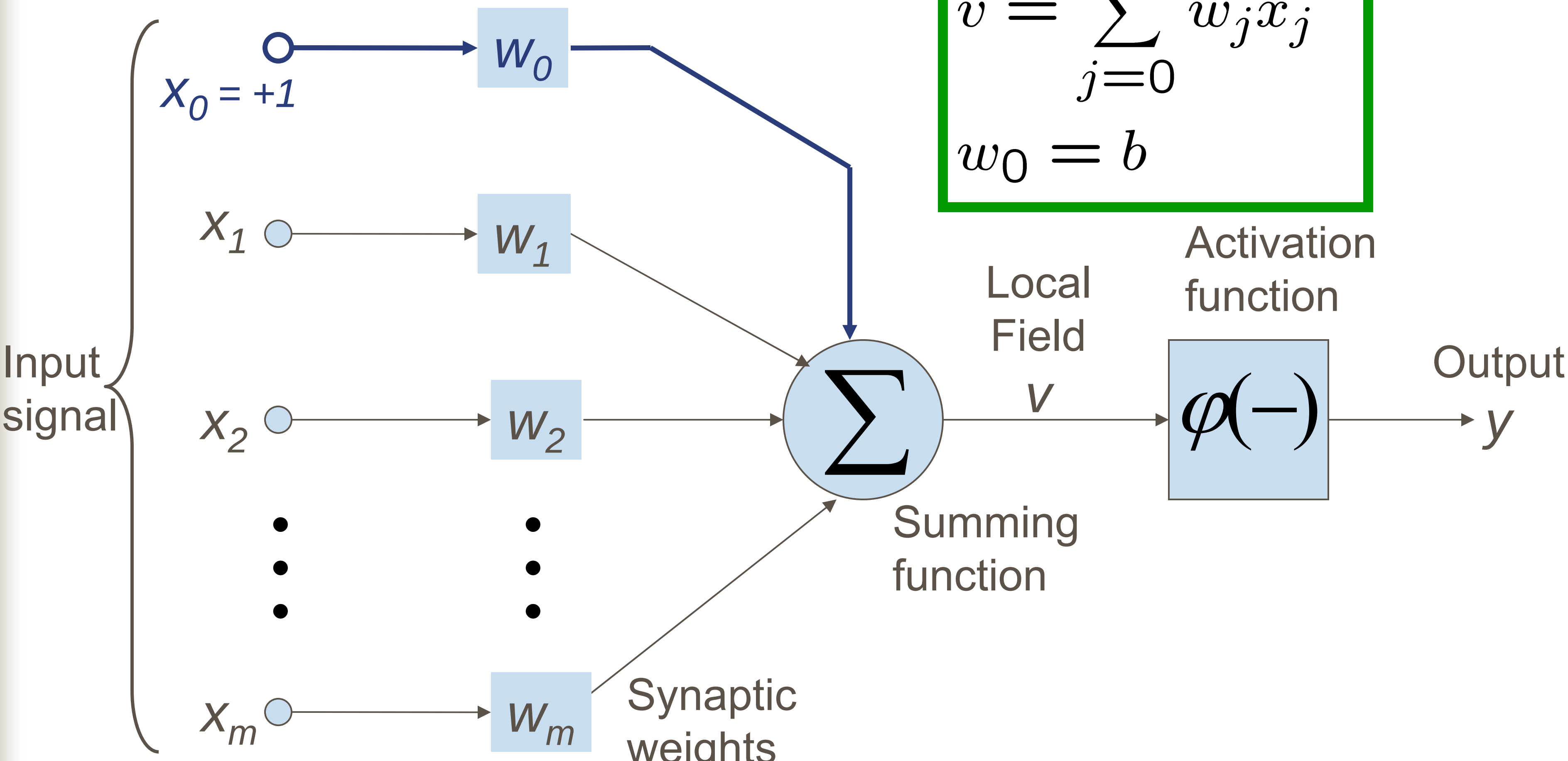
$$y = \varphi(v + b)$$

Bias as Extra Input

- The bias is an external parameter of the neuron. It can be modeled by adding an extra input.

$$v = \sum_{j=0}^{m} w_j x_j$$

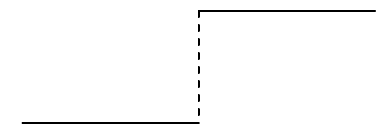
$$w_0 = b$$



Neuron Models

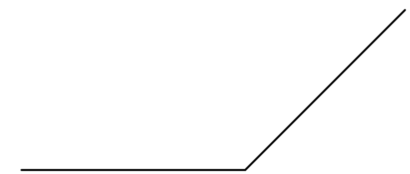
The choice of φ determines the neuron model. Examples:

- Step function: $$\varphi(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > c \end{cases}$$



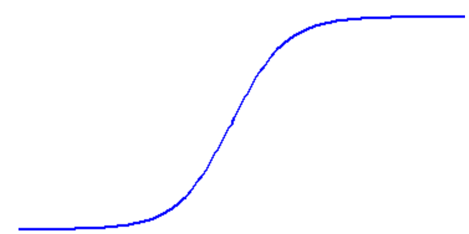
- Rectified Linear (ReLU) function:

$$\varphi(v) = \begin{cases} 0 & \text{if } v < 0 \\ v & \text{otherwise} \end{cases}$$



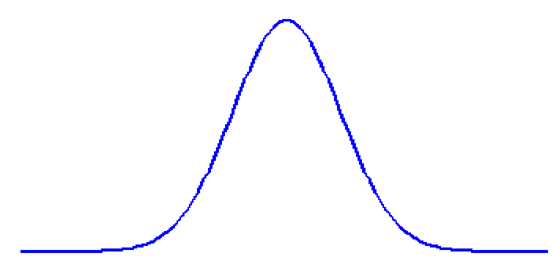
- Sigmoid function with z,x,y parameters

$$\varphi(v) = z + \frac{1}{1 + \exp(-xv + y)}$$



- Gaussian function:

$$\varphi(v) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2} \left(\frac{v - \mu}{\sigma} \right)^2 \right)$$



Learning Algorithms

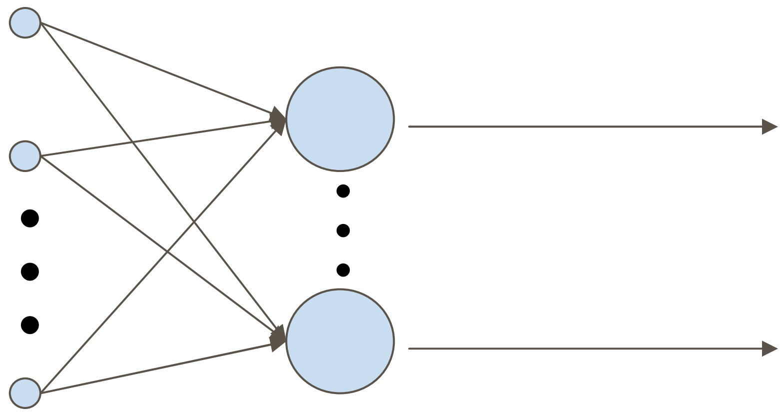
- Depend on the network architecture:
  - Error correcting learning (Perceptron)
  - Delta rule (AdaLine, Backpropagation)
  - Competitive Learning (Self Organizing Maps)

Applications

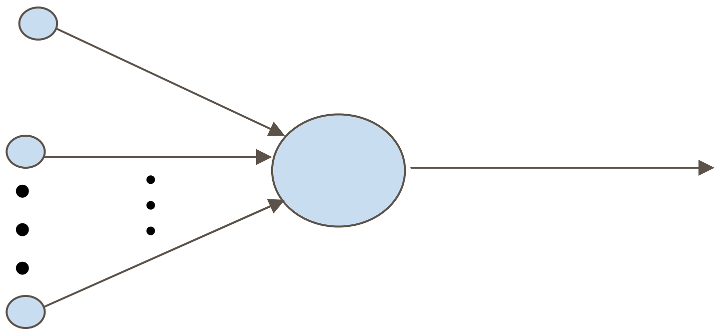
- Classification:
  - Image recognition
  - Speech recognition
  - Medical diagnostics
  - Fraud detection
- Regression:
  - Forecasting (prediction on base of past history)
- Pattern association:
  - Retrieve an image from a corrupted one: denoising, inpainting
- Clustering:
  - Client profiles
  - Disease subtypes

Single Layer Perceptron

- Single Layer Perceptron = feed-forward NN with one layer



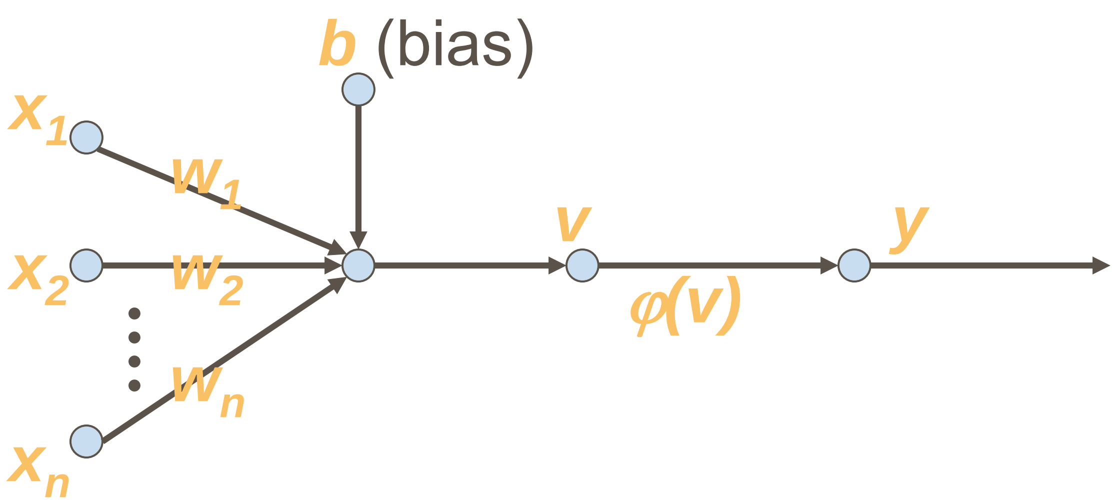
- The output units are independent of each other
- Each weight only affects one of the outputs
- It is sufficient to study single layer Perceptrons with just one neuron:



Perceptron: Neuron Model

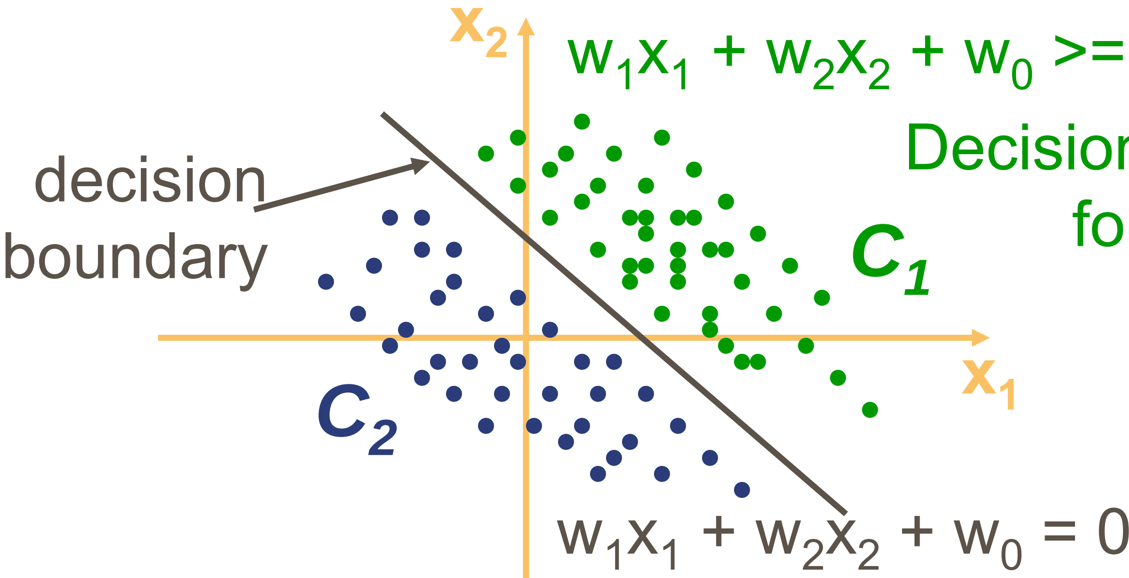
- The (McCulloch-Pitts) perceptron is a single layer NN with a non-linear φ, the sign function

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ -1 & \text{if } v < 0 \end{cases}$$



Perceptron for Classification

- The perceptron is used for binary classification.
- Given training examples of classes y=-1,y=1, train the perceptron in such a way that it classifies correctly the training examples:
  - The output of the perceptron is the class number
- Geometrically, we try to find a hyper-plane that separates the examples of the two classes.

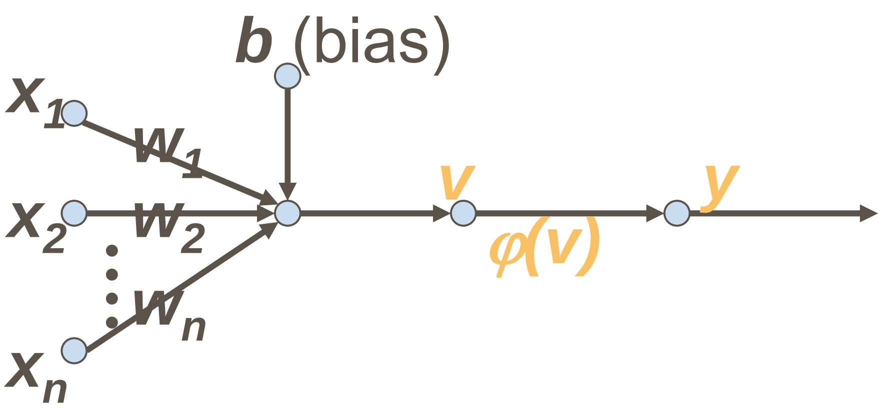


$$\sum_{i=1}^{m} w_i x_i + w_0 = 0$$

Perceptron Learning Algorithm

Variables and parameters at iteration n of the learning algorithm:

- x (n) = input vector
- $= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
- w(n) = neuron weight vector
- $= [b(n), w_1(n), w_2(n), \ldots, w_m(n)]^T$
- b(n) = bias
- y(n) = actual response
- d(n) = desired response
- η = learning rate parameter



Perceptron Learning Algorithm

n=1;

initialize w(n) randomly;

while (there are misclassified training examples)

  Select a misclassified example (x(n),d(n))

  w(n+1) = w(n) + η d(n)x(n);

  n = n+1;

end-while;

η = learning rate parameter (real number)

Example

- Consider the 2-dimensional training set $C_1 \cup C_2$,
- $C_1 = \{(1,1), (1, -1), (0, -1)\}$ with class label 1
- $C_2 = \{(-1,-1), (-1,1), (0,1)\}$ with class label -1

- Train a perceptron on $C_1 \cup C_2$

A Possible Implementation

Consider the augmented training set $C'_1 \cup C'_2$, with first entry fixed to 1 (to deal with the bias as extra weight):

(1, 1, 1), (1, 1, -1), (1, 0, -1) ,(1,-1, -1), (1,-1, 1), (1,0,1)

Replace x with -x for all $x \in C_2'$ and use the following update rule:

$$w(n+1) = \begin{cases} w(n) + \eta x(n) & \text{if } w^T(n)x(n) \leq 0 \\ w(n) & \text{else} \end{cases}$$

Epoch = application of the update rule to all examples of the training set. Execution of the learning algorithm terminates when the weights do not change after one epoch.

Epoch 1

- $w(1)=(1,0,0)$, $\eta=1$, and transformed inputs
  (1, 1, 1), (1, 1, -1), (1,0, -1), (-1,1, 1), (-1,1, -1), (-1,0, -1)

Adjusted pattern	Weight applied	w(n) x(n)	Update?	New weight
(1, 1, 1)	(1, 0, 0)	1	No	(1, 0, 0)
(1, 1, -1)	(1, 0, 0)	1	No	(1, 0, 0)
(1,0, -1)	(1, 0, 0)	1	No	(1, 0, 0)
(-1,1, 1)	(1, 0, 0)	-1	Yes	(0, 1, 1)
(-1,1, -1)	(0, 1, 1)	0	Yes	(-1, 2, 0)
(-1,0, -1)	(-1, 2, 0)	1	No	(-1, 2, 0)

End epoch 1

Epochs 2 and 3

Adjusted pattern	**Weight applied**	**w(n) ⊠(n)**	**Update?**	**New weight**
(1, 1, 1)	(-1, 2, 0)	1	No	(-1, 2, 0)
(1, 1, -1)	(-1, 2, 0)	1	No	(-1, 2, 0)
(1,0, -1)	(-1, 2, 0)	-1	Yes	(0, 2, -1)
(-1, 1, 1)	(0, 2, -1)	1	No	(0, 2, -1)
(-1, 1, -1)	(0, 2, -1)	3	No	(0, 2, -1)
(-1,0, -1)	(0,2, -1)	1	No	(0, 2, -1)

End epoch 2

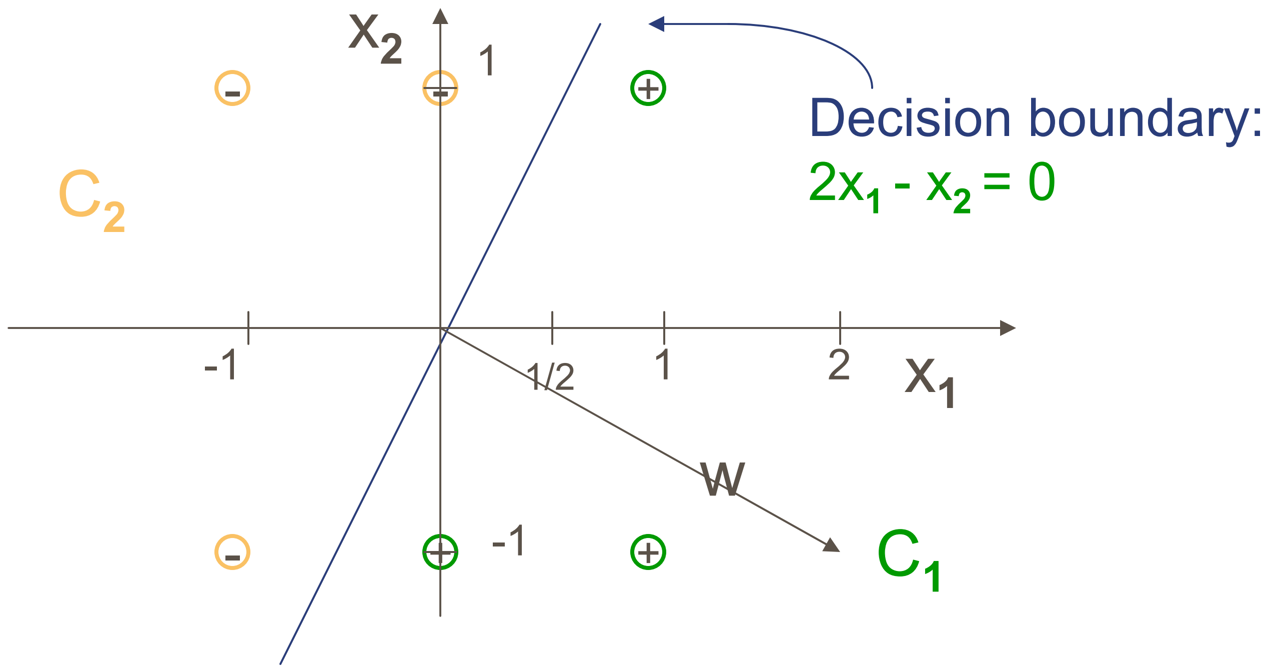
At epoch 3 no weight changes. (check!)

$\Rightarrow$ stop execution of algorithm.

Final weight vector: (0, 2, -1).

$\Rightarrow$ decision hyperplane is $2x_1 - x_2 = 0$.

Result



Termination of the Learning Algorithm

Suppose the classes C_1, C_2 are linearly separable (that is, there exists a hyper-plane that separates them). Then the perceptron algorithm applied to $C_1 \cup C_2$ terminates successfully after a finite number of iterations.

Proof:

Consider the set C containing the inputs of $C_1 \cup C_2$ transformed by replacing x with -x for each x with class label -1.

For simplicity assume $w(1) = 0$, $\eta = 1$.

Let $x(1) \ldots x(k) \in C$ be the sequence of inputs that have been used after k iterations. Then

$$\left.\begin{aligned} w(2) &= w(1) + x(1) \\ w(3) &= w(2) + x(2) \\ &\ \vdots \\ w(k+1) &= w(k) + x(k) \end{aligned}\right\} \Rightarrow w(k+1) = x(1) + \ldots + x(k)$$

Termination of the Learning Algorithm

- Since C_1 and C_2 are linearly separable then there exists w* such that $w_*^T \mathbf{x} > 0,\ \forall \mathbf{x} \in C$
- Let $\alpha = \min w_*^T \mathbf{x} > 0$
- Then

$$w_*^T w(k+1) = w_*^T \mathbf{x}(1) + \ldots + w_*^T \mathbf{x}(k) \geq k\alpha$$

- By the Cauchy-Schwarz inequality we get:

$$||w_*||^2 ||w(k+1)||^2 \geq (w_*^T w(k+1))^2$$

- So

$$||w(k+1)||^2 \geq \frac{(k\alpha)^2}{||w_*||^2}$$

Termination of the Learning Algorithm

- Now we go on another route

$$w(k+1) = w_k + \mathbf{x}(k)$$

$$||w(k+1)||^2 = ||w(k)||^2 + ||\mathbf{x}(k)||^2 + \underbrace{2w(k)^T\mathbf{x}(k)}_{\leq 0 \text{ since x is misclassified}}$$

$$||w(k+1)||^2 \leq ||w(k)||^2 + ||\mathbf{x}(k)||^2$$

$$||w(2)||^2 \leq ||w(1)||^2 + ||\mathbf{x}(1)||^2$$

$$||w(3)||^2 \leq ||w(2)||^2 + ||\mathbf{x}(2)||^2$$

$$\vdots$$

$$\Longrightarrow ||w(k+1)||^2 \leq \sum_{i=1}^{k} ||\mathbf{x}(i)||^2$$

Termination of the Learning Algorithm

- Finally, let $\beta = \max_i ||\mathbf{x}(i)||^2$
- Hence $||w(k+1)||^2 \leq k\beta$
- Remember $||w(k+1)||^2 \geq \frac{(k\alpha)^2}{||w_*||^2}$
- We obtain a contradiction if k is very large.
- Maximum k is when we have equality

$$\frac{(k_{max}\alpha)^2}{||w_*||^2} = k_{max}\beta \Rightarrow k_{max} = \frac{||w_*||^2\beta}{\alpha^2}$$

- Hence the algorithm terminates in maximum $\frac{\beta||w_*||^2}{\alpha^2}$ steps

LMS Algorithm

- Perceptron training algorithm may not converge if points are not linearly separable
- Another approach: Least Mean Squares with gradient descent
- Error function =

$$E(w) = \frac{1}{2}\sum_{i=1}^{N}(d(i) - w^T x(i))^2$$

- Gradient:

$$\nabla E(w) = \sum_{i=1}^{N}(d(i) - w^T x(i))x(i)$$

- Update iterations:

$$w \leftarrow w + \eta \sum_{i=1}^{N}(d(i) - w^T x(i))x(i)$$

Gradient Descent Issues

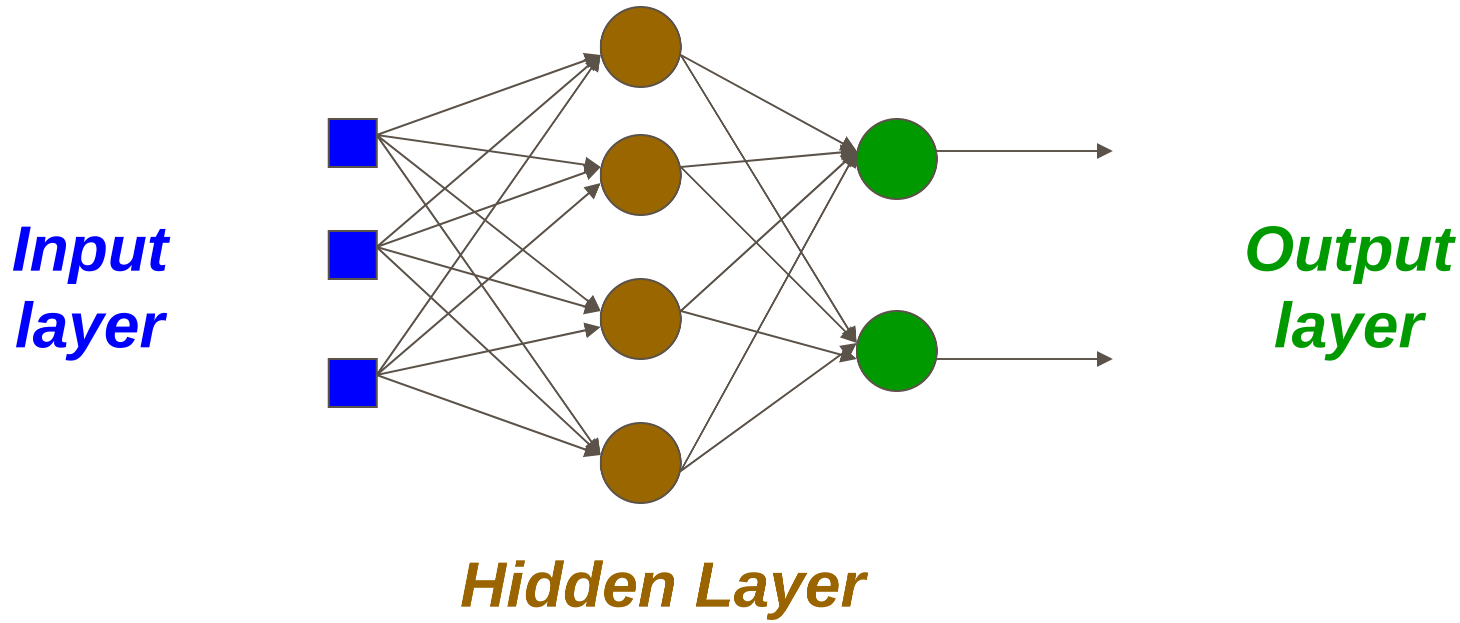
- Converging to the local minimum can be very slow
  - Might need many iterations
- Alternative: Stochastic Gradient Descent
  - Update the weights after each training example rather than all at once

$$w \leftarrow w + \eta x(i)(d(i) - w^T x(i))$$

  - Takes less memory
  - Can sometimes avoid local minima
  - η must decrease with time in order for it to converge

Multi-layer Neural Networks

- Single perceptron can only learn linearly separable functions
  - Also known as ADALINE (Adaptive Linear Neuron)
- Would like to make networks of perceptrons, but how do we determine the contribution to the output for an internal node?
- Solution: Backpropagation Algorithm



XOR

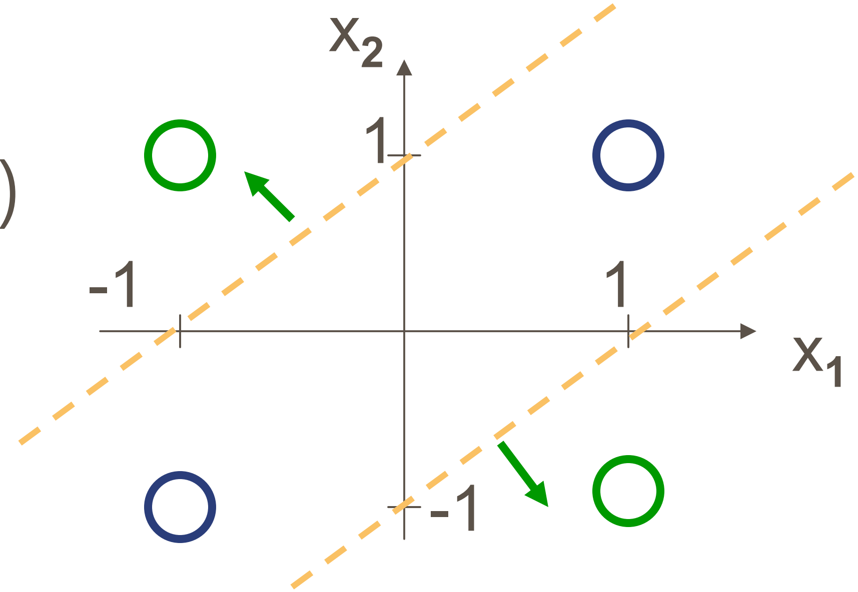
- XOR: non-linearly separable function
  - two input arguments with values in {-1,1}
  - returns one output in {-1,1}

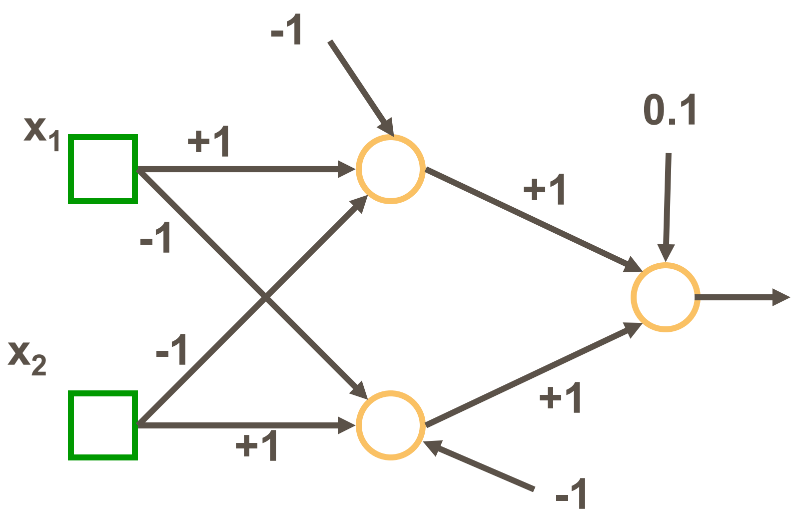
x_1	x_2	x_1 xor x_2
-1	-1	**-1**
-1	1	**1**
1	-1	**1**
1	1	**-1**

- Mimics the logical **exclusive or**:
  - **true** if and only if the two inputs have different values.

XOR Example

- Problem: Separate green and red circles.
- Not linearly separable.
- We have to use two lines (yellow)
- NN with two hidden nodes
  - Each hidden node describes one of the two yellow lines.



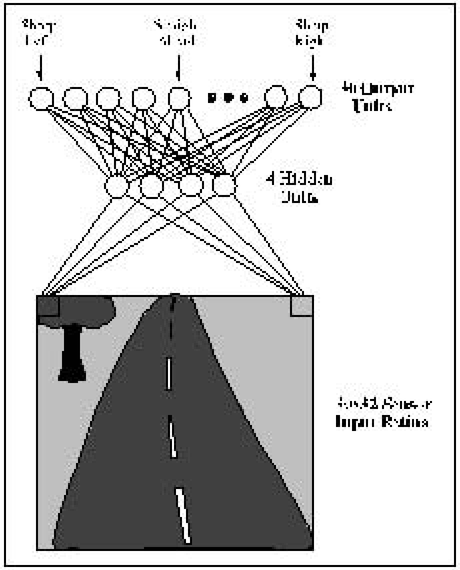


- NN uses the sign activation function.
- Green arrows = directions of the weight vectors of the two hidden nodes, (1,-1) and (-1,1).
- Output node combines the outputs of the two hidden nodes.

ALVINN

Camera image

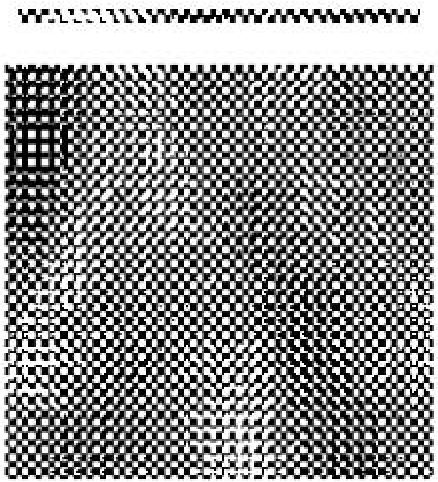
Automated driving at 70 mph on a public highway



30 outputs for steering

4 hidden units

30x32 pixels as inputs



30x32 weights into one out of the four hidden units

NETtalk (Sejnowski & Rosenberg, 1987)

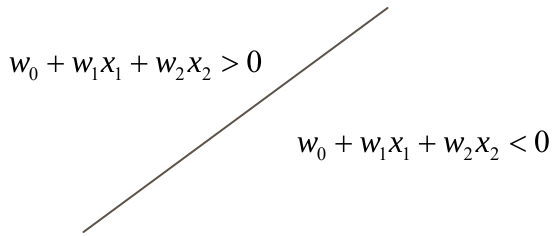
- Task: learn to pronounce English text from examples.
- Training data
  - 1024 words from a side-by-side English/phoneme source
- Input
  - 7 consecutive characters from written text
  - Presented in a moving window that scans text
- Output
  - phoneme code giving the pronunciation of the letter at the center of the input window
- Network topology:
  - 7x29 inputs (26 chars + punctuation marks)
  - 80 hidden units
  - 26 output units (phoneme code).
  - Sigmoid units in hidden and output layer.

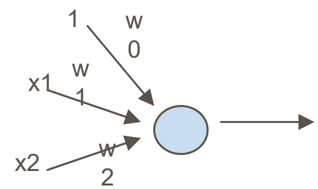
NETtalk (Sejnowski & Rosenberg, 1987)

- Training protocol:
  - 95% accuracy on training set after 50 epochs of training by full gradient descent.
  - 78% accuracy on a set-aside test set.

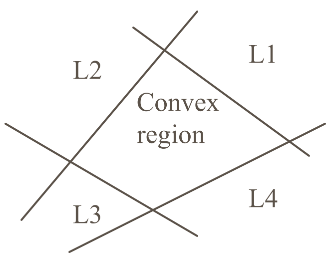
- Comparison against Dectalk (a rule based expert system):
  - Dectalk performs better; it represents a decade of analysis by linguists.
  - NETtalk learns from examples alone and was constructed with little knowledge of the task.

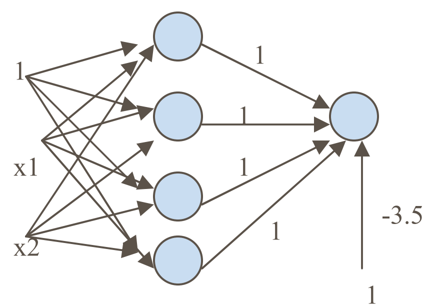
Types of Decision Regions



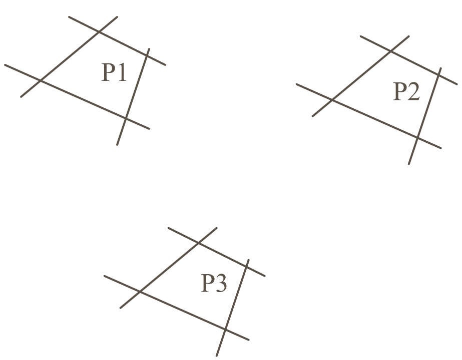


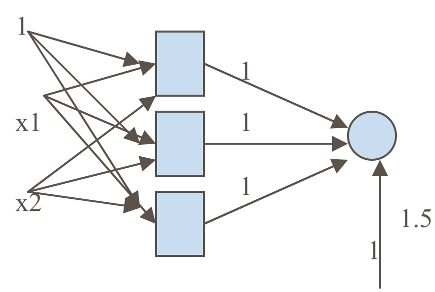
Network with a single node





One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region





two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

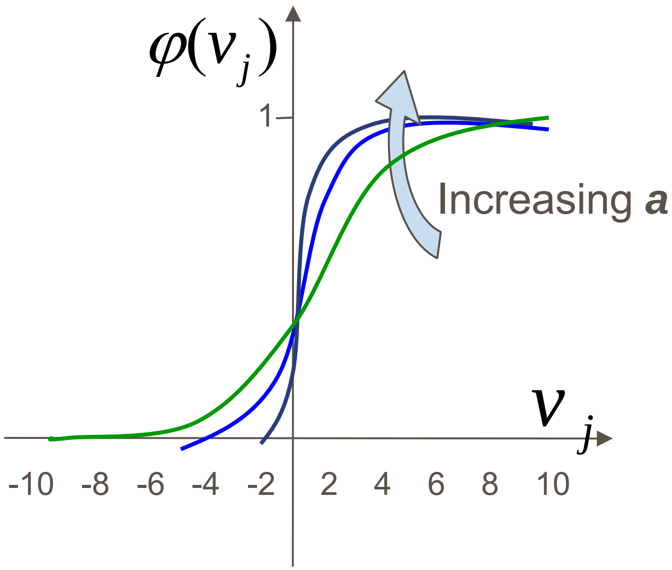
Feed Forward NN NEURON MODEL

- The classical learning algorithm of FFNN is based on the gradient descent method. For this reason the activation function used in FFNN is continuous and differentiable
- A typical activation function that can be viewed as a continuous approximation of the step (threshold) function is the Sigmoid Function. The activation function for node j is:

$$\varphi(v_j) = \frac{1}{1 + e^{-av_j}}, \ a > 0$$

where $v_j = \sum_i w_{ji} y_i$

  - w_{ji} = weight from node i to node j
  - y_i = output of node i



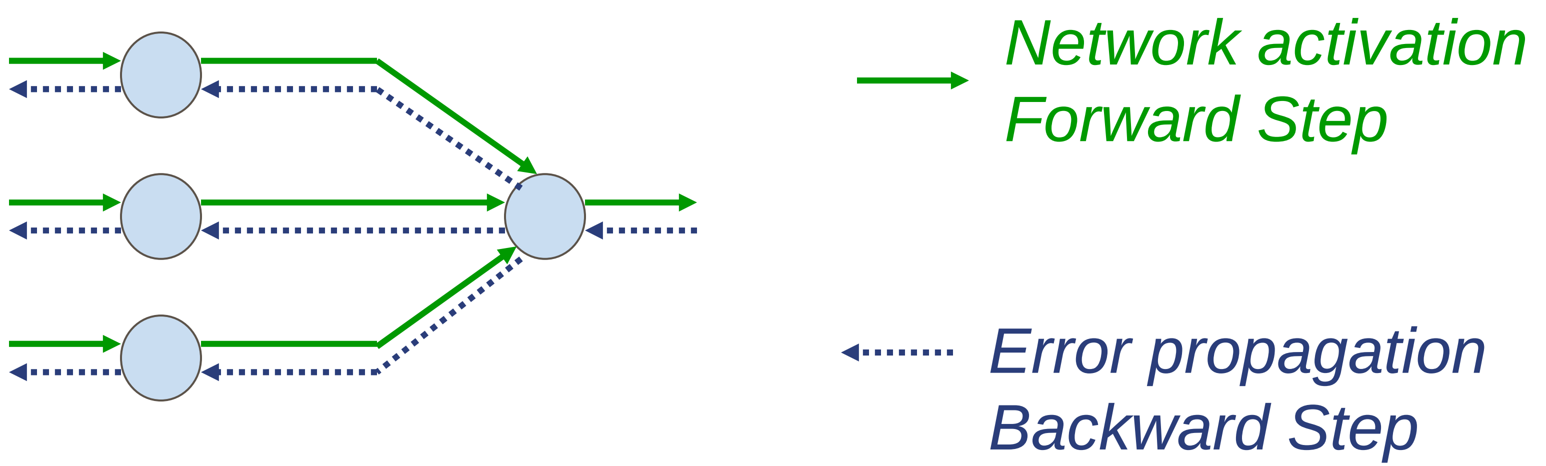
- When $a \longrightarrow \infty$ then φ becomes the step function

The Back-Propagation Algorithm

- Searches for weights that minimize the total error of the network over the training set.
- Repeated application of the following two passes:
  - Forward pass:
    - one example is passed through the network
    - the error of the output layer is computed.
  - Backward pass:
    - update the weights (credit assignment) based on the error
    - more complex than the LMS algorithm for Adaline, because hidden nodes are linked to the error not directly but by means of the nodes of the next layer.
    - Starting at the output layer, the error is propagated backwards through the network, layer by layer.
    - This is done by recursively computing the local gradient of each neuron.

Back-Propagation

- Back-propagation training algorithm



- Backprop adjusts the weights of the NN in order to minimize the network total mean squared error.

Total Mean Squared Error

- The error of output neuron *j* after the activation of the network on the *n*-th training example $(x(n), d(n))$ is:

$$e_j(n) = d_j(n) - y_j(n)$$

- The network error = sum of the squared errors of the output neurons:

$$E(n) = \frac{1}{2} \sum_{j \text{ output node}} e_j^2(n)$$

- *The total mean squared error = average of the network errors over the training examples:*

$$E_{AV} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

Weight Update Rule

- Input of neuron j is: $v_j = \sum_{i=0}^{m} w_{ji} y_i$
- Using the chain rule we can write:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}}$$

- Define the local gradient of neuron j as:

$$\delta_j = -\frac{\partial E}{\partial v_j}$$

- Then from $\frac{\partial v_j}{\partial w_{ji}} = y_i$

we get the change for updating w_{ji}

$$\Delta w_{ji} = \eta y_i \delta_j$$

Weight Update of Output Neuron

- To compute Δw_{ji} we need to know the local gradient δ_j of neuron j .
- Two cases, depending whether j is an output or a hidden neuron.
- If j is an output neuron then using the chain rule we obtain:

$$-\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -e_j(-1)\varphi'(v_j)$$

because $e_j = d_j - y_j$ and $y_j = \varphi(v_j)$

- So if j is an output node then the weight w_{ji} from neuron i to neuron j is updated by:

$$\Delta w_{ji} = \eta y_i(d_j - y_j)\varphi'(v_j)$$

Weight Update of Hidden Neurons

- If j is a hidden neuron then its local gradient δ_j is computed using the local gradients of all the neurons of the next layer.
- Using the chain rule we have: $$\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$
- And

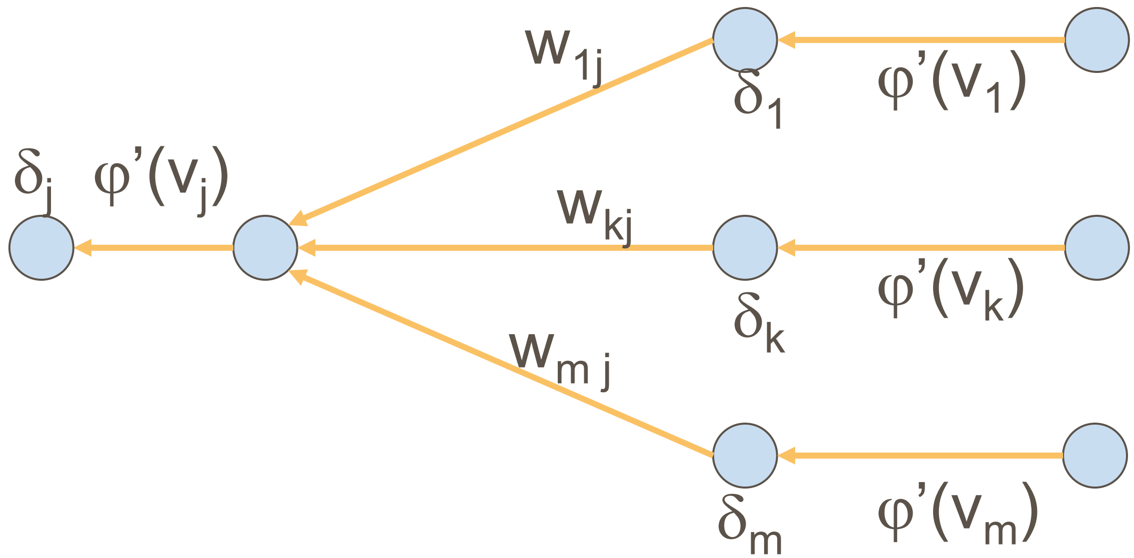
$$-\frac{\partial E}{\partial y_j} = \sum_{k \in C} \left(-\frac{\partial E}{\partial v_k}\right)\frac{\partial v_k}{\partial y_j} = \sum_{k \in C} \delta_k w_{kj}$$

- Hence $-\frac{\partial E}{\partial y_j} = \sum_{k \text{ in next layer}} \delta_k w_{kj}$ and $\frac{\partial y_j}{\partial v_j} = \varphi'(v_j)$
- So if j is a hidden node then the weight w_{ji} from neuron i to neuron j is updated by:

$$\Delta w_{ji} = \eta y_i \delta_j = \eta y_i \varphi'(v_j) \sum_{k \text{ in next layer}} \delta_k w_{kj}$$

Error Backpropagation

The flow-graph below illustrates how errors are back-propagated to hidden neuron *j*



$$\delta_j = \varphi'(v_j) \sum_{k \text{ in next layer}} \delta_k w_{kj}$$

Summary: Weight Update Rule

- Update weights by $\Delta w_{ji} = \eta y_i \delta_j$

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{if } j \text{ output node} \\ \varphi'(v_j) \displaystyle\sum_{k \text{ in next layer}} \delta_k w_{kj} & \text{if } j \text{ hidden node} \end{cases}$$

where $\varphi'(v_j) = a y_j (1 - y_j)$

Momentum

Learning difficulties:

- If η is small → slow learning
- If η is large → unstable behavior with oscillations of the weight values.

Momentum

- A technique that keeps most of the previous update.
- We obtain the following generalized Update rule:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

α momentum constant $0 \leq \alpha < 1$

- Accelerates the descent in steady downhill directions.
- Has a stabilizing effect in directions that oscillate in time.

Other techniques: η adaptation

Other heuristics for accelerating the convergence of the back-propagation algorithm through η adaptation:

- Heuristic 1: Every weight has its own η.
- Heuristic 2: Every η is allowed to vary from one iteration to the next.

Back-Propagation Learning algorithm (incremental-mode)

n=1;

initialize w(n) randomly;

while (stopping criterion not satisfied and n<max_iterations)

for each example (x,*d*)

- run the network with input x and compute the output y

- update the weights in backward order starting from those of the output layer:

$$w_{ji}(n) = w_{ji}(n-1) + \Delta w_{ji}(n-1)$$

with $\Delta w_{ji}(n)$ computed using the (generalized) update rule

end-for

n = n+1;

end-while;

Back-Propagation Algorithm

- In the batch-mode the weights are updated only after all examples have been processed, using the formula

$$w_{ji} = w_{ji} + \sum_{\text{example } x} \Delta w_{ji}^{x}$$

- The learning process continues on an epoch-by-epoch basis until the stopping condition is satisfied.
- In the incremental mode from one epoch to the next choose a randomized ordering for selecting the examples in the training set in order to avoid poor performance.

Initialization of Weights

- In general, initial weights are randomly chosen, with typical values between -1.0 and 1.0 or -0.5 and 0.5.
- If some inputs are much larger than others, random initialization may bias the network to give much more importance to larger inputs. In such a case, weights can be initialized as follows:

$$\mathrm{w}_{ji} = \pm \frac{1}{2N} \sum_{i=1,\ldots,N} \frac{1}{|\mathrm{x}_i|}$$

For weights from the input to the first layer

$$\mathrm{w}_{kj} = \pm \frac{1}{2N} \sum_{i=1,\ldots,N} \frac{1}{\varphi(\sum \mathrm{w}_{ji}\mathrm{x}_i)}$$

For weights from the first to the second layer

Training

- Rule of thumb:
  - the number of training examples should be at least five to ten times the number of weights of the network.

- Other rule of thumb:

$$N > \frac{|W|}{1-a}$$

$|W|$= number of weights

a = expected accuracy on test set

When to Stop Learning

- Learn until error on the training set is below some threshold
  - Bad idea! Can result in overfitting
  - If you match the training examples too well, your performance on the real problems may suffer
  - Early stopping

- Learn trying to get the best result on a validation set
  - Stop when the performance seems to be decreasing on this, while saving the best network seen so far.
  - There may be local minima, so watch out!
  - Keep many solutions during the optimization

Representational Capabilities

- Boolean functions – Every boolean function can be represented exactly by some network with one hidden layer
  - Size may be exponential on the number of inputs
- Continuous functions – Can be approximated to arbitrary accuracy with one hidden layer
- Arbitrary functions – Any function can be approximated to arbitrary accuracy with two hidden layers

Applications of FFNN

Classification, pattern recognition:

- FFNN can be applied to tackle non-linearly separable learning problems.
  - Recognizing printed or handwritten characters,
  - Face recognition
  - Classification of loan applications into credit-worthy and non-credit-worthy groups
  - Analysis of sonar radar to determine the nature of the source of a signal

Regression and forecasting:

- FFNN can be applied to learn non-linear functions (regression) and in particular functions whose inputs is a sequence of measurements over time (time series).

Conclusions

- Neural Networks
  - Feed-forward NN → Back-Propagation Algorithm
  - Recurrent NN
  - Self Organizing Maps (Unsupervised Learning)
- Pros
  - Fast
  - Can learn any function with any degree of accuracy
- Cons
  - Hard to train
    - Steepest descent can get stuck in local optima
    - A randomized version (stochastic gradient descent) could help
  - Must be careful to avoid overfitting
    - Use a validation set