CS4442/9542b
Artificial Intelligence II
prof. Olga Veksler

Lecture 5

Machine Learning

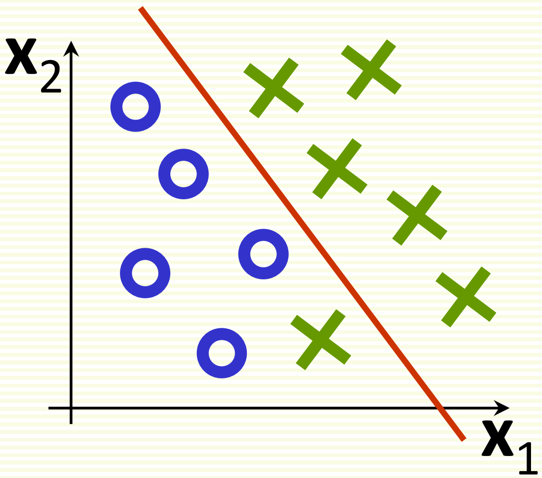
Neural Networks

Many presentation Ideas are due to Andrew NG

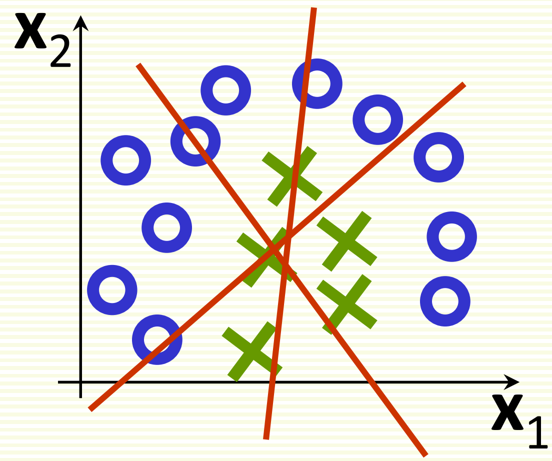
Outline

- Motivation
  - Non linear discriminant functions
- Introduction to Neural Networks
  - Inspiration from Biology
  - History
- Perceptron
- Multilayer Perceptron
- Practical Tips for Implementation

Need for Non-Linear Discriminant

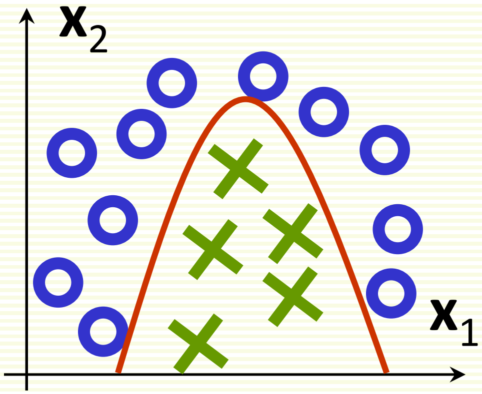


$\mathbf{g}(\mathbf{x}) = \mathbf{w}_0 + \mathbf{w}_1\mathbf{x}_1 + \mathbf{w}_2\mathbf{x}_2$



- Previous lecture studied linear discriminant
- Works for linearly (or almost) separable cases
- Many problems are far from linearly separable
  - underfitting with linear model

Need for Non-Linear Discriminant



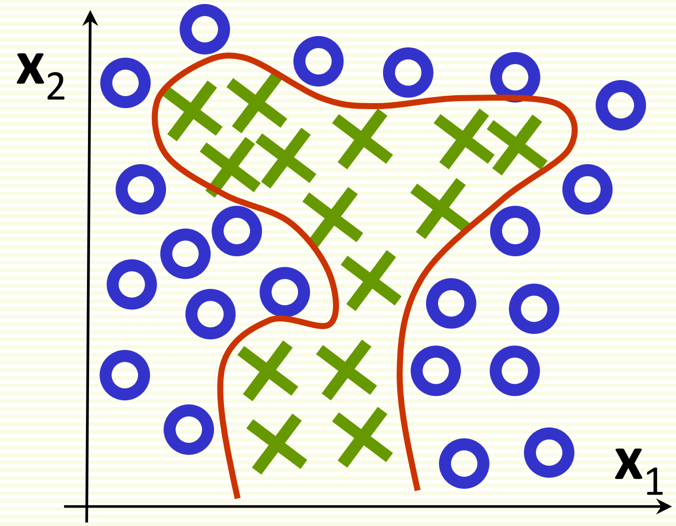
- Can use other discriminant functions, like quadratics

  $\mathbf{g}(\mathbf{x}) = \mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2+ \mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 +\mathbf{w}_{11}\mathbf{x}_1^2 +\mathbf{w}_{22}\mathbf{x}_2^2$

- Methodology is almost the same as in the linear case:
  - $\mathbf{f}(\mathbf{x}) = \text{sign}(\mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2+\mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 +\mathbf{w}_{11}\mathbf{x}_1^2 + \mathbf{w}_{22}\mathbf{x}_2^2)$
  - $\mathbf{z} = [\,1 \quad \mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_1\mathbf{x}_2 \quad \mathbf{x}_1^2 \quad \mathbf{x}_2^2\,]$
  - $\mathbf{a} = [\,\mathbf{w}_0 \quad \mathbf{w}_1 \quad \mathbf{w}_2 \quad \mathbf{w}_{12} \quad \mathbf{w}_{11} \quad \mathbf{w}_{22}\,]$
  - "normalization": multiply negative class samples by -1
  - gradient descent to minimize Perceptron objective function

$$\mathbf{J}_p(\mathbf{a}) = \sum_{\mathbf{z}\in \mathbf{Z}(\mathbf{a})} \left(-\mathbf{a}^t\mathbf{z}\right)$$

Need for Non-Linear Discriminant



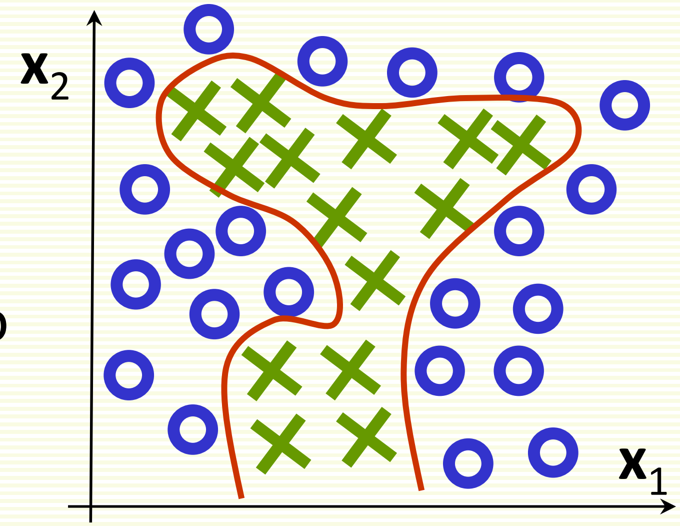
- May need highly non-linear decision boundaries
- This would require too many high order polynomial terms to fit

$$
\begin{aligned}
\mathbf{g}(\mathbf{x}) = \;& \mathbf{w}_0 + \mathbf{w}_1\mathbf{x}_1 + \mathbf{w}_2\mathbf{x}_2 + \\
& + \mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 + \mathbf{w}_{11}\mathbf{x}_1^{\;2} + \mathbf{w}_{22}\mathbf{x}_2^{\;2} + \\
& + \mathbf{w}_{111}\mathbf{x}_1^{\;3} + \mathbf{w}_{112}\mathbf{x}_1^{\;2}\mathbf{x}_2 + \mathbf{w}_{122}\mathbf{x}_1\mathbf{x}_2^{\;2} + \mathbf{w}_{222}\mathbf{x}_2^{\;3} + \\
& + \text{even more terms of degree } \mathbf{4} \\
& + \text{super many terms of degree } \mathbf{k}
\end{aligned}
$$

- For **n** features, there O($\mathbf{n}^{\mathbf{k}}$) polynomial terms of degree **k**
- Many real world problems are modeled with hundreds and even thousands features
  - $100^{\mathbf{10}}$ is too large of function to deal with

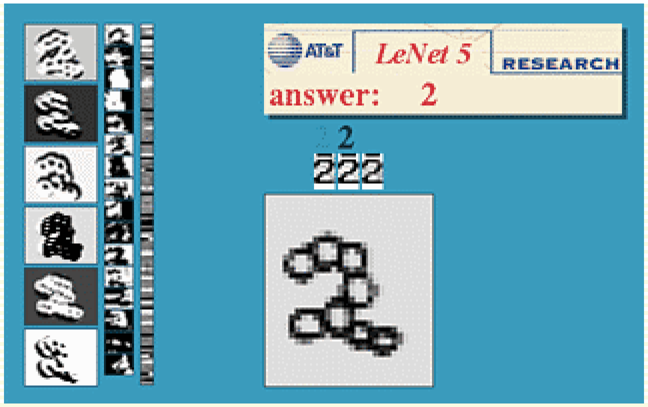
Neural Networks

- Neural Networks correspond to some discriminant function $\mathbf{g}_{\mathbf{NN}}(\mathbf{x})$
- Can carve out arbitrarily complex decision boundaries without requiring so many terms as polynomial functions
- Neural Nets were inspired by research in how human brain works
- But also proved to be quite successful in practice
- Are used nowadays successfully for a wide variety of applications
  - took some time to get them to work
  - now used by US post for postal code recognition



Neural Nets: Character Recognition

- http://yann.lecun.com/exdb/lenet/index.html



Yann LeCun et. al.

Brain vs. Computer

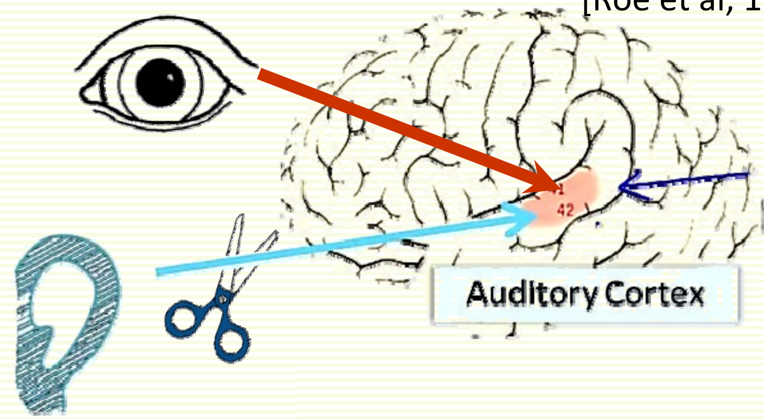
- usually one very fast processor
- high reliability
- designed to solve logic and arithmetic problems
- absolute precision
- can solve a gazillion arithmetic and logic problems in an hour

- huge number of parallel but relatively slow and unreliable processors
- not perfectly precise, not perfectly reliable
- evolved (in a large part) for pattern recognition
- learns to solve various PR problems

seek inspiration for classification from human brain

One Learning Algorithm Hypothesis

[Roe et al, 1992]



- Brain does many different things
- Seems like it runs many different "programs"
- Seems we have to write tons of different programs to mimic brain
- Hypothesis: there is a single underlying learning algorithm shared by different parts of the brain
- Evidence from neuro-rewiring experiments
  - Cut the wire from ear to auditory cortex
  - Route signal from eyes to the auditory cortex
  - Auditory cortex learns to see
    - animals will eventually learn to perform a variety of object recognition tasks
- There are other similar rewiring experiments

Seeing with Tongue

- Scientists use the amazing ability of the brain to learn to retrain brain tissue
- Seeing with tongue
  - BrainPort Technology
  - Camera connected to a tongue array sensor
  - Pictures are "painted" on the tongue
    - Bright pixels correspond to high voltage
    - Gray pixels correspond to medium voltage
    - Black pixels correspond to no voltage
  - Learning takes from 2-10 hours
  - Some users describe experience resembling a low resolution version of vision they once had
    - able to recognize high contrast object, their location, movement

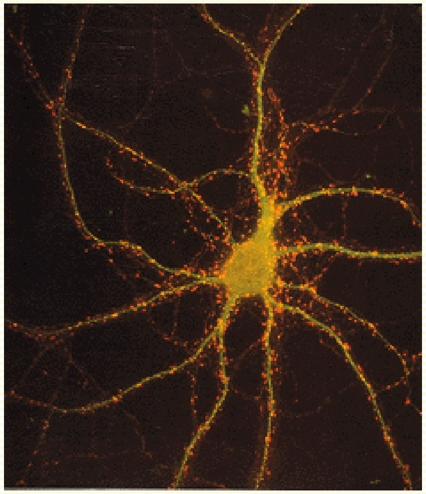
tongue array sensor

One Learning Algorithm Hypothesis

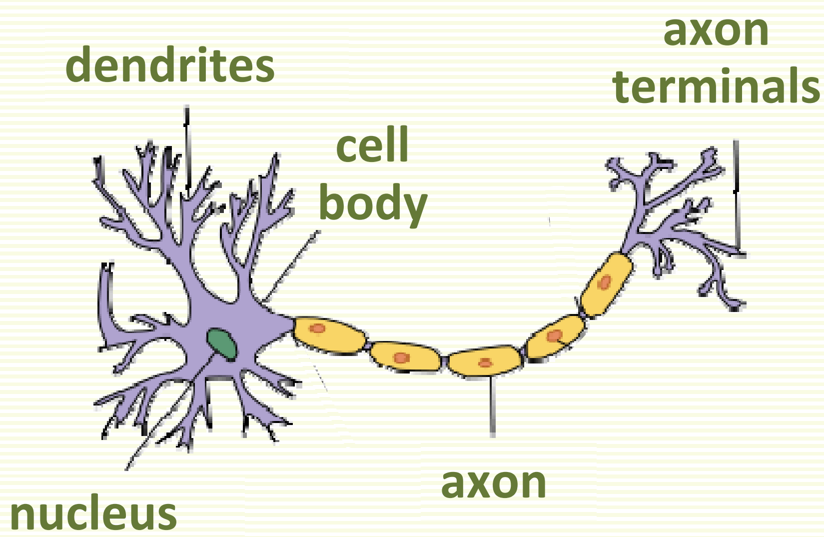
- Experimental evidence that we can plug any sensor to any part of the brain, and brain can learn how to deal with it
- Since the same physical piece of brain tissue can process sight, sound, etc.
- Maybe there is one learning algorithm can process sight, sound, etc.
- Maybe we need to figure out and implement an algorithm that approximates what the brain does
- Neural Networks were developed as a simulation of networks of neurons in human brain

Neuron: Basic Brain Processor

- Neurons (or nerve cells) are special cells that process and transmit information by electrical signaling
  - in brain and also spinal cord
- Human brain has around 10^{11} neurons
- A neuron connects to other neurons to form a network
- Each neuron cell communicates to anywhere from 1000 to 10,000 other neurons



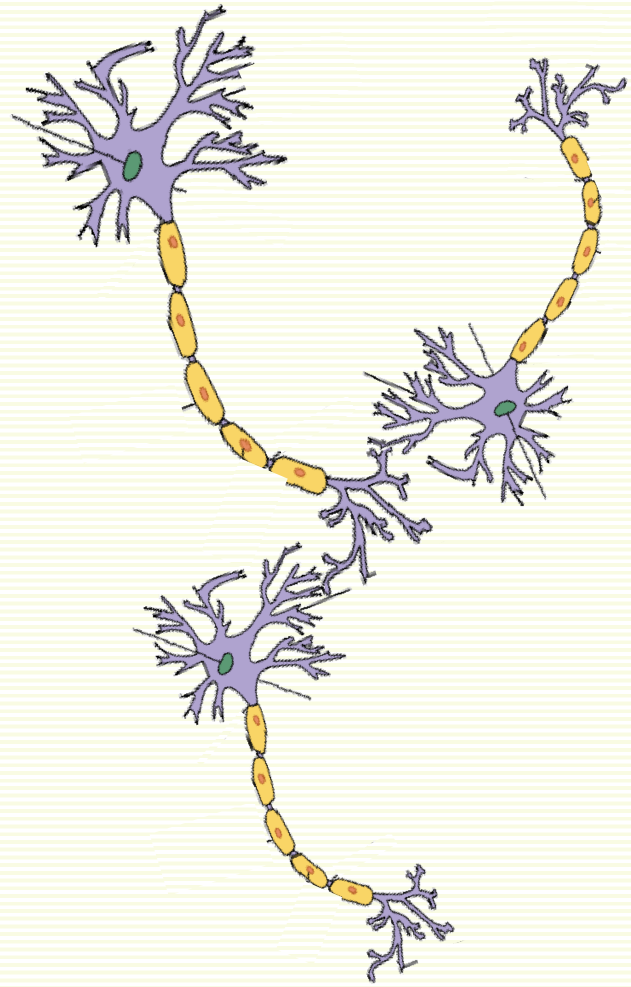
Neuron: Main Components



- **cell body**
  - computational unit
- **dendrites**
  - “input wires”, receive inputs from other neurons
  - a neuron may have thousands of dendrites, usually short
- **axon**
  - “output wire”, sends signal to other neurons
  - single long structure (up to 1 meter)
  - splits in possibly thousands branches at the end, “axon terminals”

Neurons in Action (Simplified Picture)

- Cell body collects and processes signals from other neurons through dendrites
- If there the strength of incoming signals is large enough, the cell body sends an electricity pulse (a spike) to its axon
- Its axon, in turn, connects to dendrites of other neurons, transmitting spikes to other neurons
- This is the process by which all human thought, sensing, action, etc. happens



Artificial 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?
  - Can be considered to be the birth of AI
- 1949, D. Hebb, introduced the first (purely pshychological) 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
    - that’s what we studied in lecture on linear classifiers
  - Algorithm to train the perceptron network
  - 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

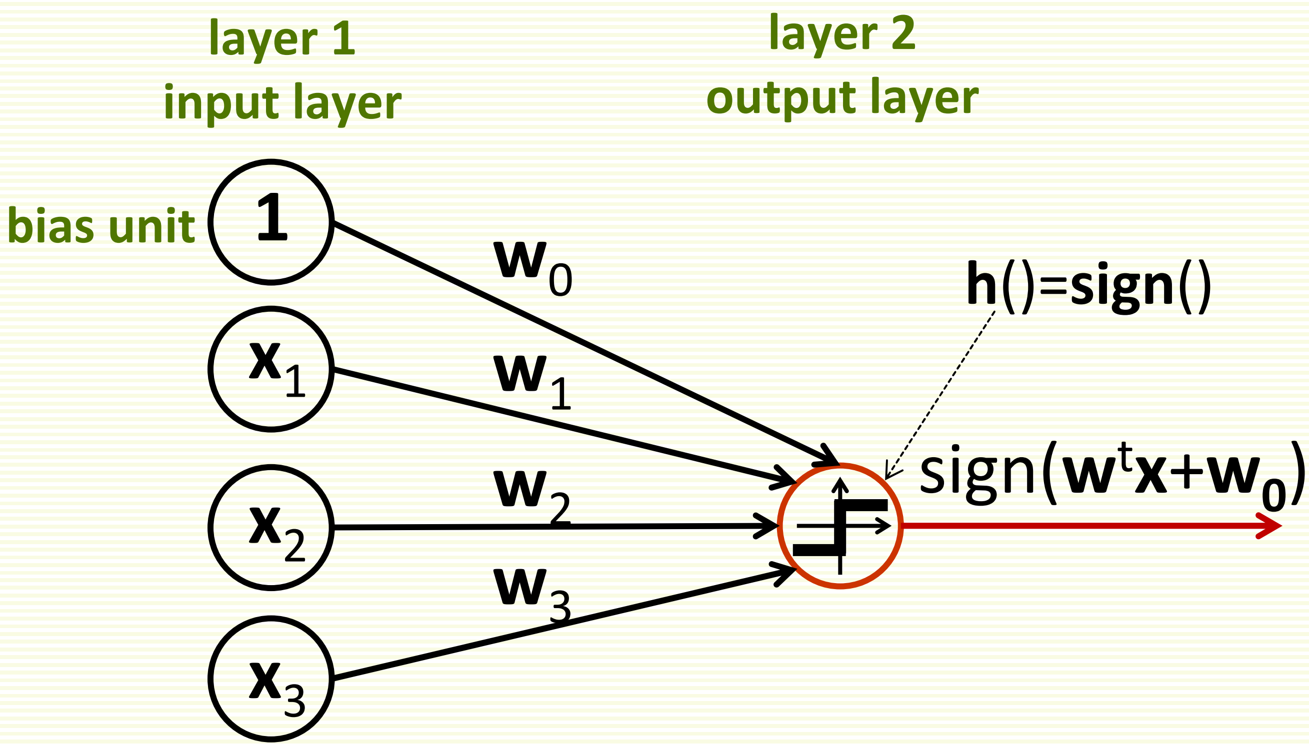
ANN History: Stagnation

- 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

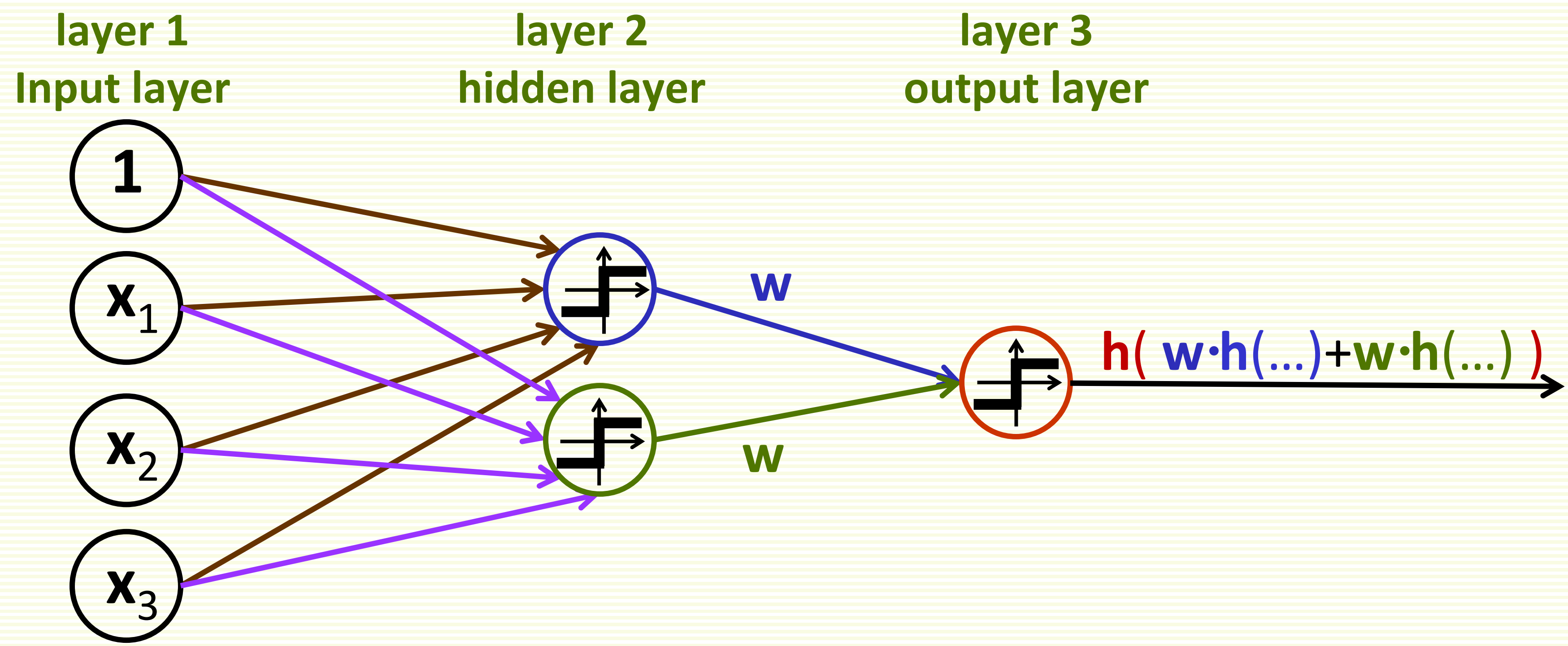
- Revival of ANN in 1980's
- 1982, J. Hopfield
  - New kind of networks (Hopfield's networks)
  - Not just model of how human brain might work, but also how to create useful devices
    - Implements associative memory
- 1982 joint US-Japanese conference on ANN
  - US worries that it will stay behind
- Many examples of mulitlayer NN appear
- 1986, re-discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows a network to learn not linearly separable classes

Artificial Neural Nets (ANN): Perceptron



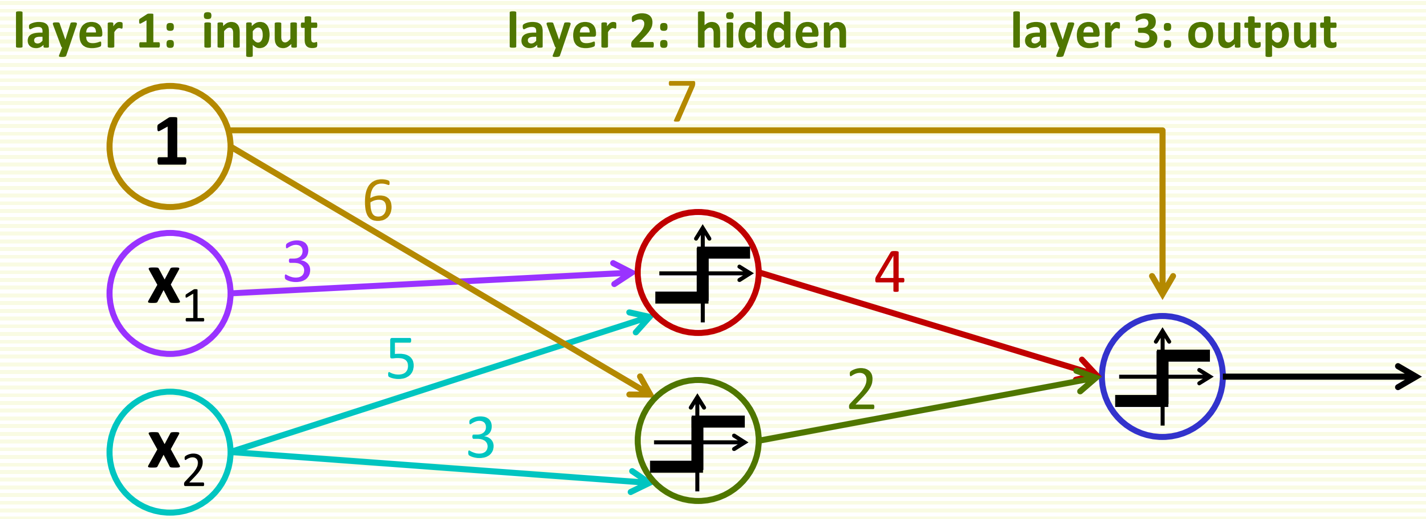
- Linear classifier $\mathbf{f}(\mathbf{x})$ = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w_0}$) is a single neuron "net"
- Input layer units output features, except bias outputs "1"
- Output layer unit applies **sign**() or some other function **h**()
- **h**() is also called an *activation function*

Multilayer Neural Network (MNN)



- First hidden unit outputs: $h(\ldots) = h(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3)$
- Second hidden unit outputs: $h(\ldots) = h(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3)$
- Network corresponds to classifier $\mathbf{f}(\mathbf{x}) = h(\ w \cdot h(\ldots) + w \cdot h(\ldots)\ )$
- More complex than Perceptron, more complex boundaries

MNN Small Example



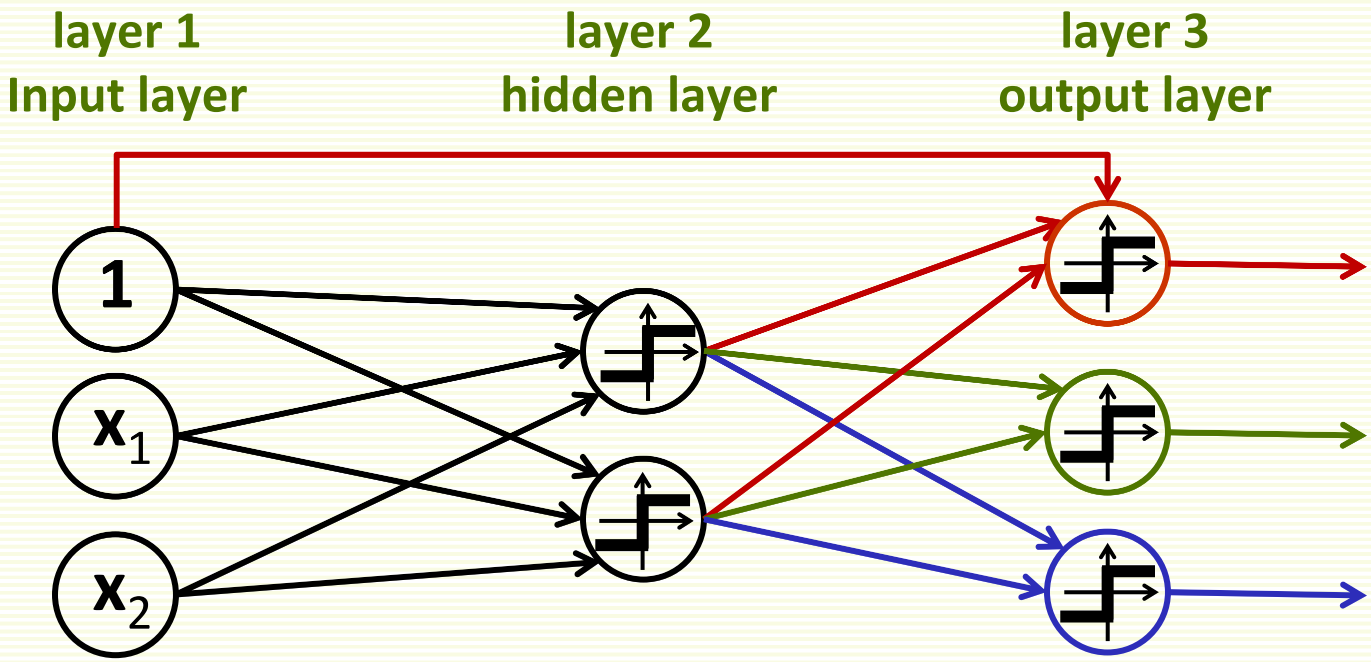
- Let activation function **h**() = sign()
- MNN Corresponds to classifier

$$\mathbf{f}(\mathbf{x}) = \text{sign}(\ 4\cdot\mathbf{h}(\ldots)+2\cdot\mathbf{h}(\ldots) + 7\ )$$

$$= \text{sign}(4\cdot\text{sign}(3\mathbf{x}_1+5\mathbf{x}_2)+2\cdot\text{sign}(6+3\mathbf{x}_2) + 7)$$

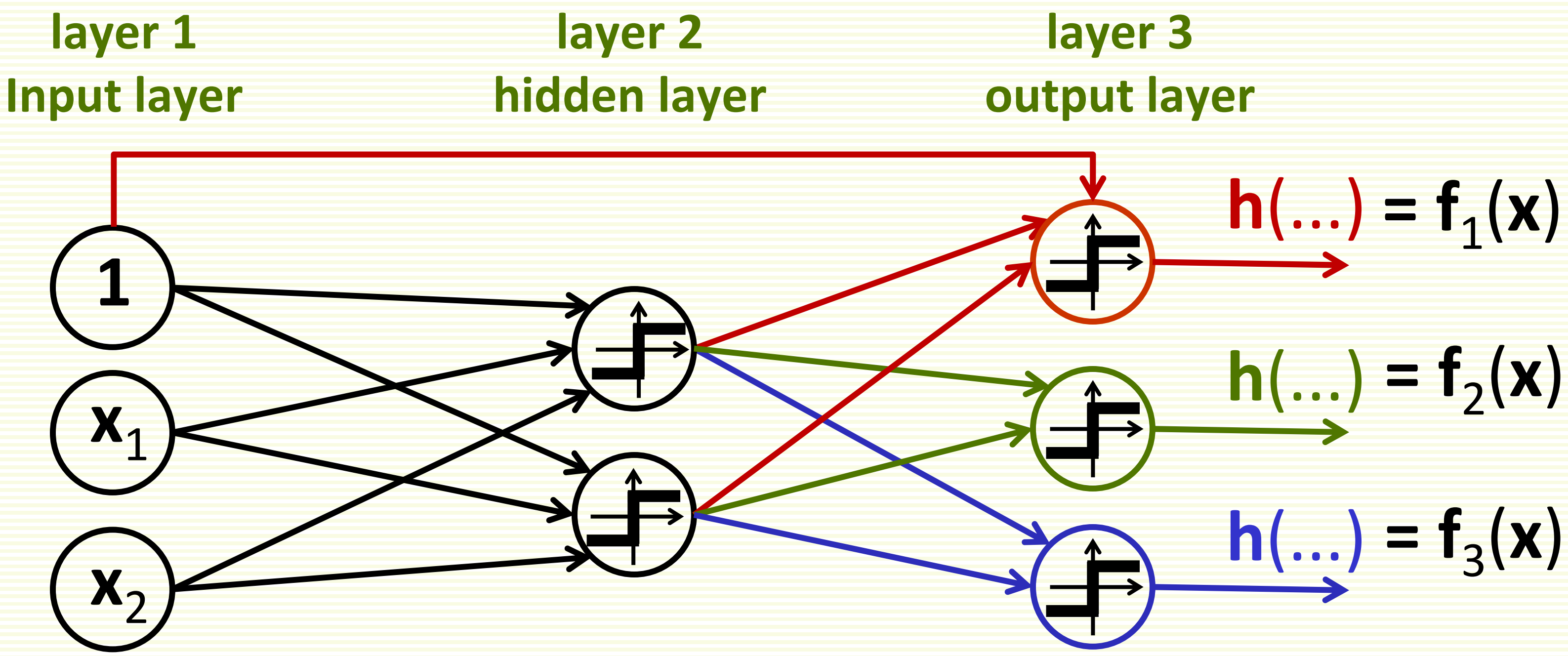
- MNN terminology: computing **f**(**x**) is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

MNN: Multiple Classes



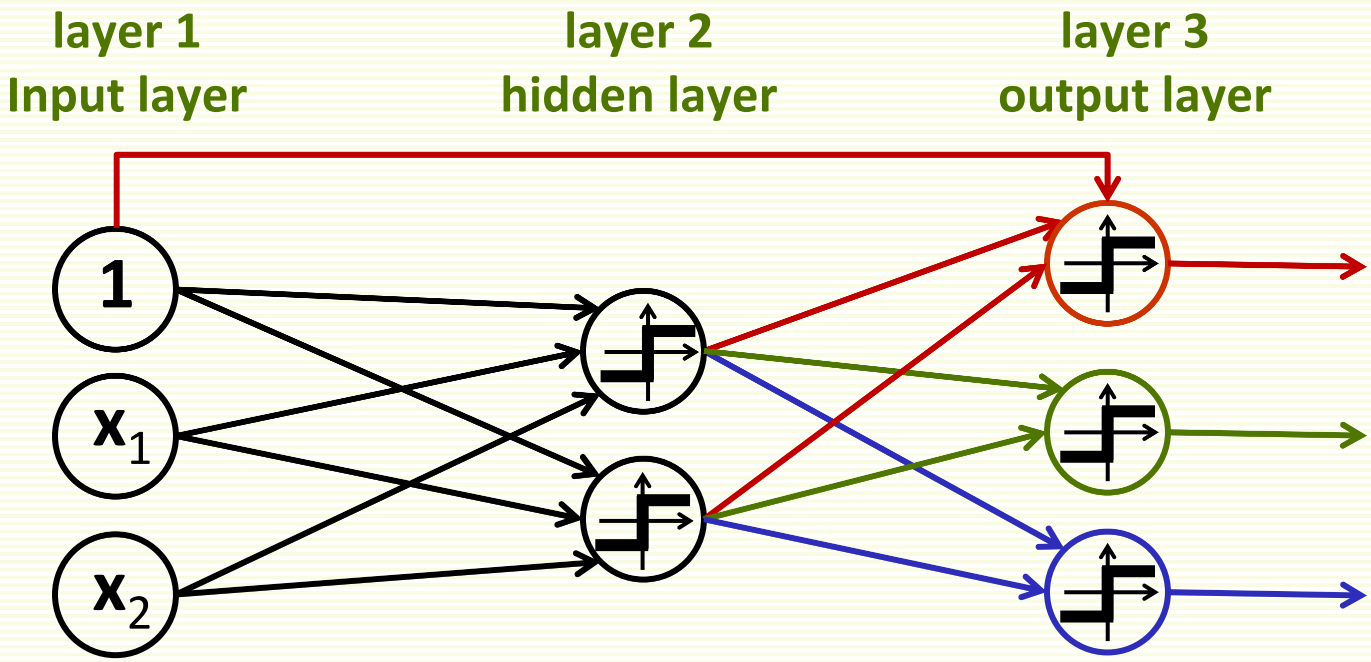
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, usually drawn in layer 1

MNN: General Structure



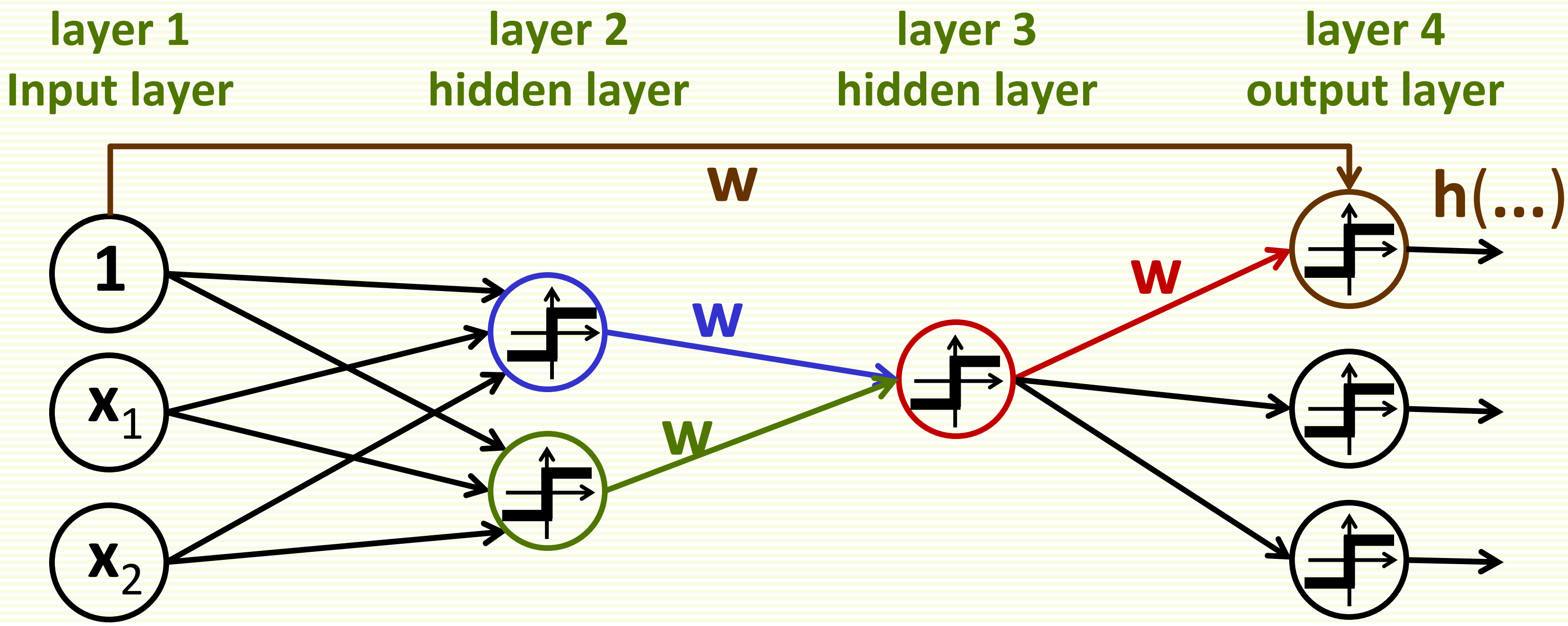
- $\mathbf{f}(\mathbf{x}) = [\mathbf{f}_1(\mathbf{x}), \mathbf{f}_2(\mathbf{x}), \mathbf{f}_3(\mathbf{x})]$ is multi-dimensional
- Classification:
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$ is largest, decide class 3

MNN: General Structure



- Input layer: **d** features, **d** input units
- Output layer: **m** classes, **m** output units
- Hidden layer: how many units?

MNN: General Structure

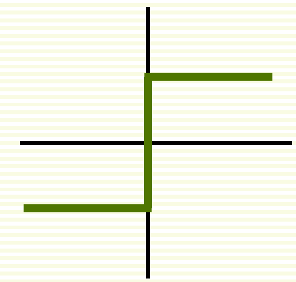


- Can have more than 1 hidden layer
  - **i**th layer connects to (**i+1**)th layer
    - except bias unit can connect to any layer
  - can have different number of units in each hidden layer
- First output unit outputs:

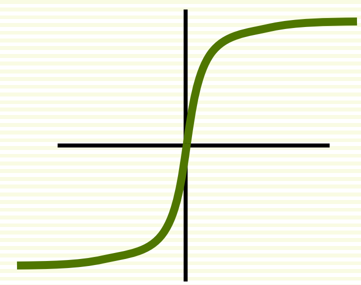
$$h(\ldots) = h(\, w \cdot h(\ldots) + w \,) = h(\, w \cdot h(w \cdot h(\ldots) + w \cdot h(\ldots)) + w \,)$$

MNN: Activation Function

- **h**() = **sign**() is discontinuous, not good for gradient descent



- Instead can use continuous sigmoid function



- Or another differentiable function
- Can even use different activation functions at different layers/units
- From now, assume **h**() is a differentiable function

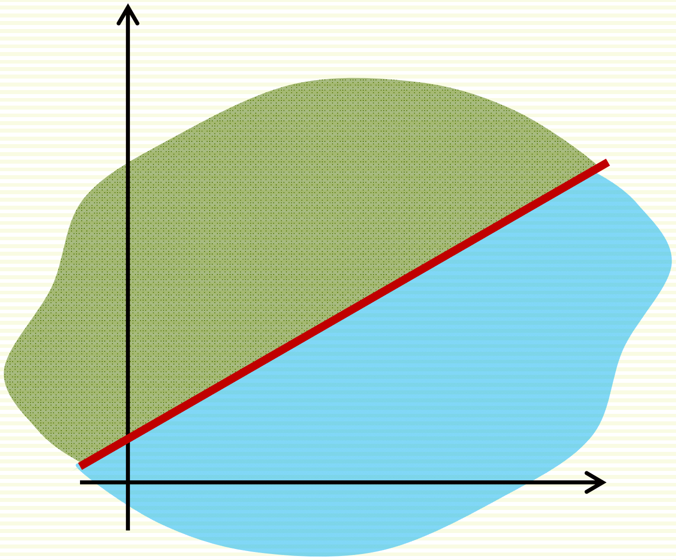
MNN: Overview

- A neural network corresponds to a classifier **f**(**x**,**w**) that can be rather complex
  - complexity depends on the number of hidden layers/units
  - f(x,w) is a composition of many functions
    - easier to visualize as a network
    - notation gets ugly
- To train neural network, just as before
  - formulate an objective function **J**(**w**)
  - optimize it with gradient descent
  - That's all!
  - Except we need quite a few slides to write down details due to complexity of **f**(**x**,**w**)

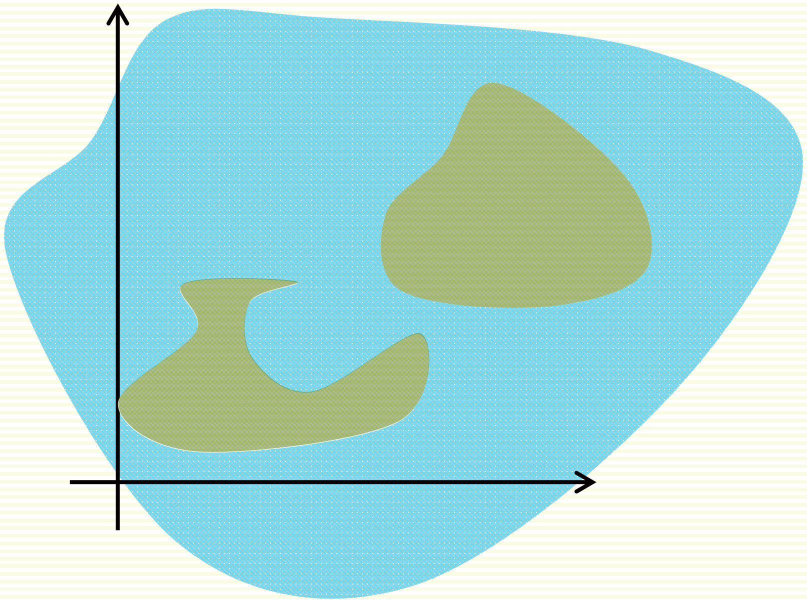
Expressive Power of MNN

- Every continuous function from input to output can be implemented with enough hidden units, 1 hidden layer, and proper *nonlinear* activation functions
  - easy to show that with linear activation function, multilayer neural network is equivalent to perceptron
- This is more of theoretical than practical interest
  - Proof is not constructive (does not tell how construct MNN)
  - Even if constructive, would be of no use, we do not know the desired function, our goal is to learn it through the samples
  - But this result gives confidence that we are on the right track
    - MNN is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron

Decision Boundaries



- Perceptron (single layer neural net)

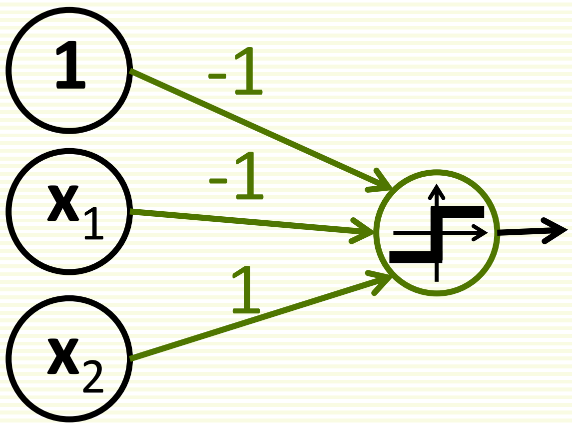


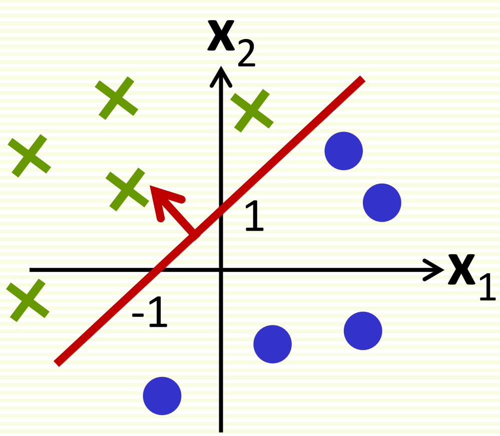
- Arbitrarily complex decision regions
- Even not contiguous

Nonlinear Decision Boundary: Example

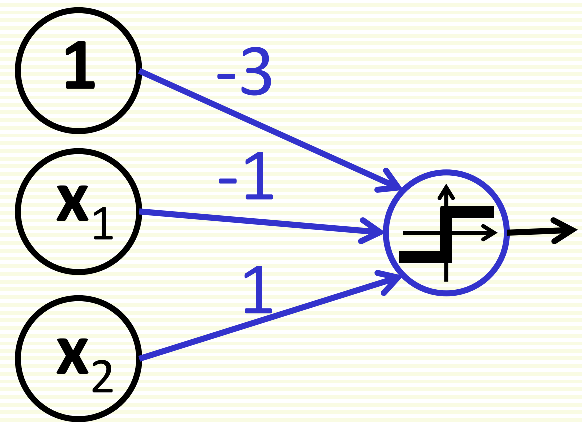
- Start with two Perceptrons, **h**() = sign()

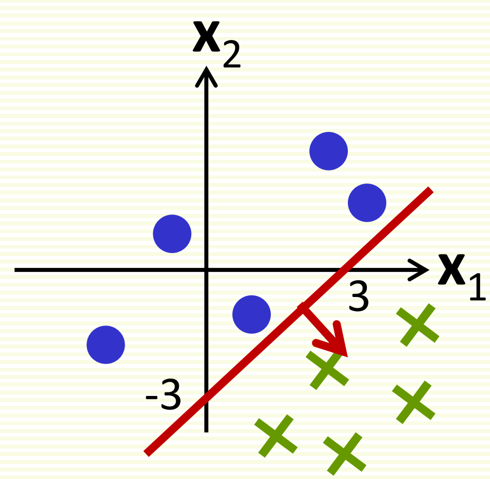
$-\mathbf{x}_1 + \mathbf{x}_2 - 1 > 0 \Rightarrow \text{class } 1$





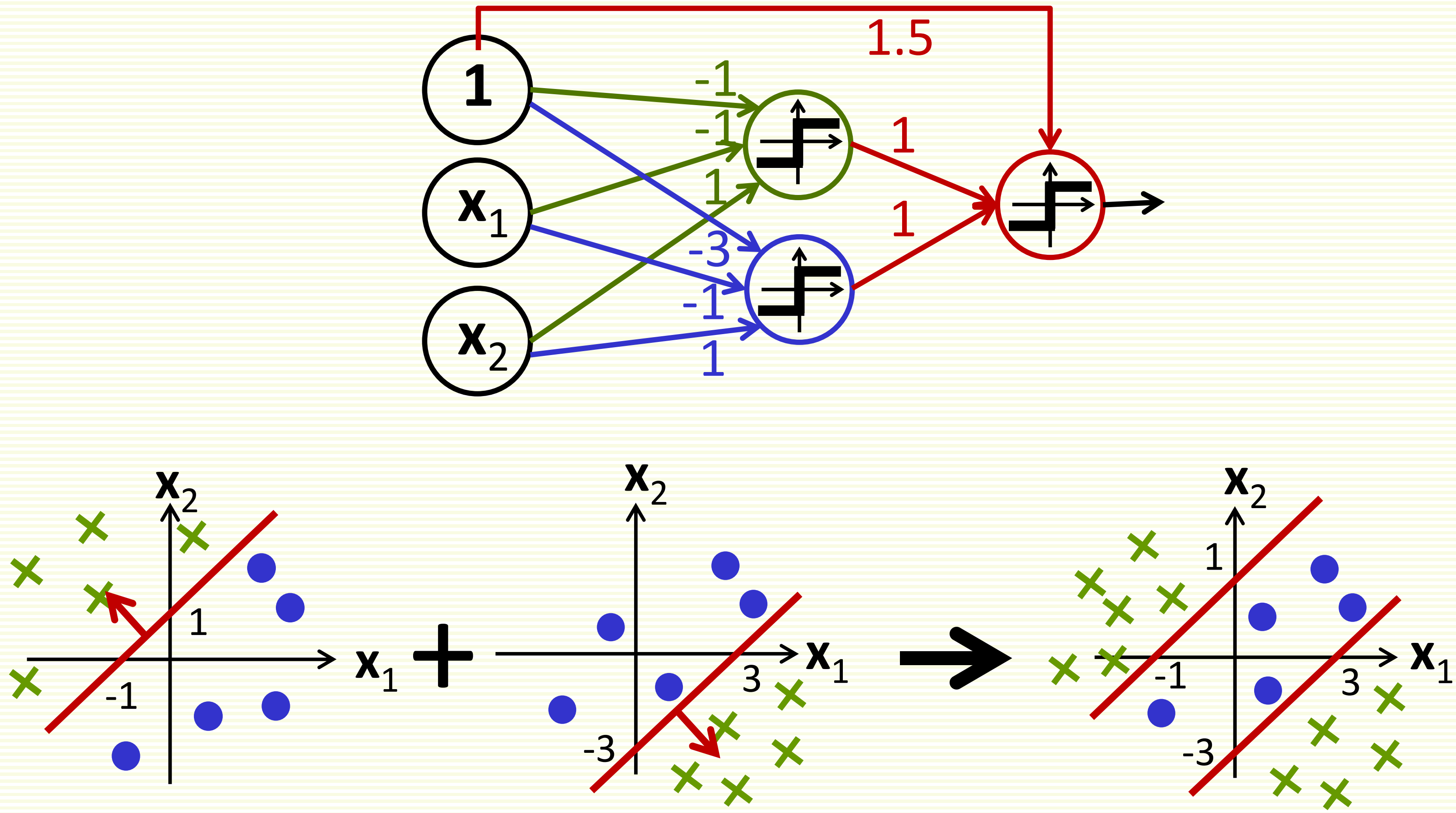
$-\mathbf{x}_1 + \mathbf{x}_2 - 3 > 0 \Rightarrow \text{class } 1$





Nonlinear Decision Boundary: Example

- Now combine them into a 3 layer NN

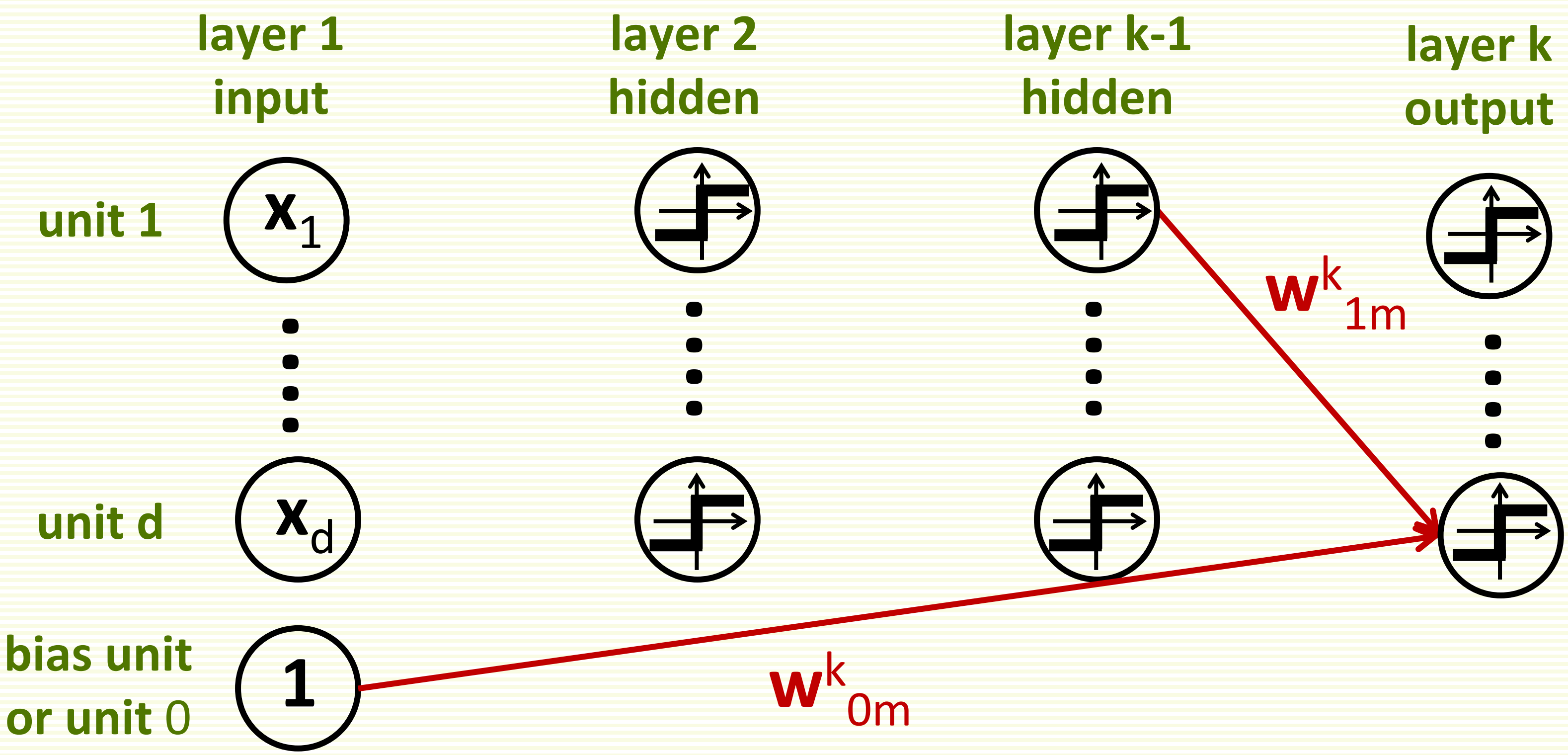


MNN: Modes of Operation

- For Neural Networks, due to historical reasons, training and testing stages have special names
  - **Backpropagation (or training)**

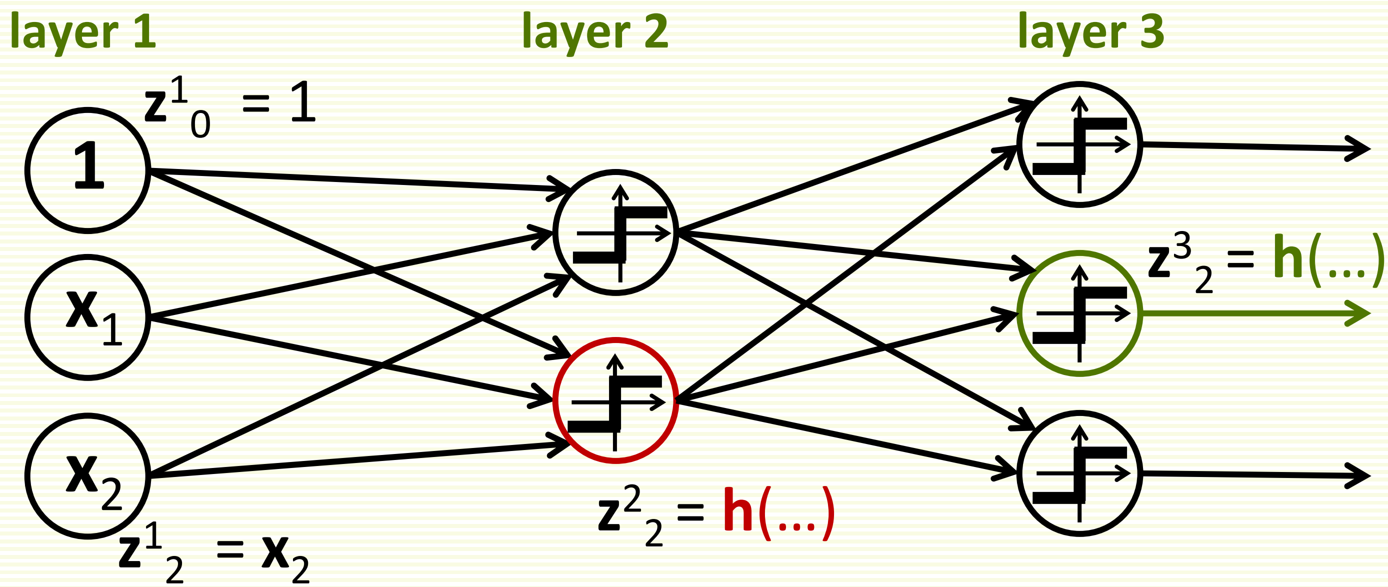
    Minimize objective function with gradient descent
  - **Feedforward (or testing)**

MNN: Notation for Edge Weights



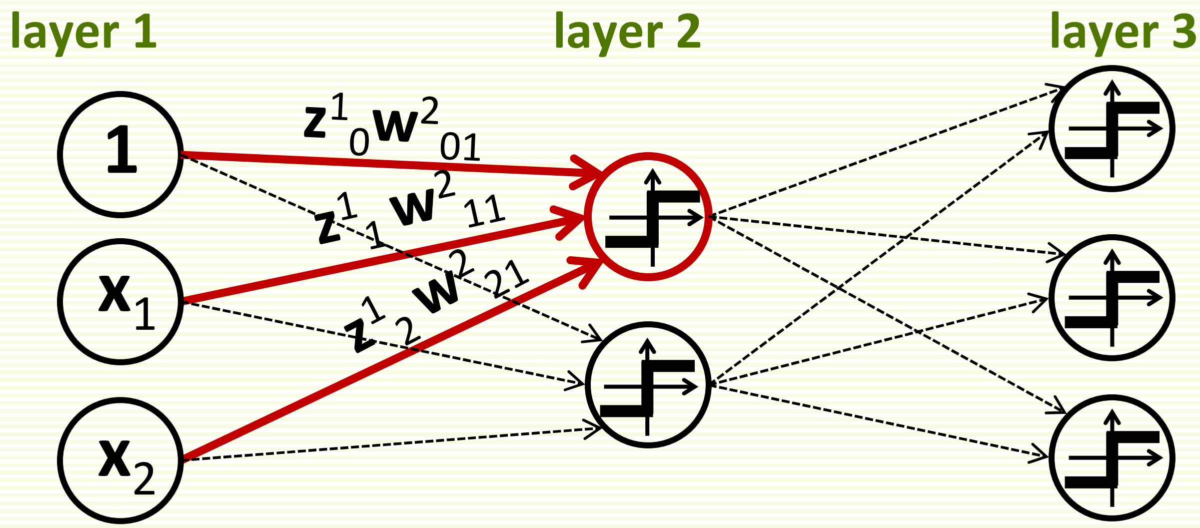
- $\mathbf{w}^k_{pj}$ is edge weight from unit **p** in layer **k**-1 to unit **j** in layer **k**
- $\mathbf{w}^k_{0j}$ is edge weight from bias unit to unit **j** in layer **k**
- $\mathbf{w}^k_{j}$ is all weights to unit **j** in layer **k**, i.e. $\mathbf{w}^k_{0j}$, $\mathbf{w}^k_{1j}$, …, $\mathbf{w}^k_{N(k-1)j}$
  - **N**(k) is the number of units in layer k, excluding the bias unit

MNN: More Notation



- Denote the output of unit **j** in layer **k** as $\mathbf{z}^k_j$
- For the input layer (**k**=1), $\mathbf{z}^1_0 = 1$ and $\mathbf{z}^1_j = \mathbf{x}_j$, $\mathbf{j} \neq 0$
- For all other layers, (**k** > 1), $\mathbf{z}^k_j = \mathbf{h}(\ldots)$
- Convenient to set $\mathbf{z}^k_0 = 1$ for all **k**
- Set $\mathbf{z}^k = [\mathbf{z}^k_0, \mathbf{z}^k_1, \ldots, \mathbf{z}^k_{\mathbf{N}(k)}]$

MNN: More Notation



- Net activation at unit **j** in layer **k** > 1 is the sum of inputs

$$\mathbf{a}_j^k = \sum_{p=1}^{N_{k-1}} \mathbf{z}_p^{k-1}\mathbf{w}_{pj}^k + \mathbf{w}_{0j}^k = \sum_{p=0}^{N_{k-1}} \mathbf{z}_p^{k-1}\mathbf{w}_{pj}^k = \mathbf{z}^{k-1}\cdot \mathbf{w}_j^k$$

$$\mathbf{a}_1^2 = \mathbf{z}_0^1\mathbf{w}_{01}^2 + \mathbf{z}_1^1\mathbf{w}_{11}^2 + \mathbf{z}_2^1\mathbf{w}_{21}^2$$

- For **k** > 1, $\mathbf{z}^k{}_j = \mathbf{h}(\mathbf{a}^k{}_j)$

MNN: Class Representation

- **m** class problem, let Neural Net have **t** layers
- Let $\mathbf{x}^i$ be a example of class **c**
- It is convenient to denote its label as $\mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$ ⟵ row **c**
- Recall that $\mathbf{z}^t_c$ is the output of unit **c** in layer **t** (output layer)
- $\mathbf{f}(\mathbf{x}) = \mathbf{z}^t = \begin{bmatrix} \mathbf{z}^t_1 \\ \vdots \\ \mathbf{z}^t_c \\ \vdots \\ \mathbf{z}^t_m \end{bmatrix}$. If $\mathbf{x}^i$ is of class **c**, want $\mathbf{z}^t = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$ ⟵ row **c**

Training MNN: Objective Function

- Want to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i)$
- Use squared difference
- Let **w** be all edge weights in MNN collected in one vector
- Error on one example $\mathbf{x}^i$: $$\mathbf{J_i}(\mathbf{w}) = \frac{1}{2}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f_c}\left(\mathbf{x^i}\right) - \mathbf{y_c^i}\right)^2$$
- Error on all examples: $$\mathbf{J}(\mathbf{w}) = \frac{1}{2}\sum_{\mathbf{i}=1}^{\mathbf{n}}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f_c}\left(\mathbf{x^i}\right) - \mathbf{y_c^i}\right)^2$$
- Gradient descent:

```
initialize w to random
choose ε, α
while α||∇J(w)|| > ε
    w = w - α∇J(w)
```

Training MNN: Single Sample

- For simplicity, first consider error for one example $\mathbf{x}^i$

$$\mathbf{J_i}(\mathbf{w}) = \frac{1}{2}\left\|\mathbf{y^i} - \mathbf{f}(\mathbf{x^i})\right\|^2 = \frac{1}{2}\sum_{\mathbf{c}=1}^{\mathbf{m}}\left(\mathbf{f_c}(\mathbf{x^i}) - \mathbf{y^i_c}\right)^2$$

  - $\mathbf{f}_c(\mathbf{x}^i)$ depends on $\mathbf{w}$
  - $\mathbf{y}^i$ is independent of $\mathbf{w}$

- Compute partial derivatives w.r.t. $\mathbf{w}^k{}_{pj}$ for all k, p, j
- Suppose have **t** layers

$$\mathbf{f_c}(\mathbf{x^i}) = \mathbf{z^t_c} = \mathbf{h}(\mathbf{a^t_c}) = \mathbf{h}(\mathbf{z^{t-1}} \cdot \mathbf{w^t_c})$$

Training MNN: Single Sample

- For derivation, we use:

$$J_i(\mathbf{w}) = \frac{1}{2}\sum_{c=1}^{m}\left(f_c(\mathbf{x}^i) - y_c^i\right)^2$$

$$f_c(\mathbf{x}^i) = h(a_c^t) = h(\mathbf{z}^{t-1} \cdot \mathbf{w}_c^t)$$

- For weights $\mathbf{w}^t_{pj}$ to the output layer $\mathbf{t}$:

$$\frac{\partial}{\partial w_{pj}^t} J(\mathbf{w}) = \left(f_j(\mathbf{x}^i) - y_j^i\right)\frac{\partial}{\partial w_{pj}^t}\left(f_j(\mathbf{x}^i) - y_j^i\right)$$

- $$\frac{\partial}{\partial w_{pj}^t}\left(f_j(\mathbf{x}^i) - y_j^i\right) = h'(a_j^t)\, z_p^{t-1}$$

- Therefore, $$\frac{\partial}{\partial w_{pj}^t} J_i(\mathbf{w}) = \left(f_j(\mathbf{x}^i) - y_j^i\right) h'(a_j^t)\, z_p^{t-1}$$
    - both $h'(a_j^t)$ and z_p^{t-1} depend on $\mathbf{x}^i$. For simpler notation, we don't make this dependence explicit.

Training MNN: Single Sample

- For a layer **k**, compute partial derivatives w.r.t. $\mathbf{w}^k{}_{pj}$
- Gets complex, since have lots of function compositions
- Will give the rest of derivatives
- First define $\mathbf{e}^k{}_j$, the error attributed to unit **j** in layer **k**:
- For layer **t** (output): $\mathbf{e}_j^t = \left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}_j^i\right)$
- For layers **k** < **t**: $$\mathbf{e}_j^k = \sum_{c=1}^{N(k+1)} \mathbf{e}_c^{k+1}\mathbf{h}'\left(\mathbf{a}_c^{k+1}\right)\mathbf{w}_{jc}^{k+1}$$
- Thus for $2 \le \mathbf{k} \le \mathbf{t}$: $$\frac{\partial}{\partial \mathbf{w}_{pj}^k}\mathbf{J}_i(\mathbf{w}) = \mathbf{e}_j^k\,\mathbf{h}'\left(\mathbf{a}_j^k\right)\mathbf{z}_p^{k-1}$$

MNN Training: Multiple Samples

- Error on one example $\mathbf{x}^i$: $$J_i(w) = \frac{1}{2}\sum_{c=1}^{m}\left(f_c\left(x^i\right) - y_c^i\right)^2$$

$$\frac{\partial}{\partial w_{pj}^k} J_i(w) = e_j^k \, h'\left(a_j^k\right) z_p^{k-1}$$

- Error on all examples: $$J(w) = \frac{1}{2}\sum_{i=1}^{n}\sum_{c=1}^{m}\left(f_c\left(x^i\right) - y_c^i\right)^2$$

$$\frac{\partial}{\partial w_{pj}^k} J(w) = \sum_{i=1}^{n} e_j^k \, h'\left(a_j^k\right) z_p^{k-1}$$

Training Protocols

- Batch Protocol
  - true gradient descent
  - weights are updated only after all examples are processed
  - might be slow to converge
- Single Sample Protocol
  - examples are chosen randomly from the training set
  - weights are updated after every example
  - converges faster than batch, but maybe to an inferior solution
- Online Protocol
  - each example is presented only once, weights update after each example presentation
  - used if number of examples is large and does not fit in memory
  - should be avoided when possible

MNN Training: Single Sample

initialize **w** to small random numbers

choose ε, α

while $\alpha \|\nabla J(\mathbf{w})\| > \varepsilon$

for i = 1 to **n**

r = random index from {1,2,…,n}

$\mathbf{delta}_{pjk} = 0 \quad \forall\ \mathbf{p,j,k}$

$$\mathbf{e}_j^t = \left(\mathbf{f}_j\left(\mathbf{x}^r\right) - \mathbf{y}_j^r\right) \quad \forall \mathbf{j}$$

for k = t to 2

$$\mathbf{delta}_{pjk} = \mathbf{delta}_{pjk} - \mathbf{e}_j^k \mathbf{h}'\left(\mathbf{a}_j^k\right) \mathbf{z}_p^{k-1}$$

$$\mathbf{e}_j^{k-1} = \sum_{c=1}^{N(k)} \mathbf{e}_c^k \mathbf{h}'\left(\mathbf{a}_c^k\right) \mathbf{w}_{jc}^k \quad \forall \mathbf{j}$$

$$\mathbf{w}^k{}_{pj} = \mathbf{w}^k{}_{pj} + \mathbf{delta}_{pjk} \ \forall\ \mathbf{p,j,k}$$

MNN Training: Batch

initialize **w** to small random numbers

choose ε, α

while $\alpha\|\nabla J(\mathbf{w})\| > \varepsilon$

for i = 1 to **n**

$\mathbf{delta}_{pjk} = 0 \quad \forall\ \mathbf{p,j,k}$

$$\mathbf{e}_j^t = \left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}_j^i\right) \quad \forall \mathbf{j}$$

for k = t to 2

$$\mathbf{delta}_{pjk} = \mathbf{delta}_{pjk} - \mathbf{e}_j^k \mathbf{h}'\left(\mathbf{a}_j^k\right)\mathbf{z}_p^{k-1}$$

$$\mathbf{e}_j^{k-1} = \sum_{c=1}^{N(k)} \mathbf{e}_c^k \mathbf{h}'\left(\mathbf{a}_c^k\right)\mathbf{w}_{jc}^k \quad \forall \mathbf{j}$$

$\mathbf{w}^k{}_{pj} = \mathbf{w}^k{}_{pj} + \mathbf{delta}_{pjk}\ \forall\ \mathbf{p,j,k}$

BackPropagation of Errors

- In MNN terminology, training is called *backpropagation*
- errors computed (propagated) backwards from the output to the input layer

$$\textbf{while } \alpha||\nabla J(\mathbf{w})|| > \varepsilon$$

$$\textbf{for i} = 1 \text{ to } \mathbf{n}$$

$$\textbf{delta}_{pjk} = 0 \qquad \forall\ \mathbf{p,j,k}$$

$$\mathbf{e}_j^t = \left(\mathbf{y}_j^r - \mathbf{f}_j\left(\mathbf{x}^r\right)\right) \qquad \forall \mathbf{j}$$

first last layer errors computed

$$\textbf{for k} = \mathbf{t} \ \ \textbf{to} \ \ 2$$

then errors computed backwards

$$\textbf{delta}_{pjk} = \textbf{delta}_{pjk} - \mathbf{e}_j^k \mathbf{h'}\left(\mathbf{a}_j^k\right) \mathbf{z}_p^{k-1}$$

$$\mathbf{e}_j^{k-1} = \sum_{c=1}^{N(k)} \mathbf{e}_c^k \mathbf{h'}\left(\mathbf{a}_c^k\right) \mathbf{w}_{jc}^k \qquad \forall \mathbf{j}$$

$$\mathbf{w}^k{}_{pj} = \mathbf{w}^k{}_{pj} + \textbf{delta}_{pjk} \ \forall\ \mathbf{p,j,k}$$

MNN Training

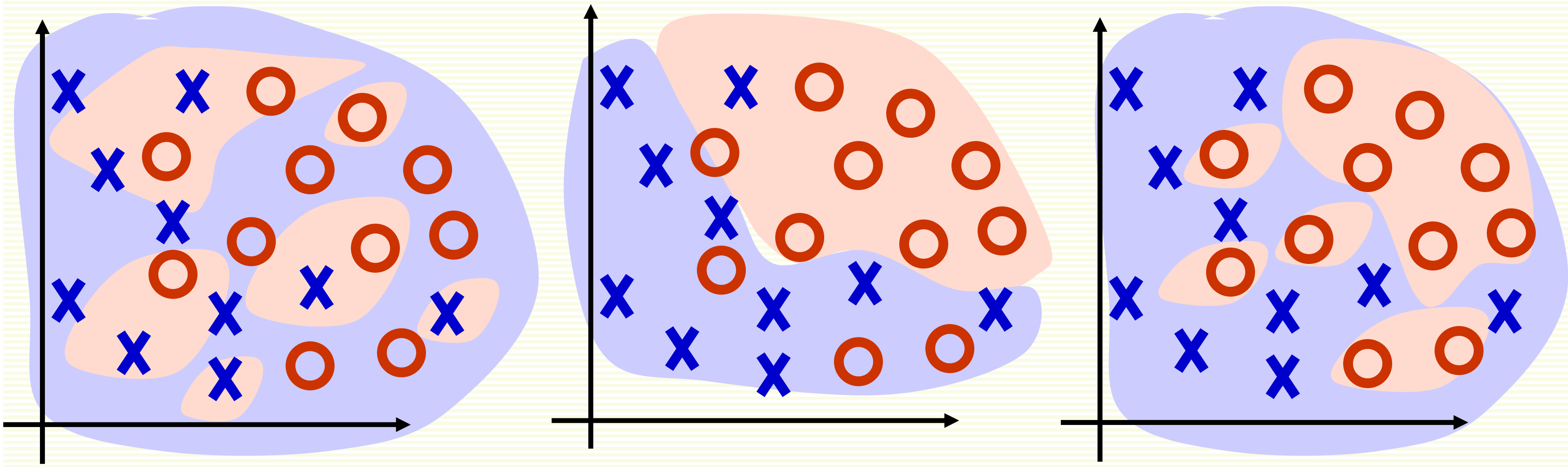
- Important: weights should be initialized to random nonzero numbers

$$\frac{\partial}{\partial \mathbf{w}_{pj}^{k}} \mathbf{J}_i(\mathbf{w}) = -\mathbf{e}_j^k \mathbf{h}'\left(\mathbf{a}_j^k\right) \mathbf{z}_p^{k-1}$$

$$\mathbf{e}_j^k = \sum_{c=1}^{N(k+1)} \mathbf{e}_c^{k+1} \mathbf{h}'\left(\mathbf{a}_c^{k+1}\right) \mathbf{w}_{jc}^{k+1}$$

- if $\mathbf{w}^k_{jc} = 0$, errors $\mathbf{e}^k_j$ are zero for layers $\mathbf{k} < \mathbf{t}$
- weights in layers $\mathbf{k} < \mathbf{t}$ will not be updated

MNN Training: How long to Train?



training time

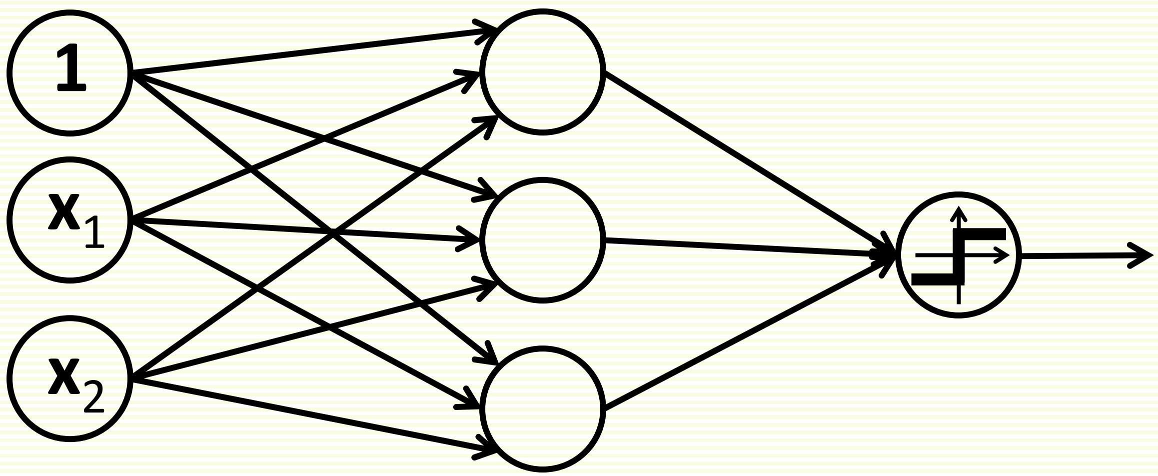
Large training error: random decision regions in the beginning - underfit

Small training error: decision regions improve with time

Zero training error: decision regions fit training data perfectly - overfit

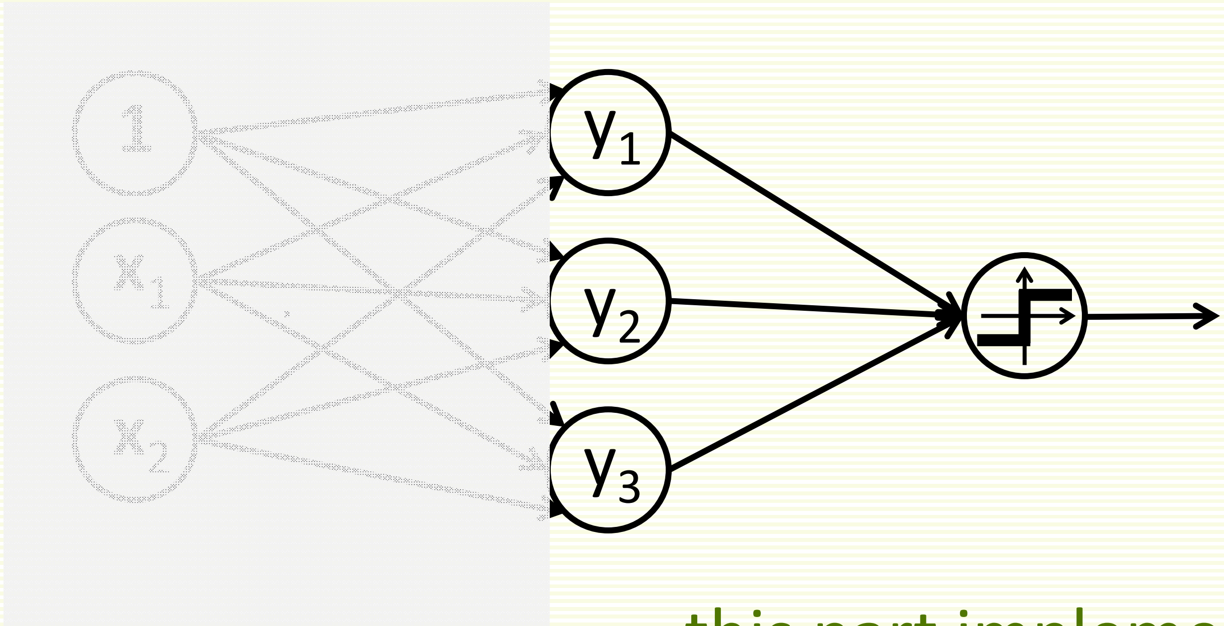
can learn when to stop training through validation

MNN as Non-Linear Feature Mapping



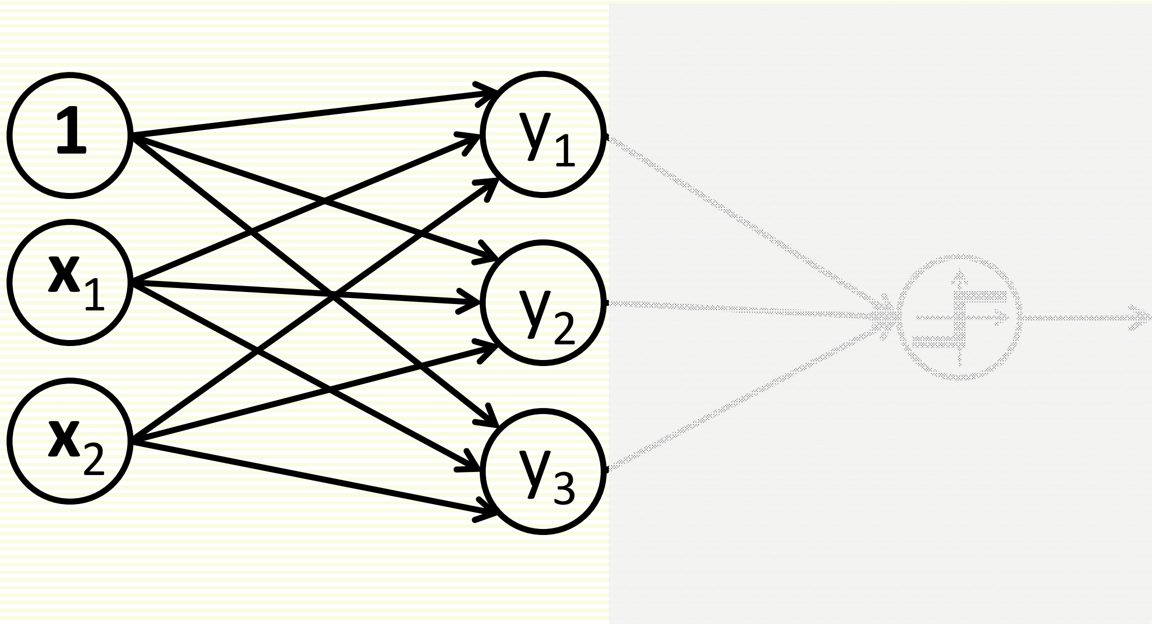
- MNN can be interpreted as first mapping input features to new features
- Then applying Perceptron (linear classifier) to the new features

MNN as Non-Linear Feature Mapping



this part implements
Perceptron (liner classifier)

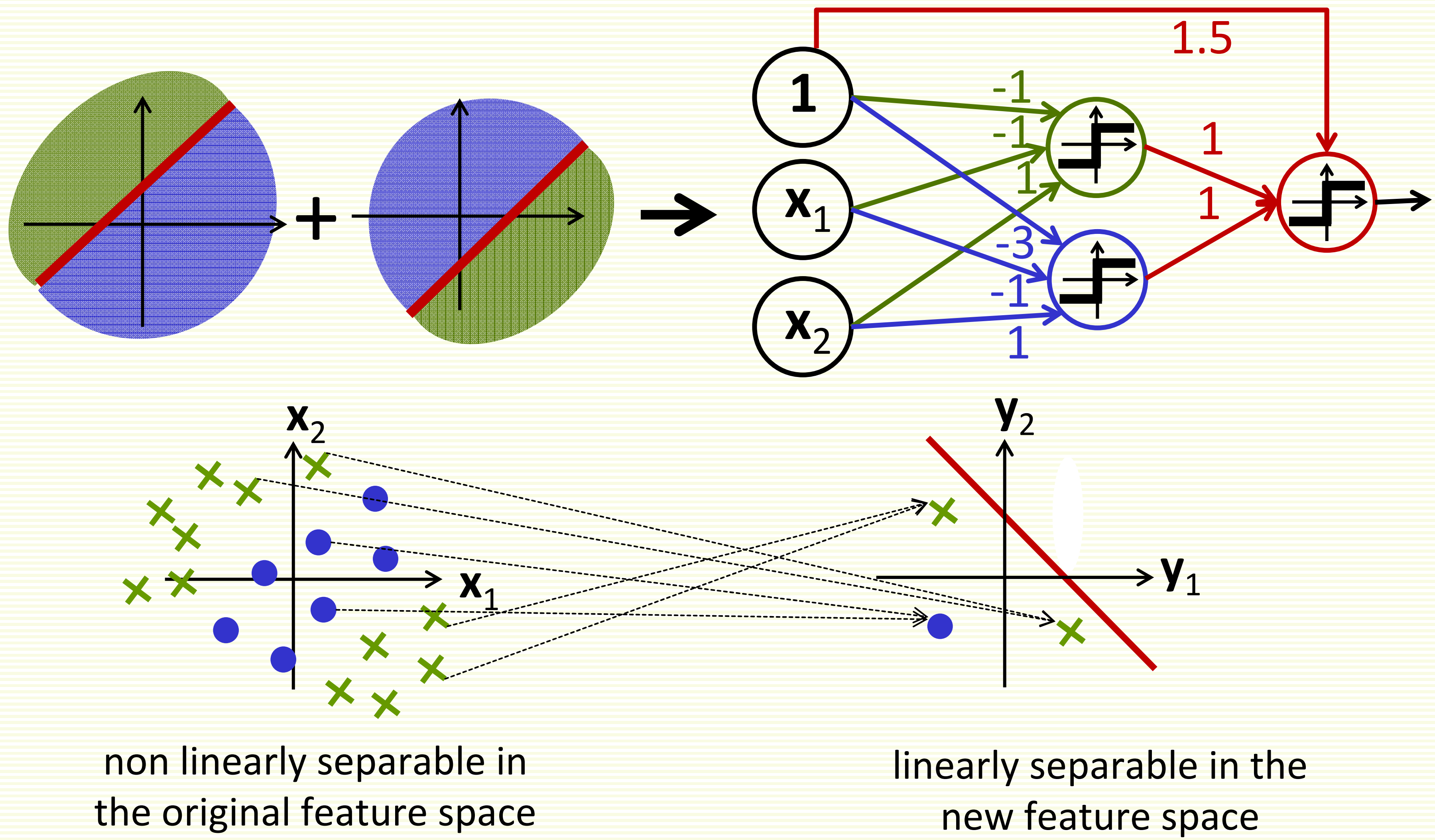
MNN as Non-Linear Feature Mapping



this part implements
mapping to new features **y**

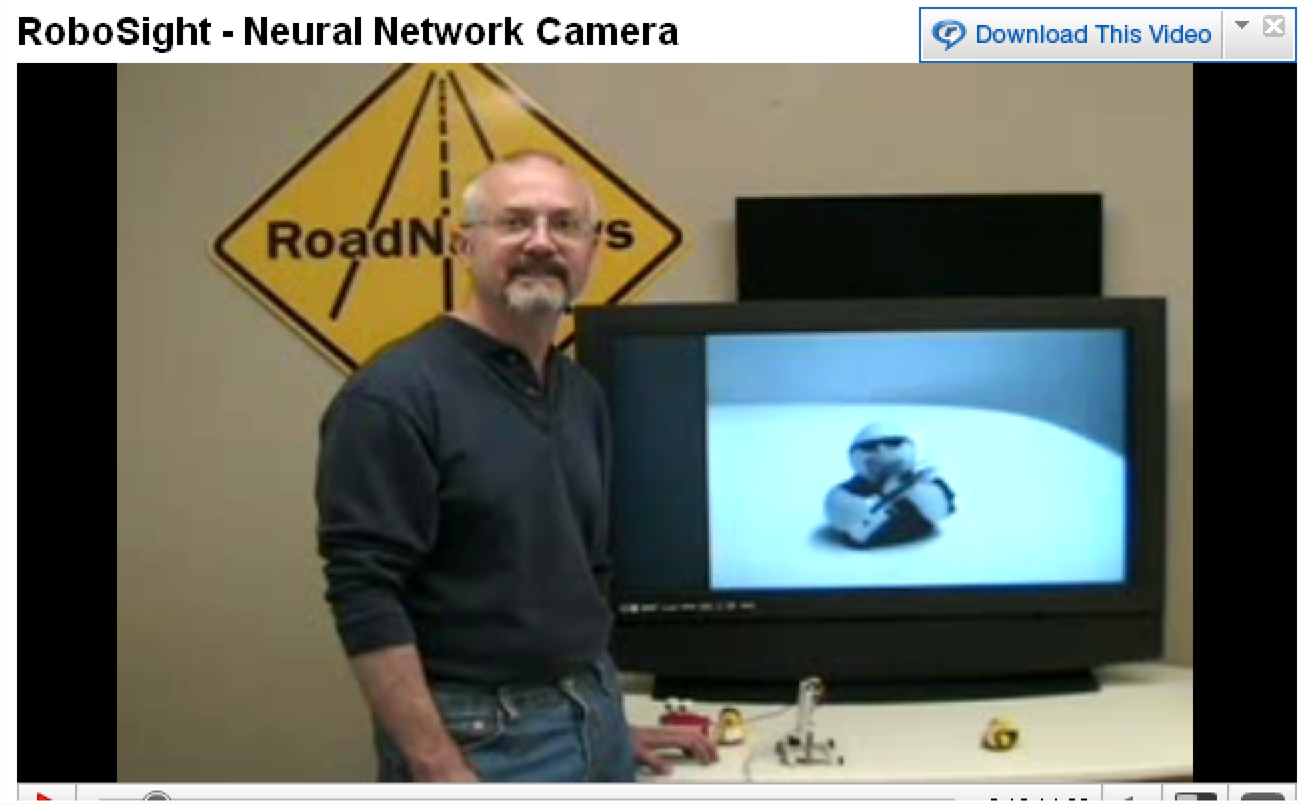
MNN as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:



Neural Network Demo

- http://www.youtube.com/watch?v=nIRGz1GEzgI



Practical Tips: Weight Decay

- To avoid overfitting, it is recommended to keep weights small
- Implement weight decay after each weight update:

$$\mathbf{w}^{new} = \mathbf{w}^{new}(1-\beta),\ 0 < \beta < 1$$

- Additional benefit is that “unused” weights grow small and may be eliminated altogether
  - a weight is “unused” if it is left almost unchanged by the backpropagation algorithm

Practical Tips for BP: Momentum

- Gradient descent finds only a local minima
- Momentum: popular method to avoid local minima and speed up descent in flat (plateau) regions
- Add temporal average direction in which weights have been moving recently
- Previous direction: $\Delta \mathbf{w}^t = \mathbf{w}^t - \mathbf{w}^{t-1}$
- Weight update rule with momentum:

$$\mathbf{w}^{\mathbf{t}+1} = \mathbf{w}^{\mathbf{t}} + (1-\beta)\underbrace{\left[\alpha \frac{\partial \mathbf{J}}{\partial \mathbf{w}}\right]}_{\text{steepest descent direction}} + \underbrace{\beta \Delta \mathbf{w}^{\mathbf{t}-1}}_{\text{previous direction}}$$

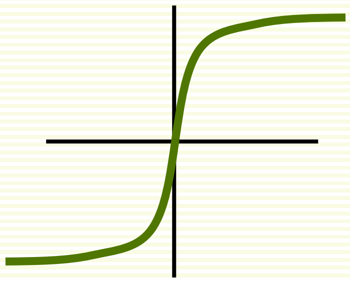
Practical Tips for BP: Activation Function

- Gradient descent works with any differentiable **h**, however some choices are better
- Desirable properties for **h**:
  - nonlinearity to express nonlinear decision boundaries
  - Saturation, that is **h** has minimum and maximum values
    - Keeps weights bounded, thus training time is reduced
  - Monotonicity so that activation function itself does not introduce additional local minima
  - Linearity for a small values, so that network can produce linear model, if data supports it
  - antisymmetry, that is **h**(-1) = -**h**(1), leads to faster learning

Practical Tips for BP: Activation Function

- Sigmoid function **h** satisfies all of the properties

$$\mathbf{h}(\mathbf{q}) = \mathbf{a}\frac{\mathbf{e}^{\mathbf{b}\cdot\mathbf{q}} - \mathbf{e}^{-\mathbf{b}\cdot\mathbf{q}}}{\mathbf{e}^{\mathbf{b}\cdot\mathbf{q}} + \mathbf{e}^{-\mathbf{b}\cdot\mathbf{q}}}$$



- Good parameter choices are **a** = 1.716, **b** = 2/3
- Asymptotic values $\pm$1.716
  - bigger than our labels, which are 1
  - If asymptotic values were smaller than 1, training error will not be small
- Linear range is roughly for –1 < **q** < 1

Practical Tips for BP: Normalization

- Features should be normalized for faster convergence
- Suppose we measure fish length in meters and weight in grams
  - Typical sample [length = 0.5, weight = 3000]
  - Feature length will be almost ignored
  - If length is in fact important, learning will be very slow
- Any normalization we looked at before (lecture on kNN) will do
  - Test samples should be normalized exactly as the training samples

Practical Tips: Initializing Weights

- Depends on the activation function
- Rule of thumb for commonly used sigmoid function
  - recall that **N**(**k**) is the number of units in layer **k**
  - for layer **k**, choose weights from the range at random

$$-\frac{1}{\sqrt{\mathbf{N}(\mathbf{k})}} < \mathbf{w}_{\mathbf{pj}}^{\mathbf{k}} < \frac{1}{\sqrt{\mathbf{N}(\mathbf{k})}}$$

Practical Tips: Learning Rate

- As any gradient descent algorithm, backpropagation depends on the learning rate α
- Rule of thumb $\alpha = 0.1$
- However can adjust α at the training time
- The objective function **J**(**w**) should decrease during gradient descent
  - If **J**(**w**) oscillates, α is too large, decrease it
  - If **J**(**w**) goes down but very slowly, α is too small, increase it

Practical Tips: Number of Hidden Layers

- Network with 1 hidden layer has the same expressive power as with several hidden layers
- Having more than 1 hidden layer may result in faster learning and less hidden units
- However, networks with more than 1 hidden layer are more prone to stuck in a local minima

Practical Tips for BP: Number of Hidden Units

- Number of hidden units determines the expressive power of the network
  - Too small may not be sufficient to learn complex decision boundaries
  - Too large may overfit the training data
- Sometimes recommended that
  - number of hidden units is larger than the number of input units
  - number of hidden units is the same in all hidden layers
- Can choose number of hidden units through validation

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", i.e. it 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 for best implementation