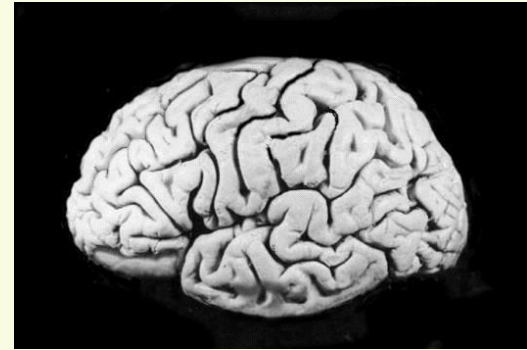


Multilayer Neural Networks

Brain vs. Computer



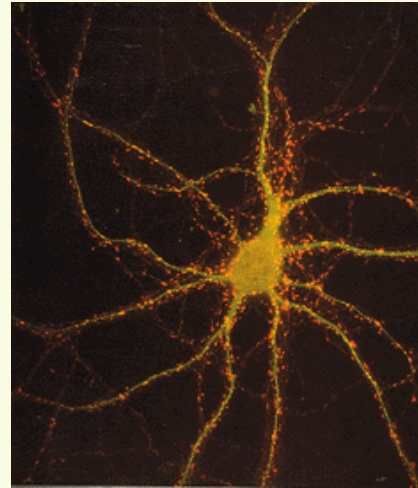
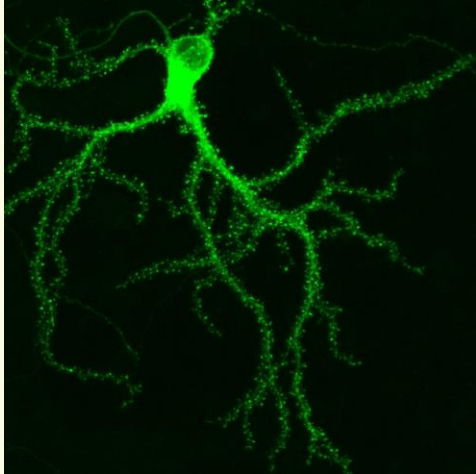
- Designed to solve logic and arithmetic problems
- Can solve a gazillion arithmetic and logic problems in an hour
- absolute precision
- Usually one very fast procesor
- high reliability



- Evolved (in a large part) for pattern recognition
- Can solve a gazillion of PR problems in an hour
- Huge number of parallel but relatively slow and unreliable processors
- not perfectly precise
- not perfectly reliable

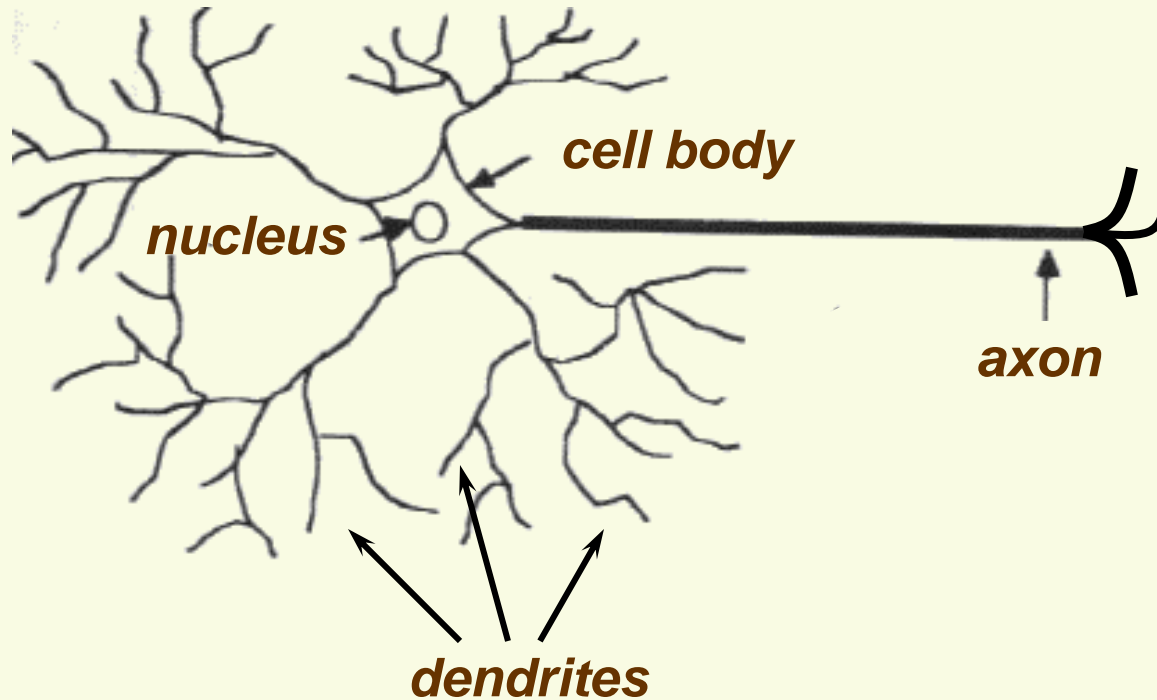
Seek an inspiration from human brain for PR?

Neuron: Basic Brain Processor



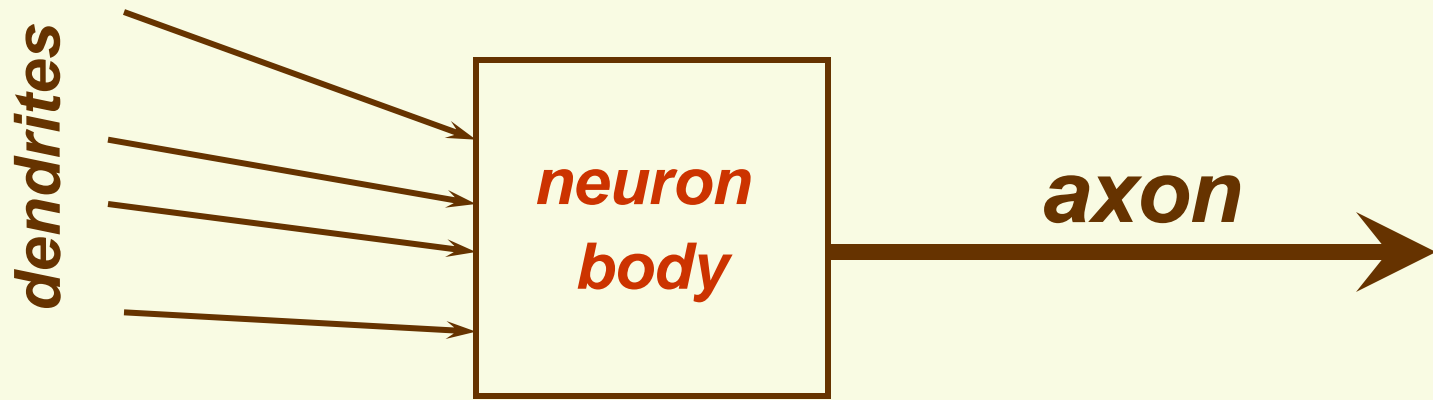
- Neurons are nerve cells that transmit signals to and from brains at the speed of around 200mph
- Each neuron cell communicates to anywhere from 1000 to 10,000 other neurons, muscle cells, glands, so on
- Have around 10^{10} neurons in our brain (network of neurons)
- Most neurons a person is ever going to have are already present at birth

Neuron: Basic Brain Processor



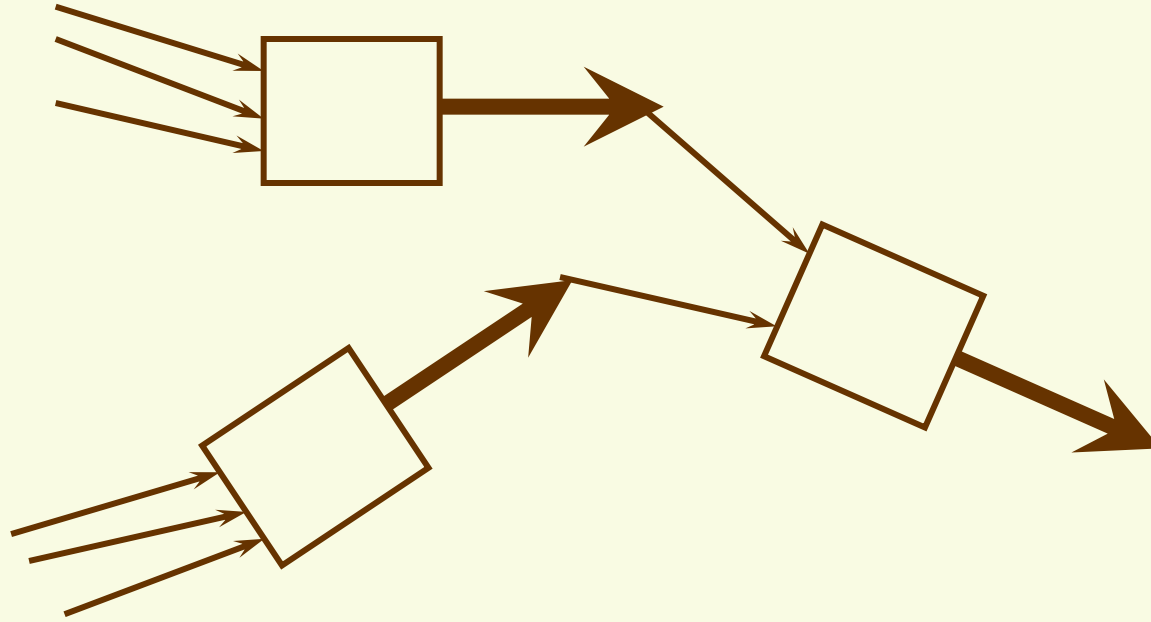
- Main components of a neuron
 - **Cell body** which holds DNA information in **nucleus**
 - **Dendrites** may have thousands of dendrites, usually short
 - **axon** long structure, which splits in possibly thousands branches at the end. May be up to 1 meter long

Neuron in Action (simplified)



- ***Input***: neuron collects signals from other neurons through dendrites, may have thousands of dendrites
- ***Processor***: Signals are accumulated and processed by the cell body
- ***Output***: If the strength of incoming signals is large enough, the cell body sends a signal (a spike of electrical activity) to the axon

Neural Network



ANN History: Birth

- 1943, famous paper by W. McCulloch (neurophysiologist) and W. Pitts (mathematician)
 - Using only math and algorithms, constructed a model of how neural network may work
 - Showed it is possible to construct any computable function with their network
 - Was it possible to make a model of thoughts of a human being?
 - Considered to be the birth of AI
- 1949, D. Hebb, introduced the first (purely psychological) theory of learning
 - Brain learns at tasks through life, thereby it goes through tremendous changes
 - If two neurons fire together, they strengthen each other's responses and are likely to fire together in the future

ANN History: First Successes

- 1958, F. Rosenblatt,
 - perceptron, oldest neural network still in use today
 - Algorithm to train the perceptron network (training is still the most actively researched area today)
 - Built in hardware
 - Proved convergence in linearly separable case
- 1959, B. Widrow and M. Hoff
 - Madaline
 - First ANN applied to real problem (eliminate echoes in phone lines)
 - Still in commercial use

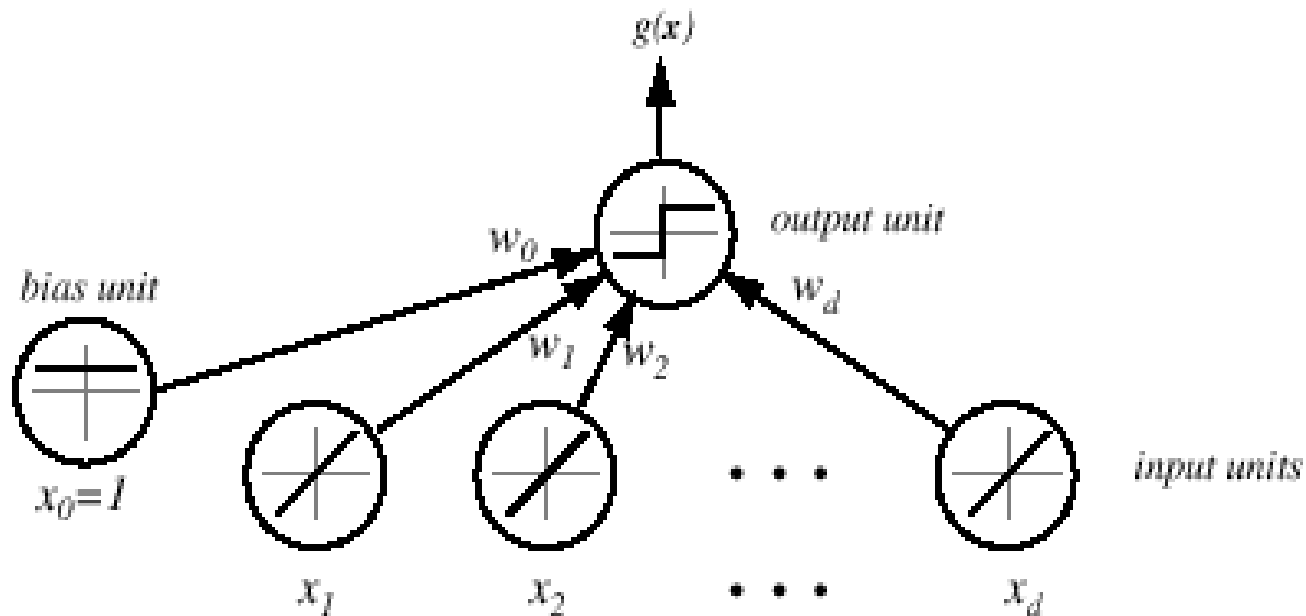
ANN History: Stagnation

- Early success lead to a lot of claims which were not fulfilled
- 1969, M. Minsky and S. Pappert
 - Book “Perceptrons”
 - Proved that perceptrons can learn only linearly separable classes
 - In particular cannot learn very simple XOR function
 - Conjectured that multilayer neural networks also limited by linearly separable functions
- No funding and almost no research (at least in North America) in 1970’s as the result of 2 things above

ANN History: Revival

- Revival of ANN in 1980's
- 1982, J. Hopfield
 - New kind of networks (Hopfield's networks)
 - Bidirectional connections between neurons
 - Implements associative memory
- 1982 joint US-Japanese conference on ANN
 - US worries that it will stay behind
- Many examples of multilayer NN appear
- 1982, discovery of backpropagation algorithm
 - Allows a network to learn not linearly separable classes
 - Discovered independently by
 1. Y. Lecunn
 2. D. Parker
 3. Rumelhart, Hinton, Williams

ANN: Perceptron



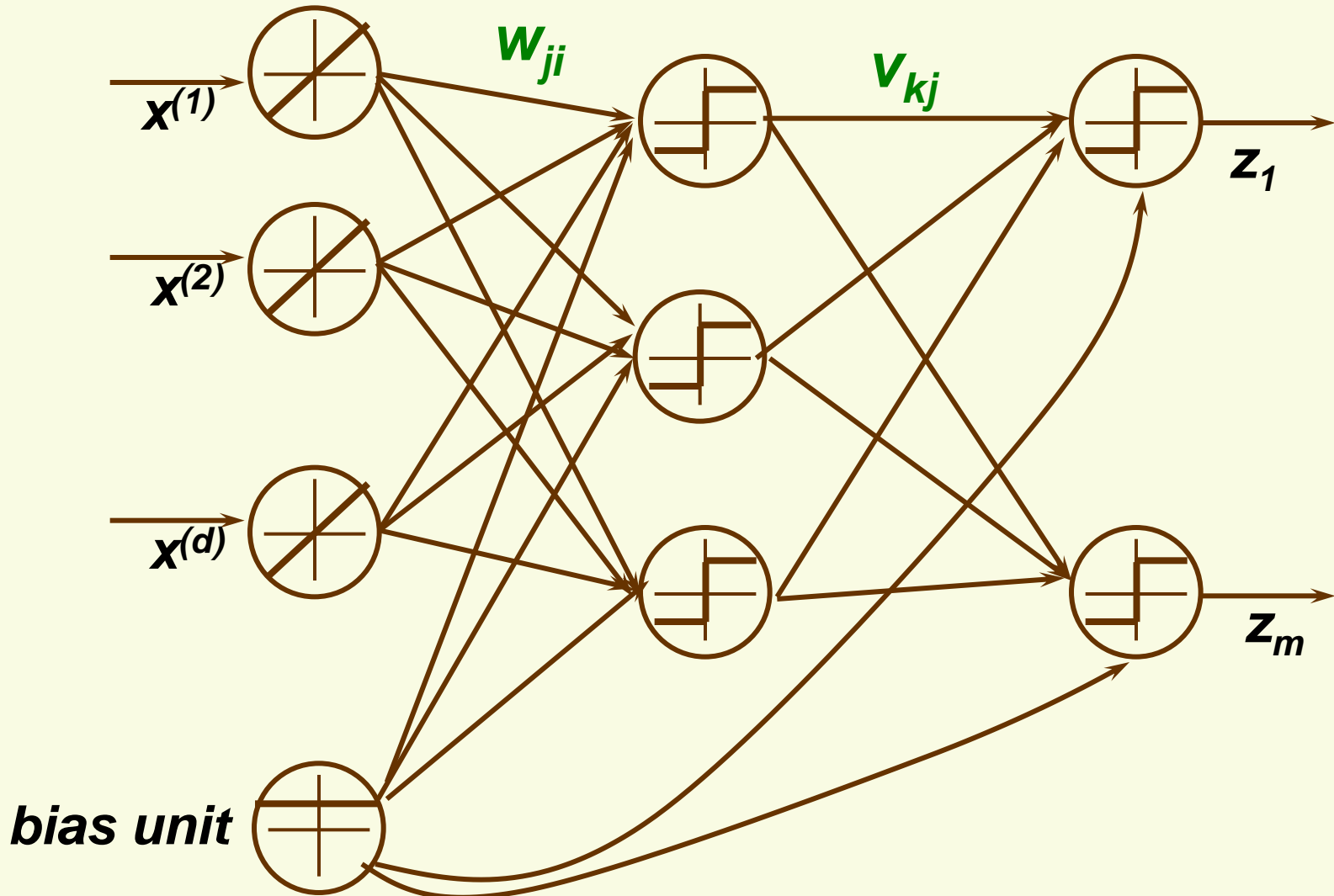
- Input and output layers
- $\mathbf{g}(\mathbf{x}) = \mathbf{w}^t \mathbf{x} + w_0$
- Limitation: can learn only linearly separable classes

MNN: Feed Forward Operation

input layer:
 d features

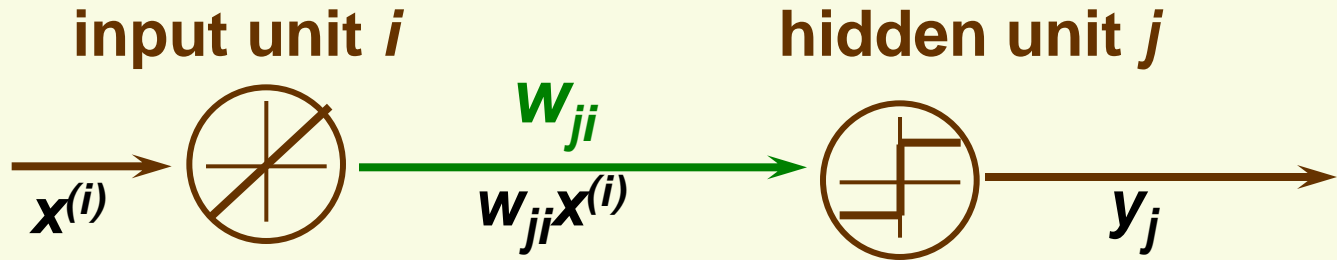
hidden layer:

output layer:
 m outputs, one for each class

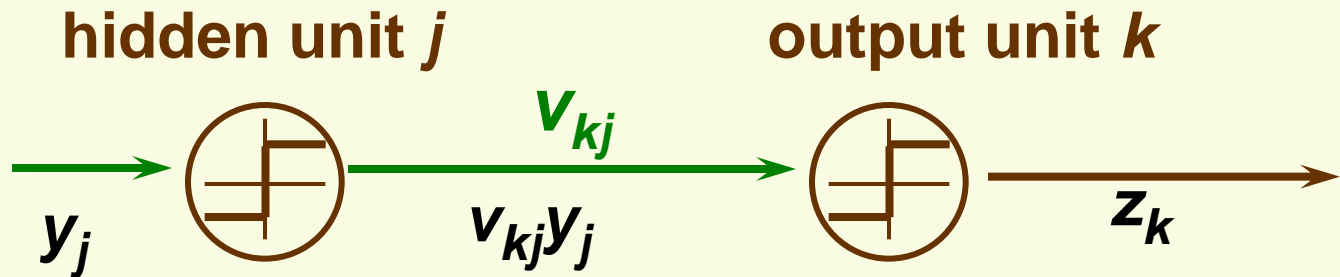


MNN: Notation for Weights

- Use w_{ji} to denote the weight between input unit i and hidden unit j



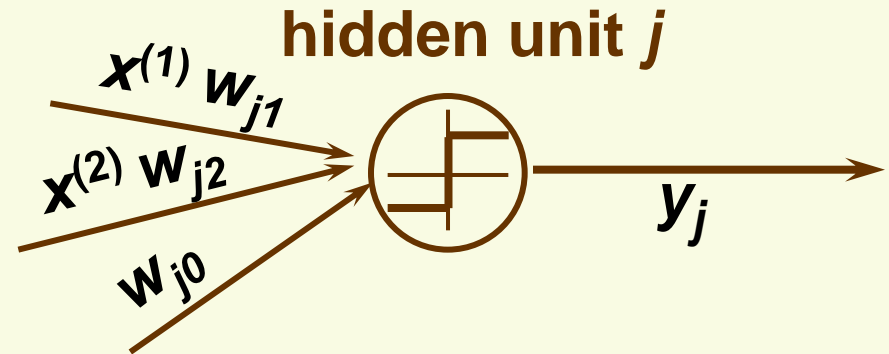
- Use v_{kj} to denote the weight between hidden unit j and output unit k



MNN: Notation for Activation

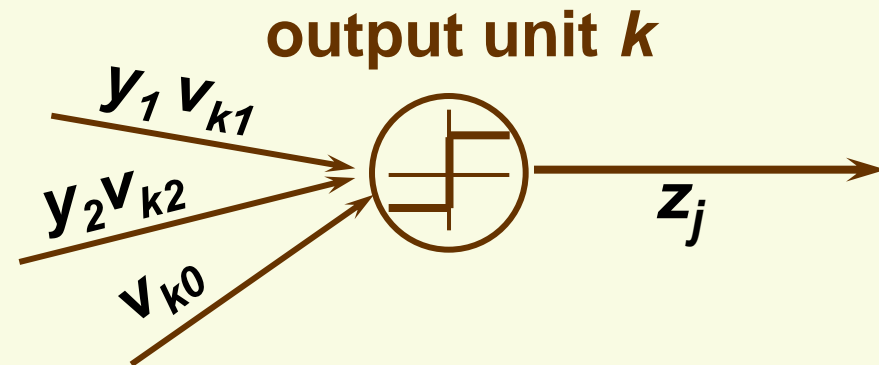
- Use net_j to denote the activation and hidden unit j

$$net_j = \sum_{i=1}^d x^{(i)} w_{ji} + w_{j0}$$



- Use net_k^* to denote the activation at output unit k

$$net_k^* = \sum_{j=1}^{N_H} y_j v_{kj} + v_{k0}$$



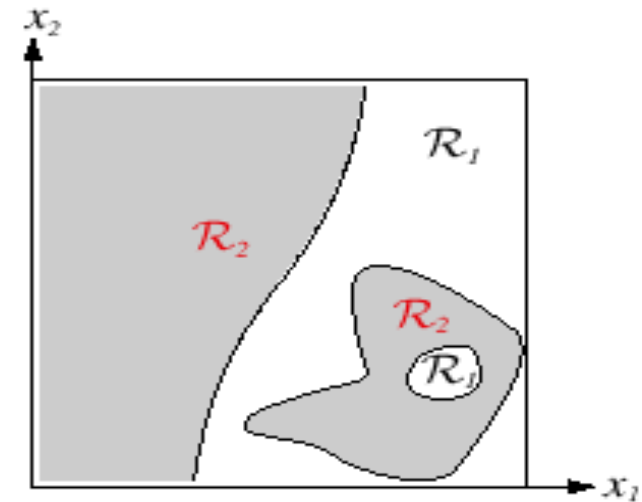
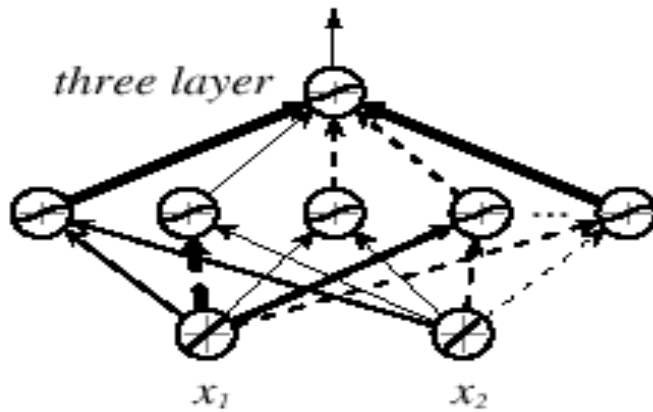
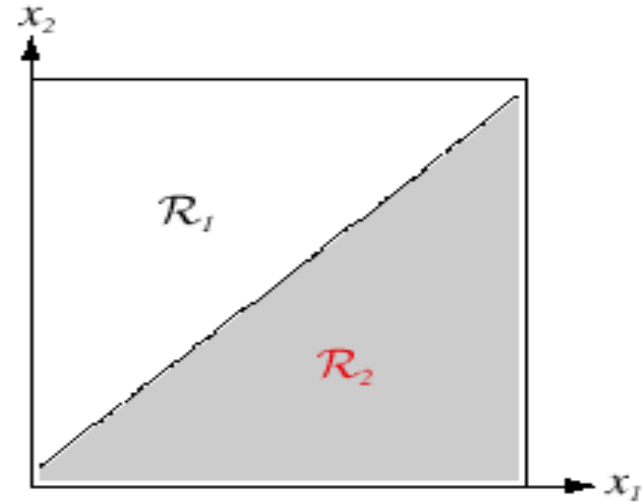
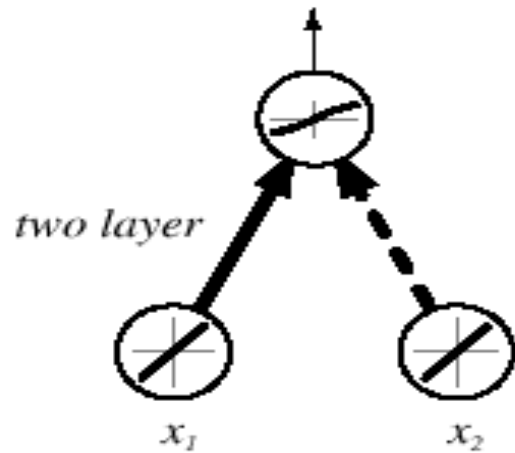
Discriminant Function

- Discriminant function for class k (the output of the k th output unit)

$$\begin{aligned} g_k(\mathbf{x}) = \mathbf{z}_k = & \\ = f \left(\underbrace{\sum_{j=1}^{N_H} v_{kj} f \left(\sum_{i=1}^d w_{ji} x^{(i)} + w_{j0} \right)}_{\text{activation at } k\text{th output unit}} + v_{k0} \right) \end{aligned}$$

activation at j th hidden unit

Discriminant Function



Expressive Power of MNN

- It can be shown that every **continuous** function from input to output can be implemented with enough hidden units, 1 hidden layer, and proper nonlinear activation functions
- This is more of theoretical than practical interest
 - The proof is not constructive (does not tell us exactly how to construct the MNN)
 - Even if it were constructive, would be of no use since we do not know the desired function anyway, our goal is to learn it through the samples
 - But this result does give us confidence that we are on the right track
 - MNN is general enough to construct the correct decision boundaries, unlike the Perceptron

MNN Activation function

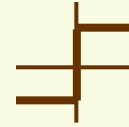
- Must be nonlinear for expressive power larger than that of perceptron
 - If use linear activation function at hidden layer, can only deal with linearly separable classes
 - Suppose at hidden unit j , $h(\mathbf{u}) = \mathbf{a}_j \mathbf{u}$

$$\begin{aligned} \mathbf{g}_k(\mathbf{x}) &= \mathbf{f} \left(\sum_{j=1}^{N_H} \mathbf{v}_{kj} h \left(\sum_{i=1}^d \mathbf{w}_{ji} \mathbf{x}^{(i)} + \mathbf{w}_{j0} \right) + \mathbf{v}_{k0} \right) \\ &= \mathbf{f} \left(\sum_{j=1}^{N_H} \mathbf{v}_{kj} \mathbf{a}_j \left(\sum_{i=1}^d \mathbf{w}_{ji} \mathbf{x}^{(i)} + \mathbf{w}_{j0} \right) + \mathbf{v}_{k0} \right) \\ &= \mathbf{f} \left(\sum_{i=1}^d \sum_{j=1}^{N_H} \left(\mathbf{v}_{kj} \mathbf{a}_j \mathbf{w}_{ji} \mathbf{x}^{(i)} + \mathbf{v}_{kj} \mathbf{a}_j \mathbf{w}_{j0} \right) + \mathbf{v}_{k0} \right) \\ &= \mathbf{f} \left(\sum_{i=1}^d \mathbf{x}^{(i)} \sum_{j=1}^{N_H} \mathbf{v}_{kj} \mathbf{a}_j \mathbf{w}_{ji}^{new} + \left(\sum_{j=1}^{N_H} \mathbf{v}_{kj} \mathbf{a}_j \mathbf{w}_{j0}^{new} + \mathbf{v}_{k0} \right) \right) \end{aligned}$$

MNN Activation function

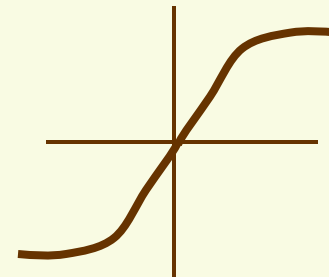
- could use a discontinuous activation function

$$f(\text{net}_k) = \begin{cases} 1 & \text{if } \text{net}_k \geq 0 \\ -1 & \text{if } \text{net}_k < 0 \end{cases}$$



- However, we will use gradient descent for learning, so we need to use a continuous activation function

sigmoid function



- From now on, assume f is a differentiable function

MNN: Modes of Operation

- Network have two modes of operation:
 - ***Feedforward***

The feedforward operations consists of presenting a pattern to the input units and passing (or feeding) the signals through the network in order to get outputs units (no cycles!)
 - ***Learning***

The supervised learning consists of presenting an input pattern and modifying the network parameters (weights) to reduce distances between the computed output and the desired output

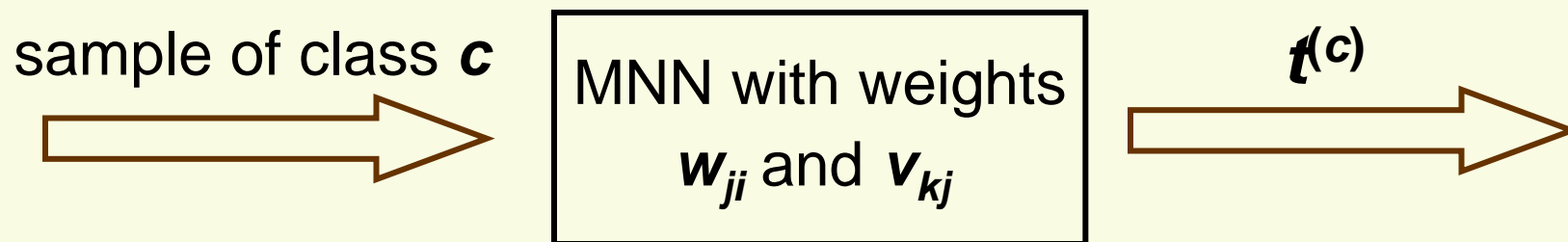
- Can vary
 - number of hidden layers
 - Nonlinear activation function
 - Can use different function for hidden and output layers
 - Can use different function at each hidden and output node

MNN: Class Representation

- Training samples $\mathbf{x}_1, \dots, \mathbf{x}_n$ each of class $1, \dots, m$
- Let network output \mathbf{z} represent class \mathbf{c} as **target** $\mathbf{t}^{(c)}$

$$\mathbf{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_c \\ \vdots \\ z_m \end{bmatrix} = \mathbf{t}^{(c)} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{c-th row}$$

Our Ultimate Goal For FeedForward Operation

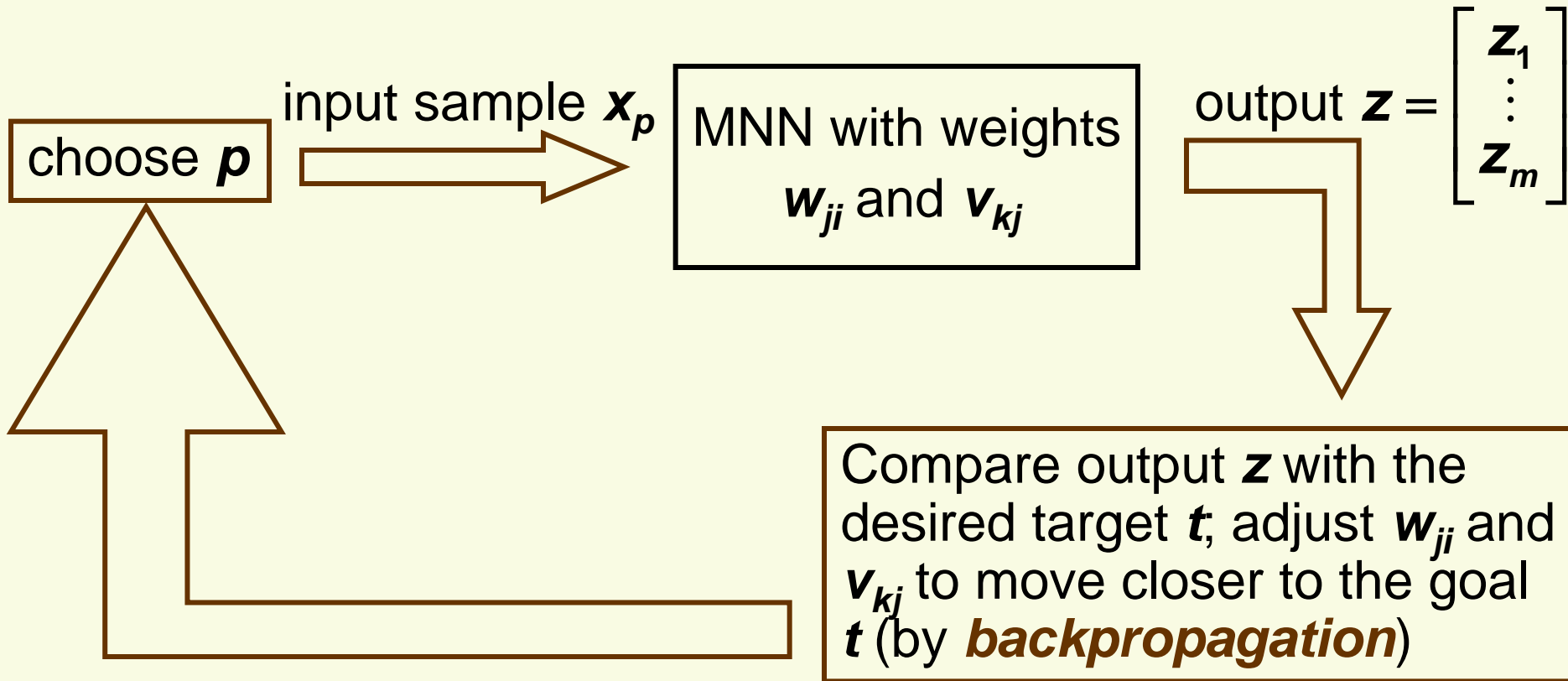


MNN training to achieve the Ultimate Goal

Modify (learn) MNN parameters \mathbf{w}_{ji} and \mathbf{v}_{kj} so that for each **training** sample of class \mathbf{c} MNN output $\mathbf{z} = \mathbf{t}^{(c)}$

Network Training (learning)

1. Initialize weights w_{ji} and v_{kj} randomly **but not to 0**
2. Iterate until a stopping criterion is reached



BackPropagation

- Learn \mathbf{w}_{ji} and \mathbf{v}_{kj} by minimizing the training error
- What is the training error?
- Suppose the output of MNN for sample \mathbf{x} is \mathbf{z} and the target (desired output for \mathbf{x}) is \mathbf{t}
- Error on one sample:
$$\mathcal{J}(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{c=1}^m (t_c - z_c)^2$$
- Training error:
$$\mathcal{J}(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{i=1}^n \sum_{c=1}^m (t_c^{(i)} - z_c^{(i)})^2$$

- Use gradient descent:

$\mathbf{v}^{(0)}, \mathbf{w}^{(0)} = \text{random}$

repeat until convergence:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}^{(t)})$$

$$\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} - \eta \nabla_{\mathbf{v}} \mathcal{J}(\mathbf{v}^{(t)})$$

BackPropagation

- For simplicity, first take training error for one sample \mathbf{x}_i

$$J(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{c=1}^m (t_c - \mathbf{z}_c)^2$$

function of w,v

fixed constant

$$\mathbf{z}_k = f \left(\sum_{j=1}^{N_H} \mathbf{v}_{kj} f \left(\sum_{i=1}^d \mathbf{w}_{ji} \mathbf{x}^{(i)} + \mathbf{w}_{j0} \right) + \mathbf{v}_{k0} \right)$$

- Need to compute

- partial derivative w.r.t. hidden-to-output weights $\frac{\partial J}{\partial \mathbf{v}_{kj}}$
- partial derivative w.r.t. input-to-hidden weights $\frac{\partial J}{\partial \mathbf{w}_{ji}}$

BackPropagation: Layered Model

activation at
hidden unit j

$$net_j = \sum_{i=1}^d x^{(i)} w_{ji} + w_{j0}$$



output at
hidden unit j

$$y_j = f(net_j)$$



activation at
output unit k

$$net_k^* = \sum_{j=1}^{N_H} y_j v_{kj} + v_{k0}$$



activation at
output unit k

$$z_k = f(net_k^*)$$



objective function

$$J(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{c=1}^m (t_c - z_c)^2$$

chain rule

$$\frac{\partial J}{\partial v_{kj}}$$

chain rule

$$\frac{\partial J}{\partial w_{ji}}$$

BackPropagation

$$net_k^* = \sum_{j=1}^{N_H} y_j v_{kj} + v_{k0} \Rightarrow z_k = f(net_k^*) \Rightarrow J(w, v) = \frac{1}{2} \sum_{c=1}^m (t_c - z_c)^2$$

- First compute hidden-to-output derivatives $\frac{\partial J}{\partial v_{kj}}$

$$\frac{\partial J}{\partial v_{kj}} = \frac{1}{2} \sum_{c=1}^m \frac{\partial}{\partial v_{kj}} (t_c - z_c)^2 = \sum_{c=1}^m (t_c - z_c) \frac{\partial}{\partial v_{kj}} (t_c - z_c)$$

$$= (t_k - z_k) \frac{\partial}{\partial v_{kj}} (t_k - z_k) = -(t_k - z_k) \frac{\partial}{\partial v_{kj}} (z_k)$$

$$= -(t_k - z_k) \frac{\partial z_k}{\partial net_k^*} \frac{\partial net_k^*}{\partial v_{kj}}$$

$$= \begin{cases} -(t_k - z_k) f'(net_k^*) y_j & \text{if } j \neq 0 \\ -(t_k - z_k) f'(net_k^*) & \text{if } j = 0 \end{cases}$$

BackPropagation

Gradient Descent **Single Sample** Update Rule for hidden-to-output weights v_{kj}

$$j > 0: \mathbf{v}_{kj}^{(t+1)} = \mathbf{v}_{kj}^{(t)} + \eta(t_k - \mathbf{z}_k) f'(net_k^*) y_j$$

$$j = 0 \text{ (bias weight): } \mathbf{v}_{k0}^{(t+1)} = \mathbf{v}_{k0}^{(t)} + \eta(t_k - \mathbf{z}_k) f'(net_k^*)$$

BackPropagation

- Now compute input-to-hidden $\frac{\partial J}{\partial w_{ji}}$

$$\begin{aligned}\frac{\partial J}{\partial w_{ji}} &= \sum_{k=1}^m (t_k - z_k) \frac{\partial}{\partial w_{ji}} (t_k - z_k) \\ &= - \sum_{k=1}^m (t_k - z_k) \frac{\partial z_k}{\partial w_{ji}} = - \sum_{k=1}^m (t_k - z_k) \frac{\partial z_k}{\partial net_k^*} \frac{\partial net_k^*}{\partial w_{ji}} \\ &= - \sum_{k=1}^m (t_k - z_k) f'(net_k^*) \frac{\partial net_k^*}{\partial y_j} \frac{\partial y_j}{\partial w_{ji}} \\ &= - \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \begin{cases} - \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj} f'(net_j) x^{(i)} & \text{if } i \neq 0 \\ - \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj} f'(net_j) & \text{if } i = 0 \end{cases}\end{aligned}$$

$net_h = \sum_{h=1}^d x^{(i)} w_{hi} + w_{h0}$

$y_j = f(net_j)$

$net_k^* = \sum_{s=1}^{N_H} y_s v_{ks} + v_{k0}$

$z_k = f(net_k^*)$

$J(w, v) = \frac{1}{2} \sum_{c=1}^m (t_c - z_c)^2$

BackPropagation

$$\frac{\partial J}{\partial w_{ji}} = \begin{cases} -f'(net_j) x^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj} & \text{if } i \neq 0 \\ -f'(net_j) \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj} & \text{if } i = 0 \end{cases}$$

Gradient Descent **Single Sample** Update Rule for input-to-hidden weights w_{ji}

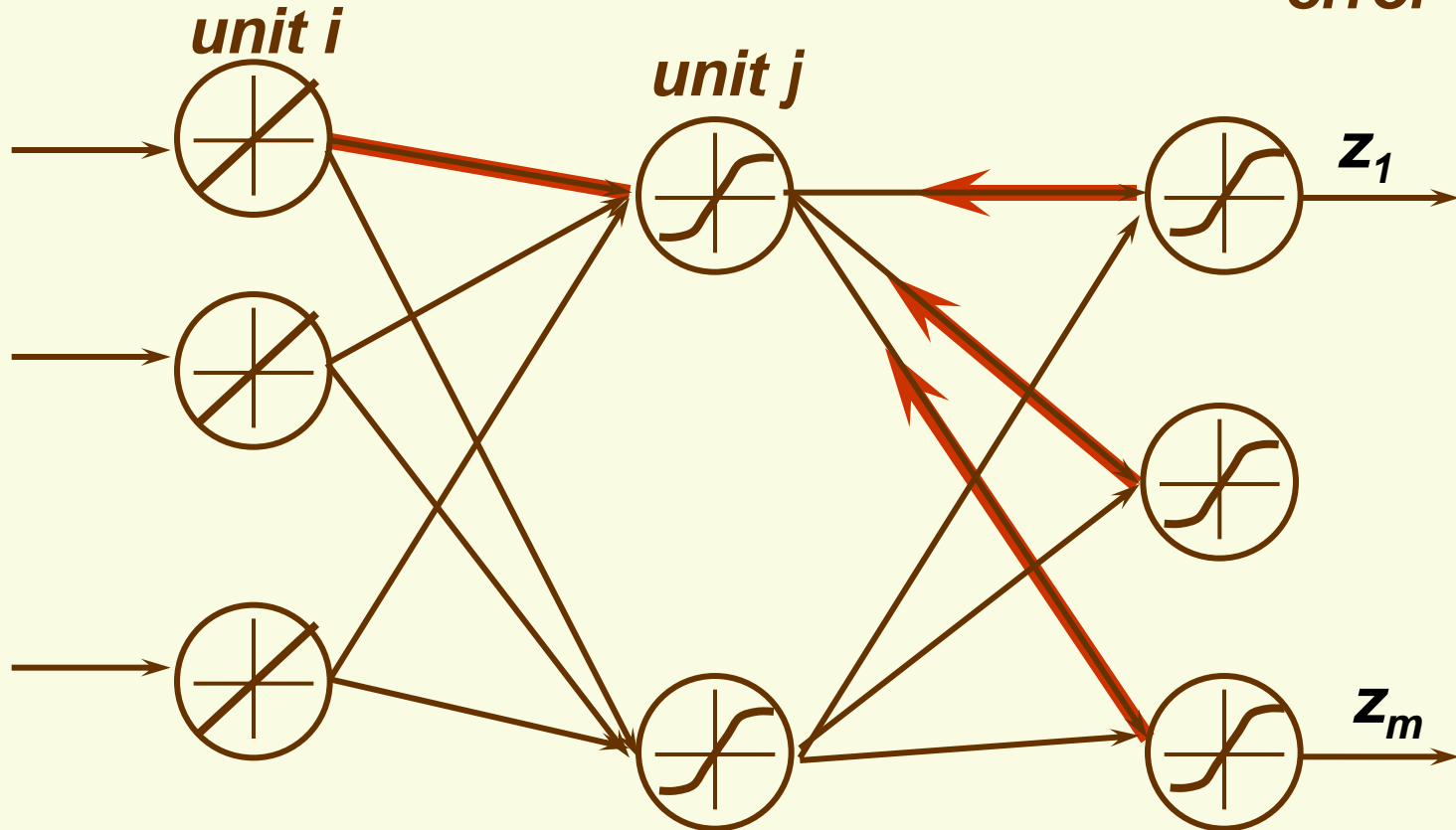
$$i > 0: \quad w_{ji}^{(t+1)} = w_{ji}^{(t)} + \eta f'(net_j) x^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}^{(t)}$$

$$i = 0 \text{ (bias weight): } w_{j0}^{(t+1)} = w_{j0}^{(t)} + \eta f'(net_j) \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}^{(t)}$$

BackPropagation of Errors

$$\frac{\partial J}{\partial w_{ji}} = -f'(net_j) x^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

$$\frac{\partial J}{\partial v_{kj}} = - \underbrace{(t_k - z_k)}_{\text{error}} f'(net_k^*) y_j$$



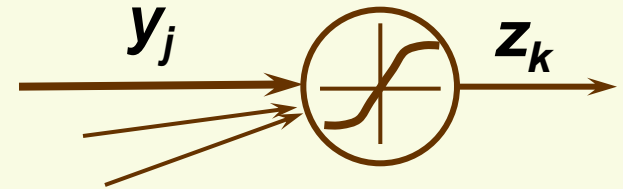
- Name “backpropagation” because during training, errors propagated back from output to hidden layer

BackPropagation

- Consider update rule for hidden-to-output weights:

$$\mathbf{v}_{kj}^{(t+1)} = \mathbf{v}_{kj}^{(t)} + \eta(\mathbf{t}_k - \mathbf{z}_k) f'(\mathbf{net}_k^*) \mathbf{y}_j$$

- Suppose $\mathbf{t}_k - \mathbf{z}_k > 0$
- Then output of the k th hidden unit is too small: $\mathbf{t}_k > \mathbf{z}_k$
- Typically activation function f is s.t. $f' > 0$
- Thus $(\mathbf{t}_k - \mathbf{z}_k) f'(\mathbf{net}_k^*) > 0$
- There are 2 cases:



- $\mathbf{y}_j > 0$, then to increase \mathbf{z}_k , should increase weight \mathbf{v}_{kj} which is exactly what we do since $\eta(\mathbf{t}_k - \mathbf{z}_k) f'(\mathbf{net}_k^*) \mathbf{y}_j > 0$
- $\mathbf{y}_j < 0$, then to increase \mathbf{z}_k , should decrease weight \mathbf{v}_{kj} which is exactly what we do since $\eta(\mathbf{t}_k - \mathbf{z}_k) f'(\mathbf{net}_k^*) \mathbf{y}_j < 0$

BackPropagation

- The case $t_k - z_k < 0$ is analogous
- Similarly, can show that input-to-hidden weights make sense
- Important: weights should be initialized to random **nonzero** numbers

$$\frac{\partial J}{\partial w_{ji}} = -f'(net_j) x^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

- if $v_{kj} = 0$, input-to-hidden weights w_{ji} never updated

Training Protocols

- How to present samples in training set and update the weights?
- Three major training protocols:
 1. Stochastic
 - Patterns are chosen randomly from the training set, and network weights are updated after every sample presentation
 2. Batch
 - weights are update based on all samples; iterate weight update
 3. Online
 - each sample is presented only once, weight update after each sample presentation

Stochastic Back Propagation

1. Initialize

- number of hidden layers n_H
- weights w, v
- convergence criterion θ and learning rate η
- time $t = 0$

2. do

$x \leftarrow$ randomly chosen training pattern

for all $0 \leq i \leq d, 0 \leq j \leq n_H, 0 \leq k \leq m$

$$v_{kj} = v_{kj} + \eta(t_k - z_k) f'(net_k^*) y_j$$

$$v_{k0} = v_{k0} + \eta(t_k - z_k) f'(net_k^*)$$

$$w_{ji} = w_{ji} + \eta f'(net_j) x^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

$$w_{j0} = w_{j0} + \eta f'(net_j) \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

$$t = t + 1$$

until $\|J\| < \theta$

3. return v, w

Batch Back Propagation

- This is the **true** gradient descent, (unlike stochastic propagation)
- For simplicity, derived backpropagation for a single sample objective function:

$$J(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{c=1}^m (\mathbf{t}_c - \mathbf{z}_c)^2$$

- The full objective function:

$$J(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{i=1}^n \sum_{c=1}^m (\mathbf{t}_c^{(i)} - \mathbf{z}_c^{(i)})^2$$

- Derivative of full objective function is just a sum of derivatives for each sample:

$$\frac{\partial}{\partial \mathbf{w}} J(\mathbf{w}, \mathbf{v}) = \frac{1}{2} \sum_{i=1}^n \frac{\partial}{\partial \mathbf{w}} \left(\sum_{c=1}^m (\mathbf{t}_c^{(i)} - \mathbf{z}_c^{(i)})^2 \right)$$

already derived this

Batch Back Propagation

- For example,

$$\frac{\partial J}{\partial \mathbf{w}_{ji}} = \sum_{p=1}^n -f'(\mathit{net}_j) \mathbf{x}_p^{(i)} \sum_{k=1}^m (t_k - \mathbf{z}_k) f'(\mathit{net}_k^*) \mathbf{v}_{kj}$$

Batch Back Propagation

1. Initialize n_H , w , v , θ , η , $t = 0$

2. do

$$\Delta v_{kj} = \Delta v_{k0} = \Delta w_{ji} = \Delta w_{j0} = 0$$

for all $1 \leq p \leq n$

for all $0 \leq i \leq d$, $0 \leq j \leq n_H$, $0 \leq k \leq m$

$$\Delta v_{kj} = \Delta v_{kj} + \eta(t_k - z_k) f'(net_k^*) y_j$$

$$\Delta v_{k0} = \Delta v_{k0} + \eta(t_k - z_k) f'(net_k^*)$$

$$\Delta w_{ji} = \Delta w_{ji} + \eta f'(net_j) x_p^{(i)} \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

$$\Delta w_{j0} = \Delta w_{j0} + \eta f'(net_j) \sum_{k=1}^m (t_k - z_k) f'(net_k^*) v_{kj}$$

$$v_{kj} = v_{kj} + \Delta v_{kj}; v_{k0} = v_{k0} + \Delta v_{k0}; w_{ji} = w_{ji} + \Delta w_{ji}; w_{j0} = w_{j0} + \Delta w_{j0}$$

$$t = t + 1$$

until $\|J\| < \theta$

3. return v , w

Training Protocols

1. Batch

- True gradient descent

2. Stochastic

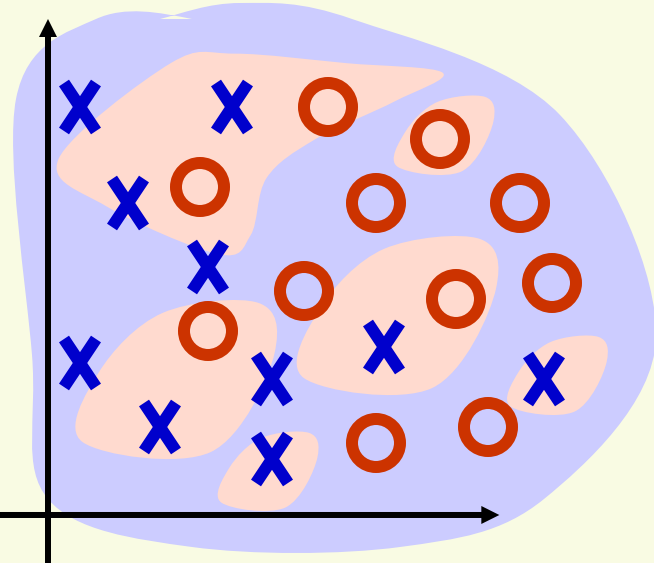
- Faster than batch method
- Usually the recommended way

3. Online

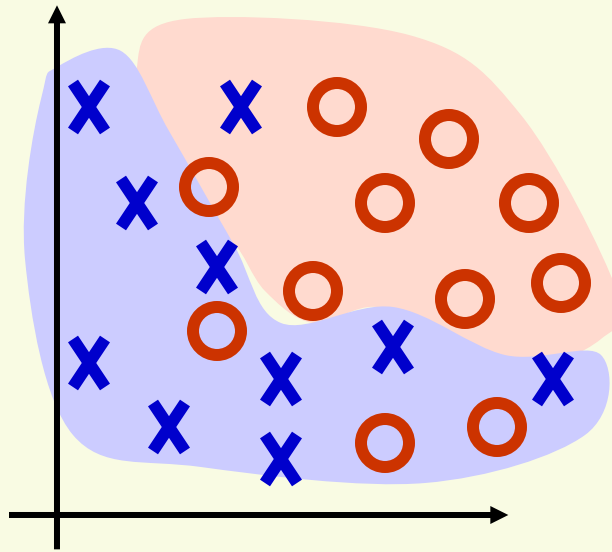
- Used when number of samples is so large it does not fit in the memory
- Dependent on the order of sample presentation
- Should be avoided when possible

MNN Training

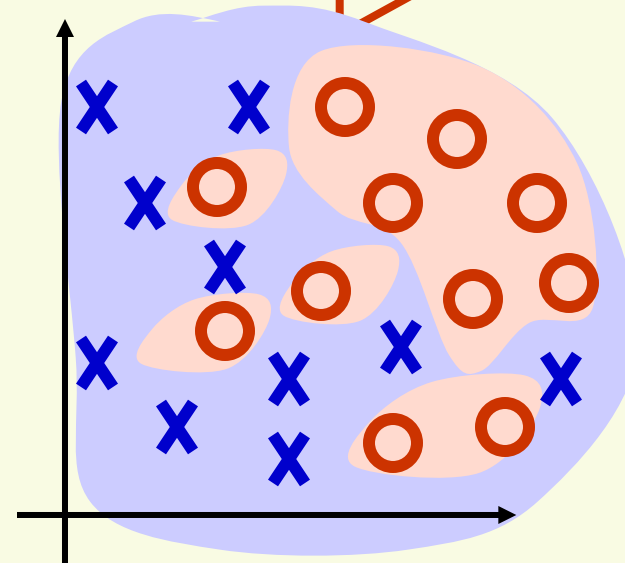
training time



Large training error: in the beginning random decision regions



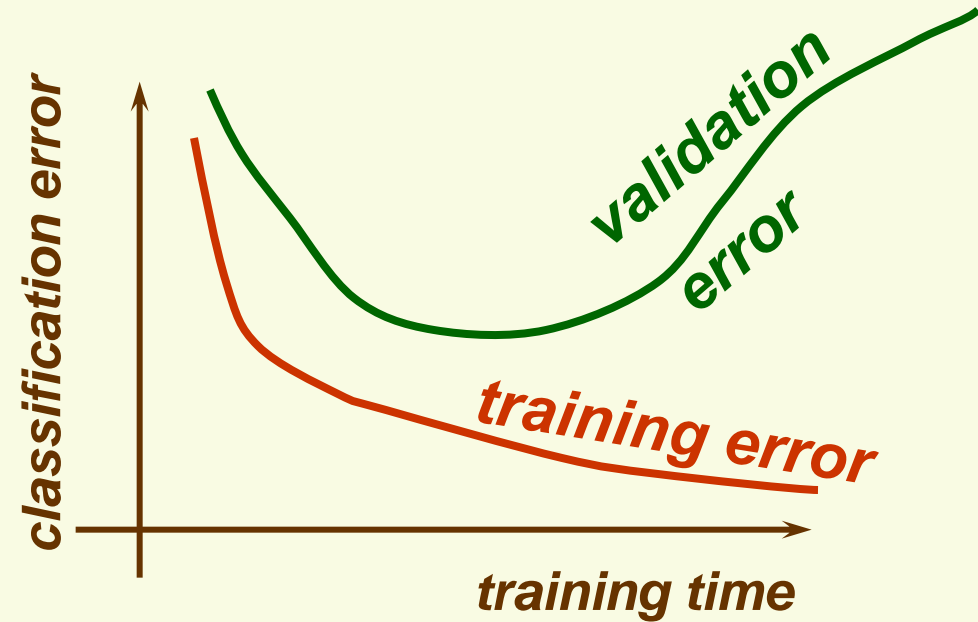
Small training error: decision regions improve with time



Zero training error: decision regions separate training data perfectly, but we overfitted the network

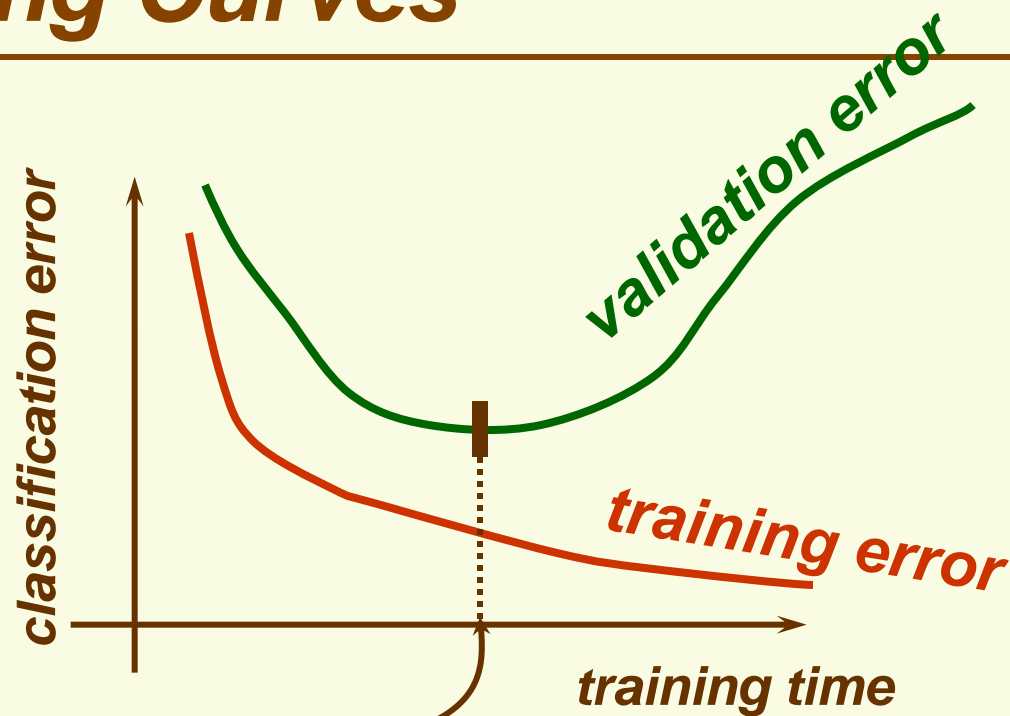
MNN Learning Curves

- **Training data:** data on which learning (gradient descent for MNN) is performed
- **Validation data:** used to assess network generalization capabilities
- Training error typically goes down, since with enough hidden units, can find discriminant function which classifies training patterns exactly



- Validation error first goes down, but then goes up since at some point we start to **overfit** the network to the validation data

Learning Curves



- this is a good time to stop training, since after this time we start to overfit
- Stopping criterion is part of training phase, thus validation data is part of the training data
- To assess how the network will work on the unseen examples, we still need test data

Learning Curves

- validation data is used to determine “parameters”, in this case when learning should stop



- Stop training after the first local minimum on validation data
- We are assuming performance on test data will be similar to performance on validation data

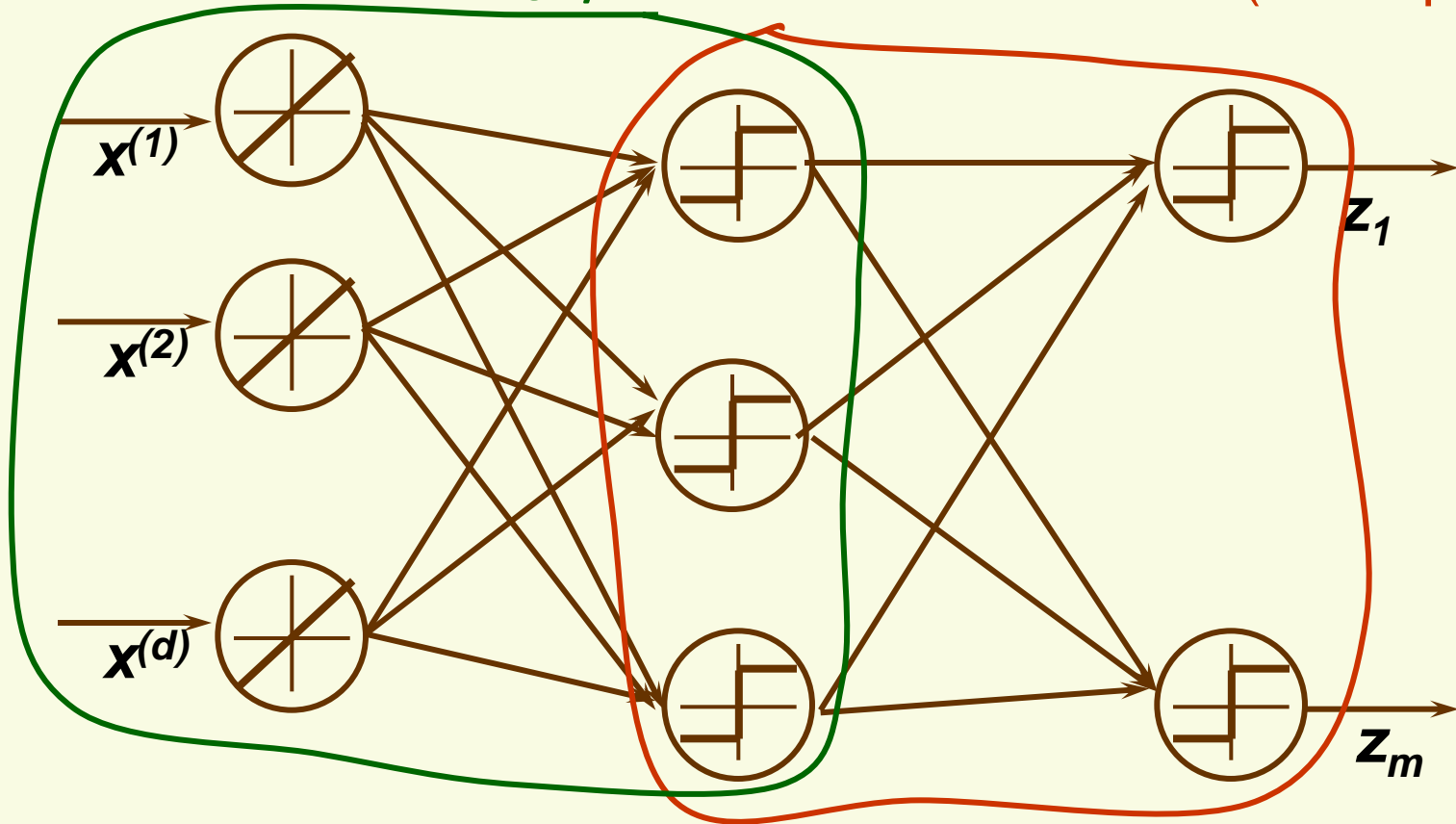
Data Sets

- **Training data**
 - data on which learning is performed
- **Validation data**
 - validation data is used to determine any free parameters of the classifier
 - k in the knn neighbor classifier
 - h for parzen windows
 - number of hidden layers in the MNN
 - etc
- **Test data**
 - used to assess network generalization capabilities

MNN as Nonlinear Mapping

this module implements
nonlinear input mapping φ

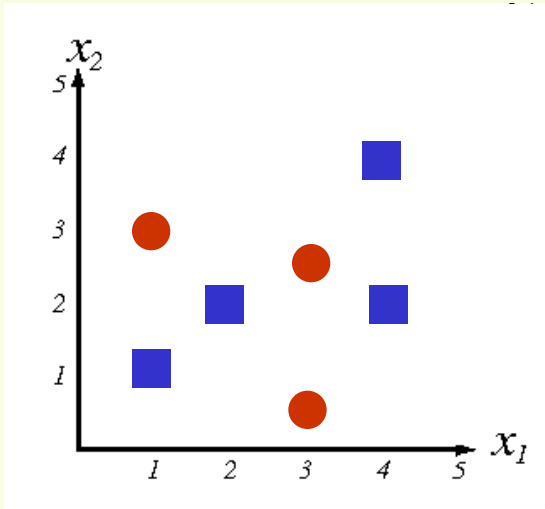
this module implements
linear classifier (Perceptron)



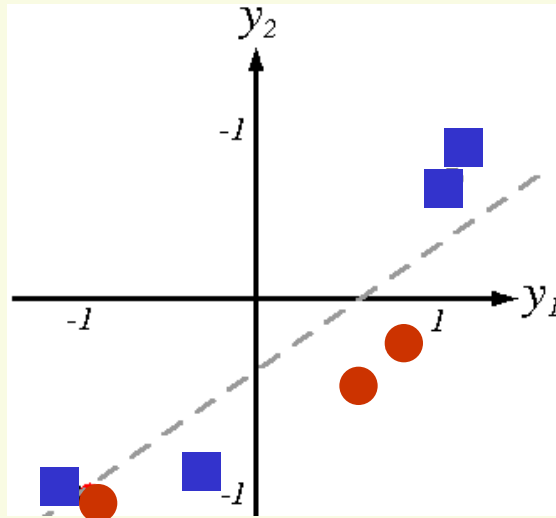
MNN as Nonlinear Mapping

- Thus MNN can be thought as learning 2 things at the same time
 - the nonlinear mapping of the inputs
 - linear classifier of the nonlinearly mapped inputs

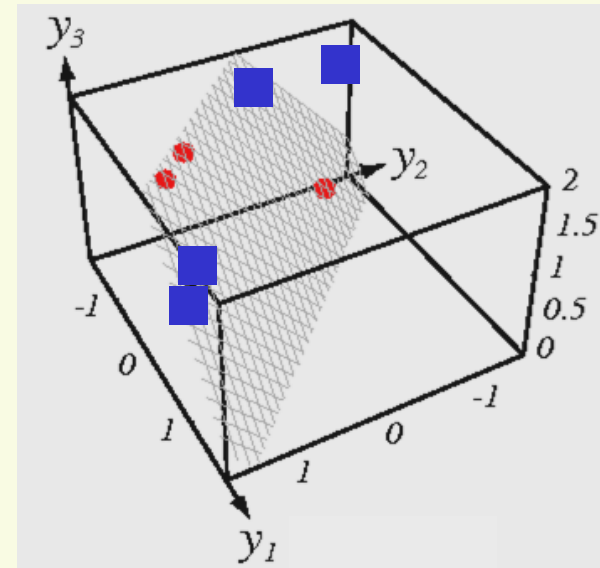
MNN as Nonlinear Mapping



original feature space \mathbf{x} ; patterns are not linearly separable



MNN finds nonlinear mapping $\mathbf{y}=\varphi(\mathbf{x})$ to 2 dimensions (2 hidden units); patterns are almost linearly separable



MNN finds nonlinear mapping $\mathbf{y}=\varphi(\mathbf{x})$ to 3 dimensions (3 hidden units) that; patterns are linearly separable

Concluding Remarks

- Advantages
 - MNN can learn complex mappings from inputs to outputs, based only on the training samples
 - Easy to use
 - Easy to incorporate a lot of heuristics
- Disadvantages
 - It is a “black box”, that is difficult to analyze and predict its behavior
 - May take a long time to train
 - May get trapped in a bad local minima
 - A lot of “tricks” to implement for the best performance