CS4442/9542b
Artificial Intelligence II
prof. Olga Veksler

Lecture 5

Machine Learning

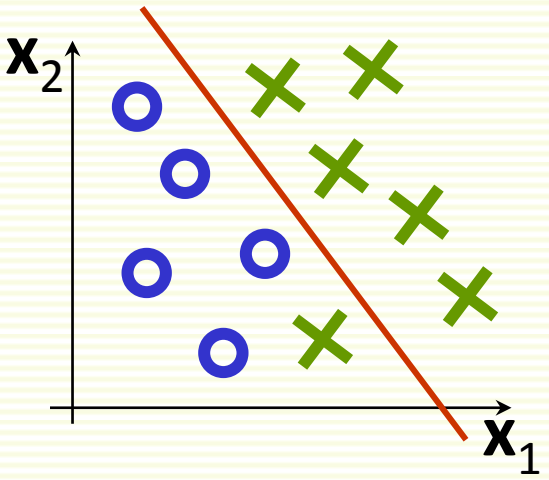
Neural Networks

Many presentation Ideas are due to Andrew NG

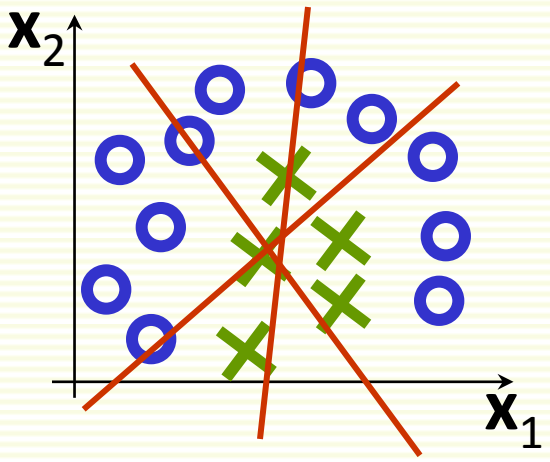
Outline

- 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



$$g(x) = w_0 + w_1 x_1 + w_2 x_2$$

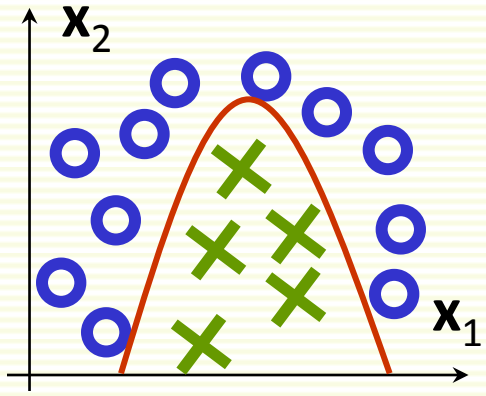


- Previous lecture studied linear discriminant
- Works for linearly (or almost) separable cases
- Many problems are far from linearly separable
  - underfitting with linear model

Need for Non-Linear Discriminant

- Can use other discriminant functions, like quadratics

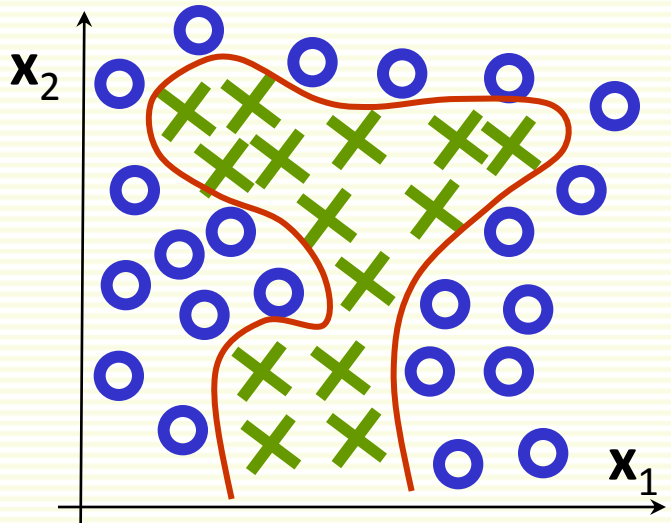
$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2$$



- Methodology is almost the same as in the linear case:
  - $f(\mathbf{x}) = \text{sign}(w_0 + w_1 x_1 + w_2 x_2 + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2)$
  - $\mathbf{z} = [\,1 \quad x_1 \quad x_2 \quad x_1 x_2 \quad x_1^2 \quad x_2^2\,]$
  - $\mathbf{a} = [\,w_0 \quad w_1 \quad w_2 \quad w_{12} \quad w_{11} \quad w_{22}\,]$
  - "normalization": multiply negative class samples by -1
  - gradient descent to minimize Perceptron objective function

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

Need for Non-Linear Discriminant



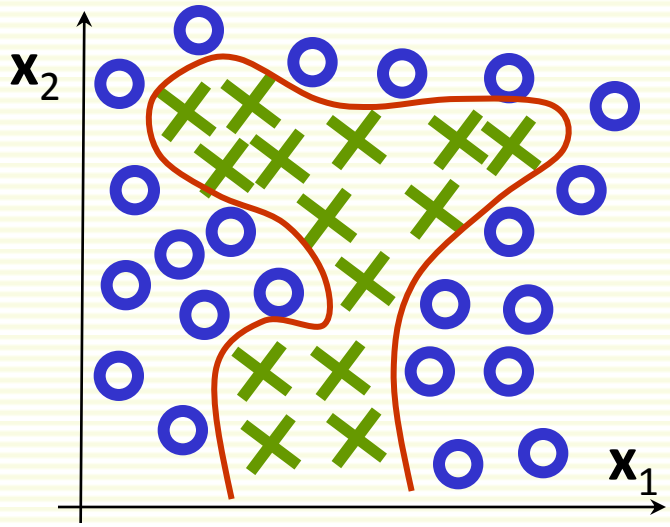
- May need highly non-linear decision boundaries
- This would require too many high order polynomial terms to fit

$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 +$$
$$+ w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2 +$$
$$+ w_{111} x_1^3 + w_{112} x_1^2 x_2 + w_{122} x_1 x_2^2 + w_{222} x_2^3 +$$
$$+ \text{even more terms of degree } \mathbf{4}$$
$$+ \text{super many terms of degree } \mathbf{k}$$

- 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

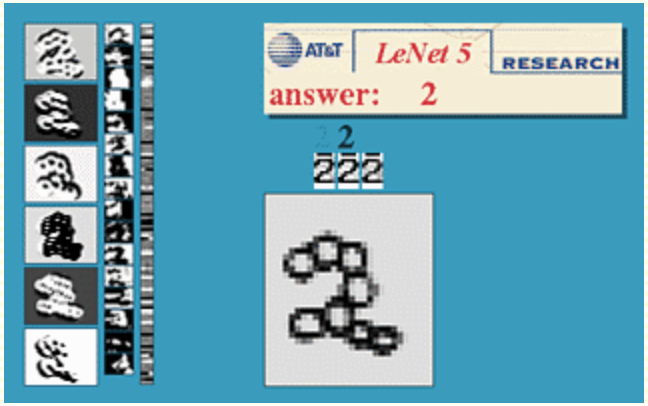
Neural Networks

- Neural Networks correspond to some discriminant function $\mathbf{g_{NN}(x)}$
- 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

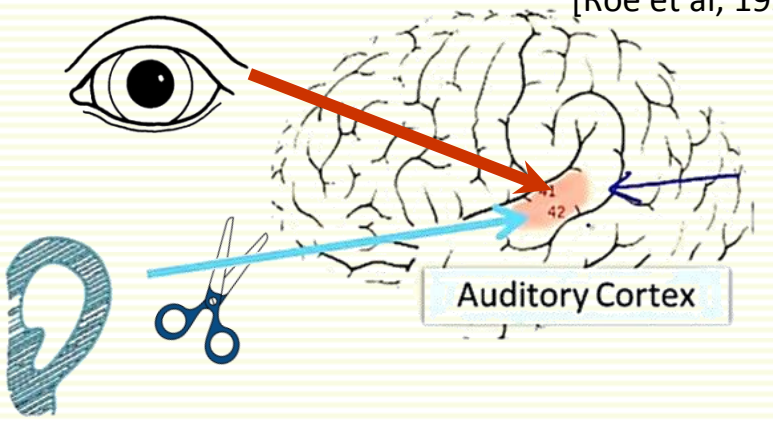
- usually one very fast processor
- high reliability
- designed to solve logic and arithmetic problems
- absolute precision
- can solve a gazillion arithmetic and logic problems in an hour

- 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

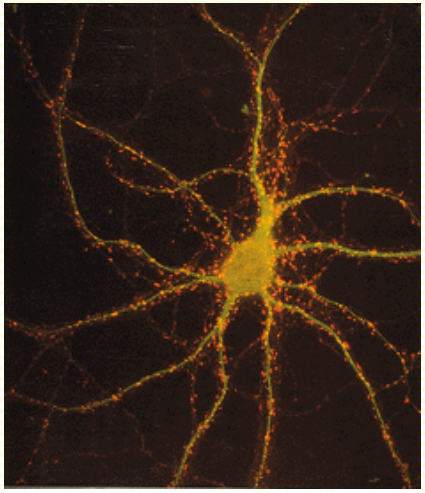
tongue array sensor

One Learning Algorithm Hypothesis

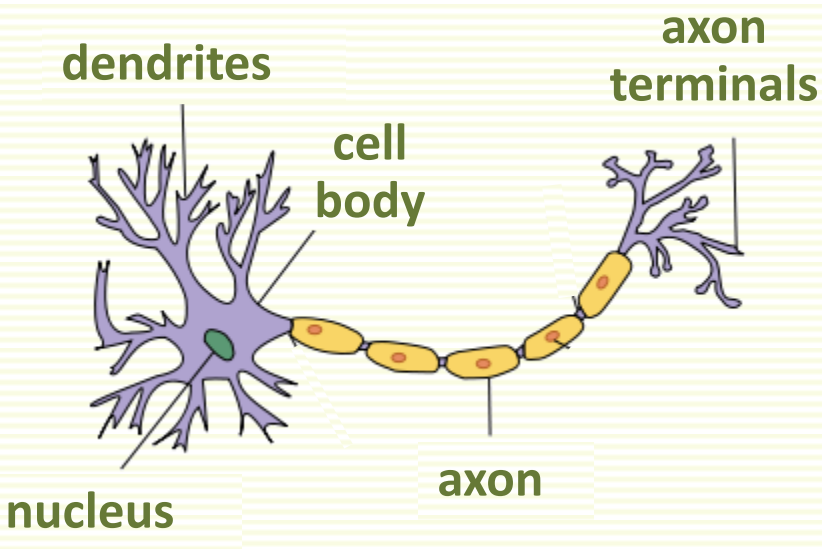
- Experimental evidence that we can plug any sensor to any part of the brain, and brain can learn how to deal with it
- Since the same physical piece of brain tissue can process sight, sound, etc.
- Maybe there is one learning algorithm can process sight, sound, etc.
- Maybe we need to figure out and 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



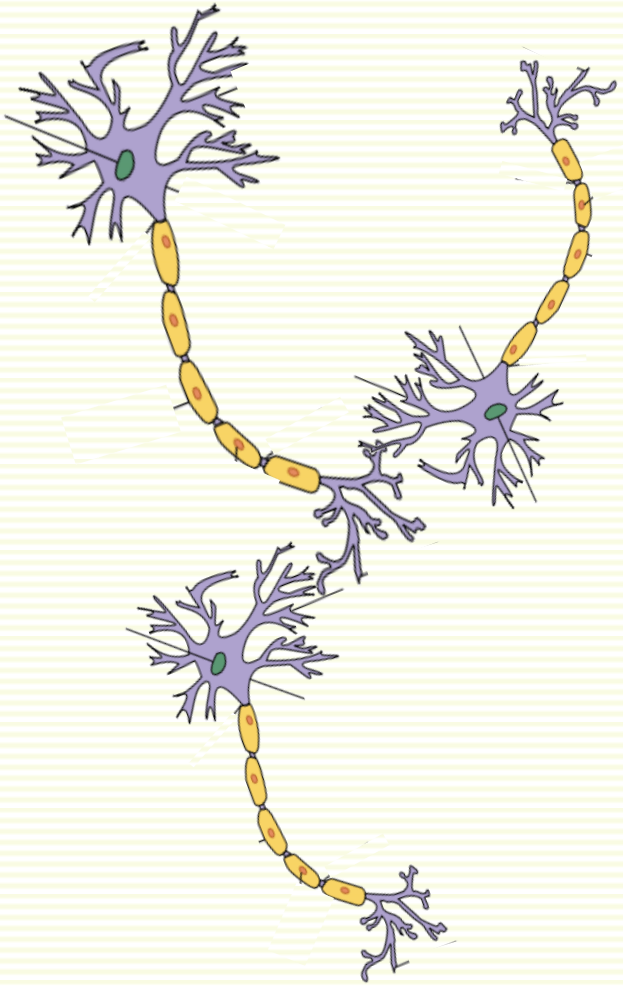
Neuron: Main Components



- **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
    - eliminates echoes in phone lines

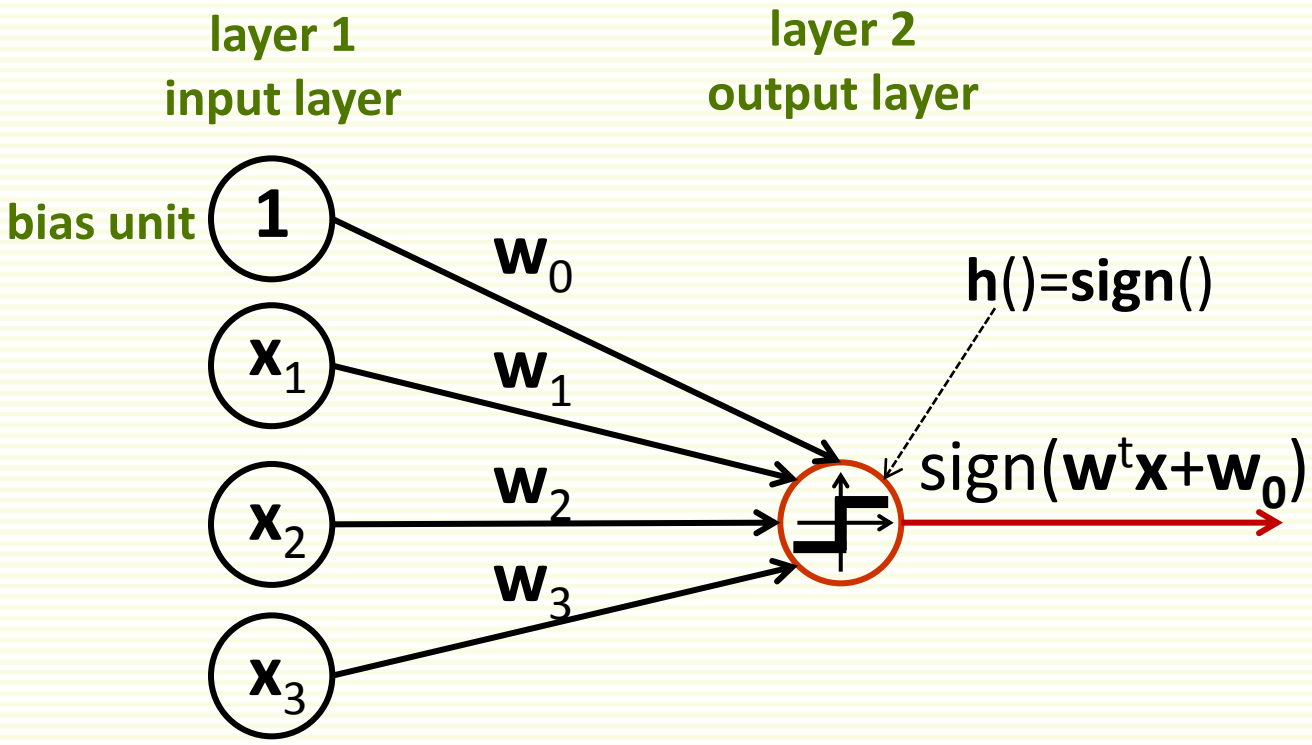
ANN History: Stagnation

- 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

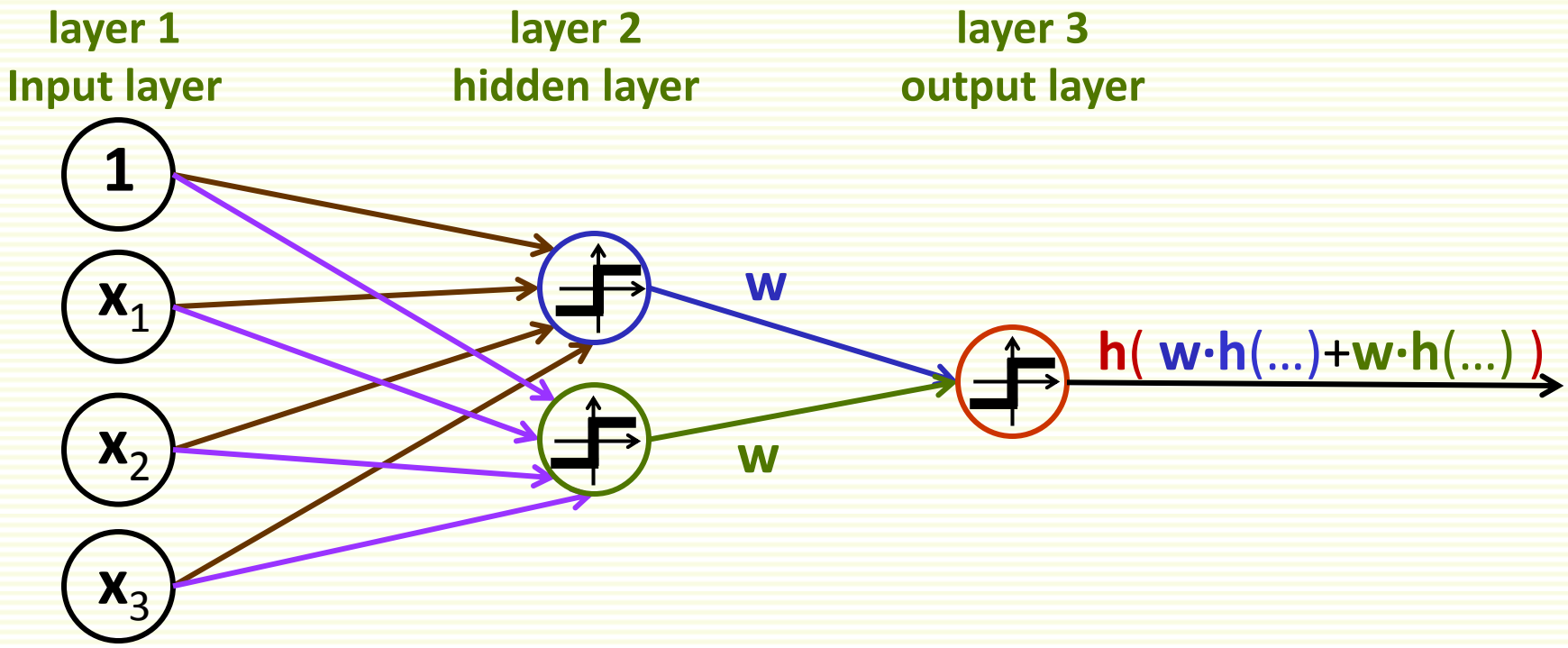
- Revival of ANN in 1980's
- 1982, J. Hopfield
  - New kind of networks (Hopfield's networks)
  - Not just model of how human brain might work, but also how to create useful devices; implements associative memory
- 1982 joint US-Japanese conference on ANN
  - US worries that it will stay behind
- Many 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
  - several successes, in particular on digit recognition, autonomous driving
- 2008-now: deep neural networks
  - Better training procedures, much larger datasets for training, GPU
  - more successes, several benchmark competitions won

Artificial Neural Nets (ANN): Perceptron



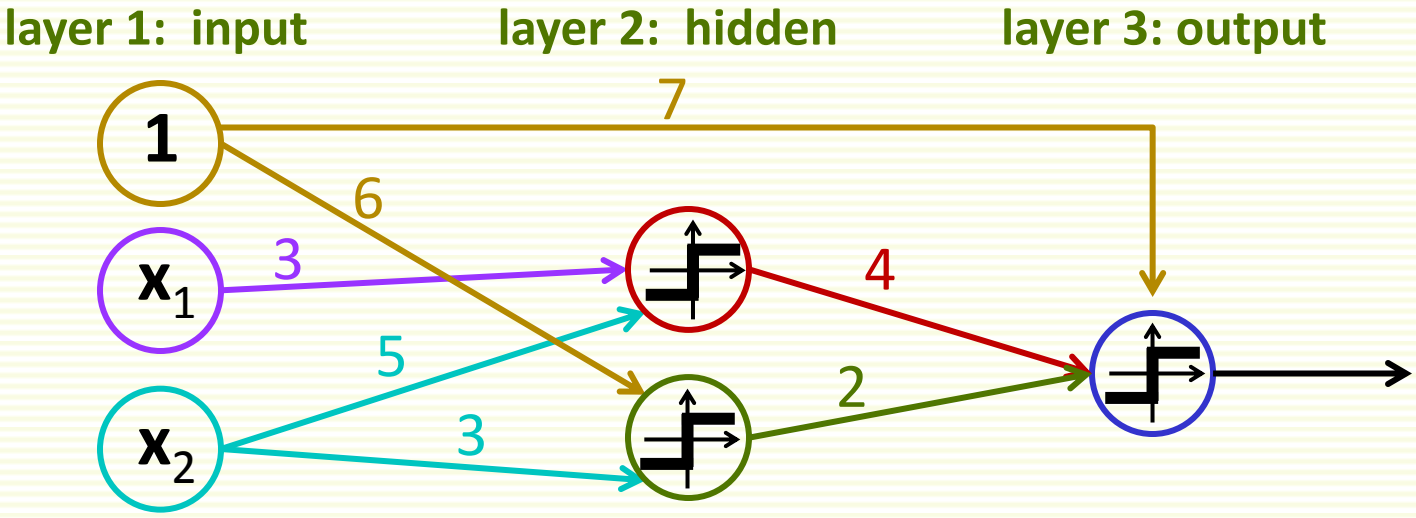
- Linear classifier **f**(**x**) = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w_0}$) is a single neuron "net"
- Input layer units output features, except bias outputs "1"
- Output layer unit applies **sign**() or some other function **h**()
- **h**() is also called an *activation function*

Multilayer Perceptron (MLP)



- First hidden unit outputs: $h(\ldots) = h(w_0+w_1x_1+w_2x_2+w_3x_3)$
- Second hidden unit outputs: $h(\ldots) = h(w_0+w_1x_1+w_2x_2+w_3x_3)$
- Network corresponds to classifier $\mathbf{f}(\mathbf{x}) = h(w\cdot h(\ldots)+w\cdot h(\ldots))$
- More complex than Perceptron, more complex boundaries

MLP Small Example



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

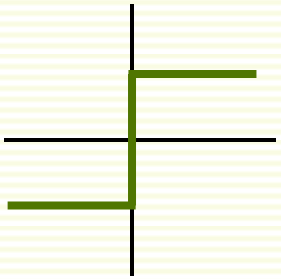
$$\mathbf{f}(\mathbf{x}) = \text{sign}(\ 4\cdot\mathbf{h}(\ldots)+2\cdot\mathbf{h}(\ldots) + 7\ )$$

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

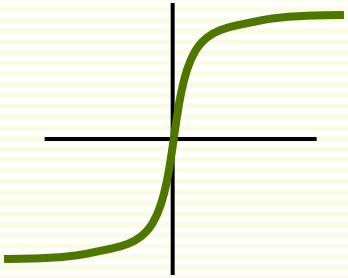
- MLP terminology: computing **f**(**x**) is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

MLP: Activation Function

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

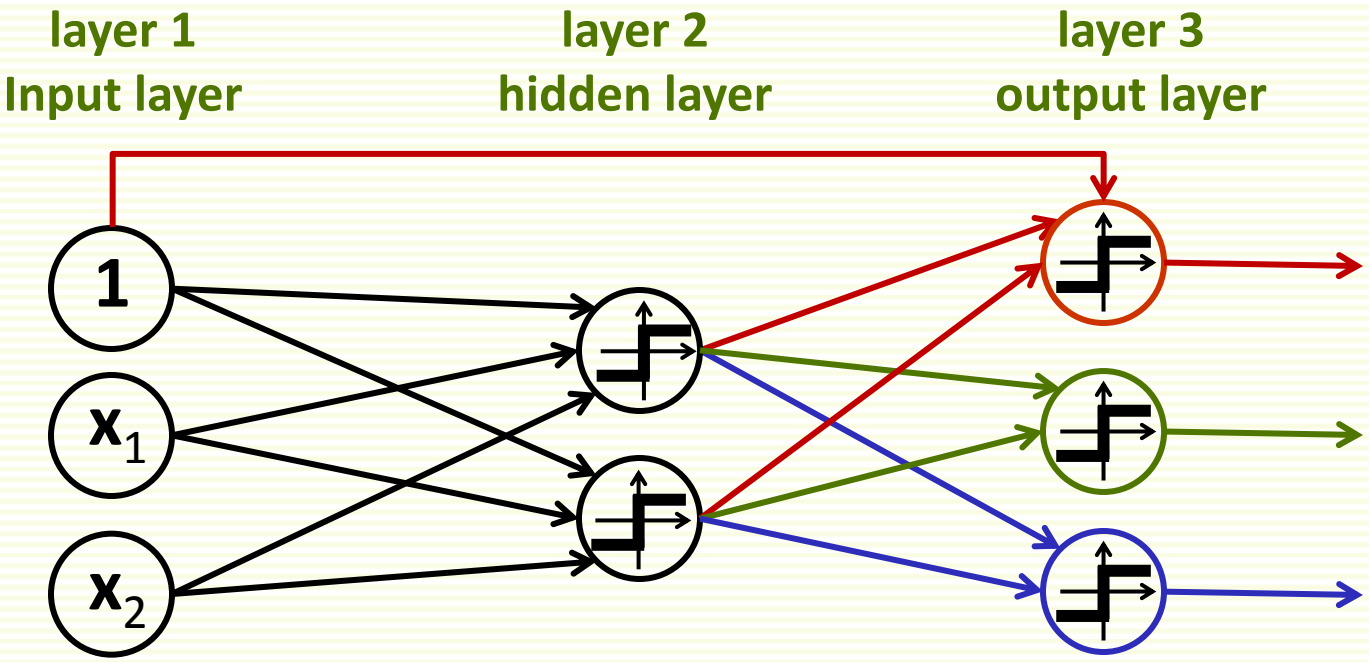


- Instead can use continuous sigmoid function



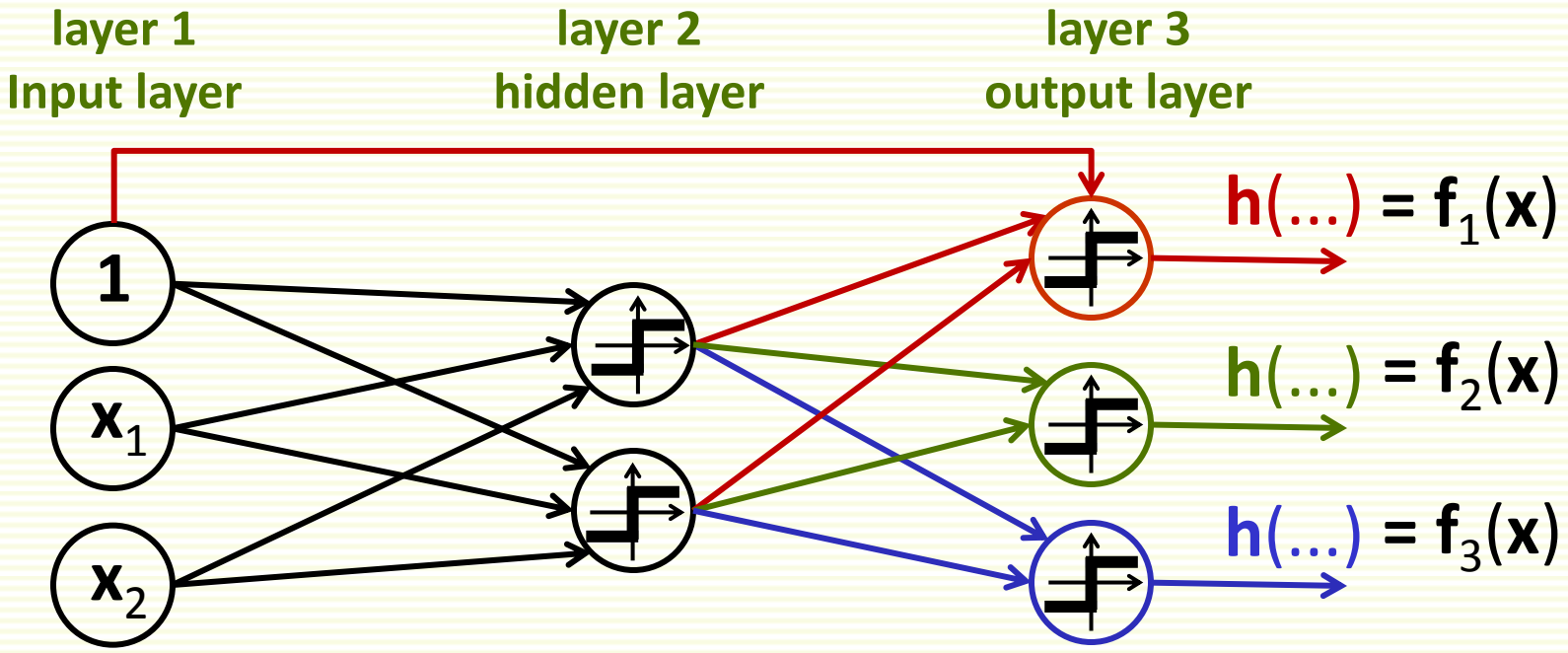
- Or another differentiable function
- Can even use different activation functions at different layers/units
- From now, assume **h**() is a differentiable function

MLP: Multiple Classes



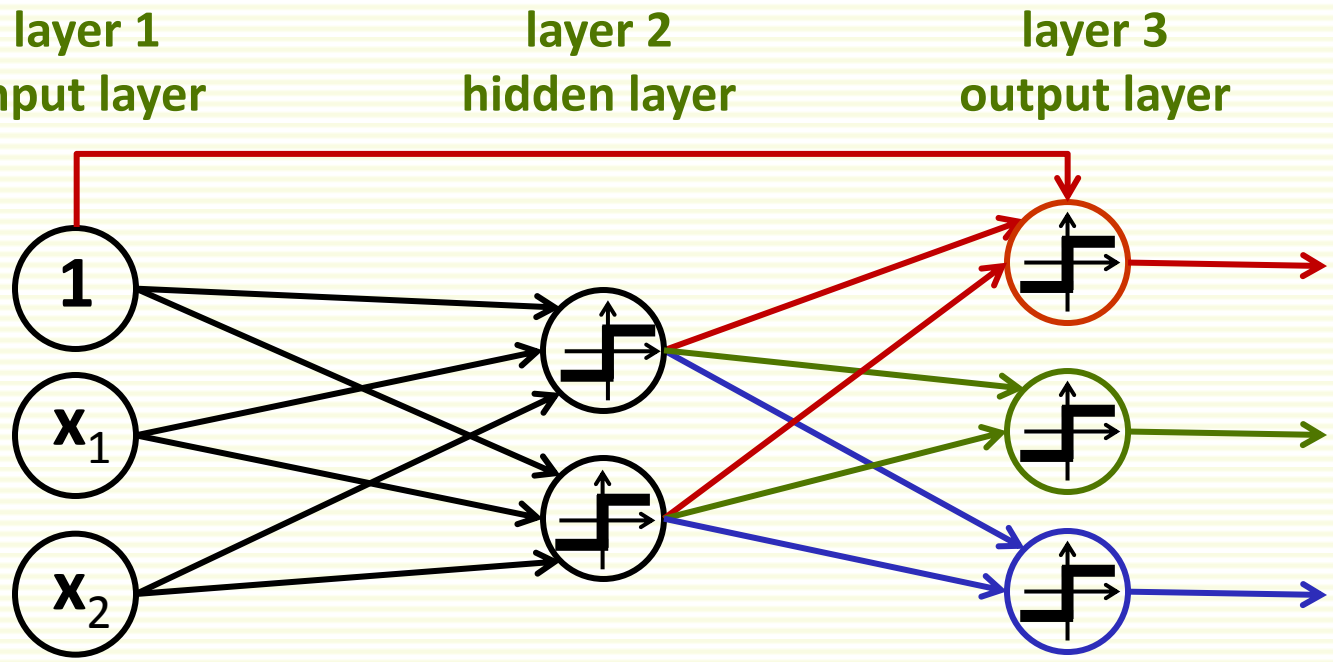
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, usually drawn in layer 1

MLP: General Structure



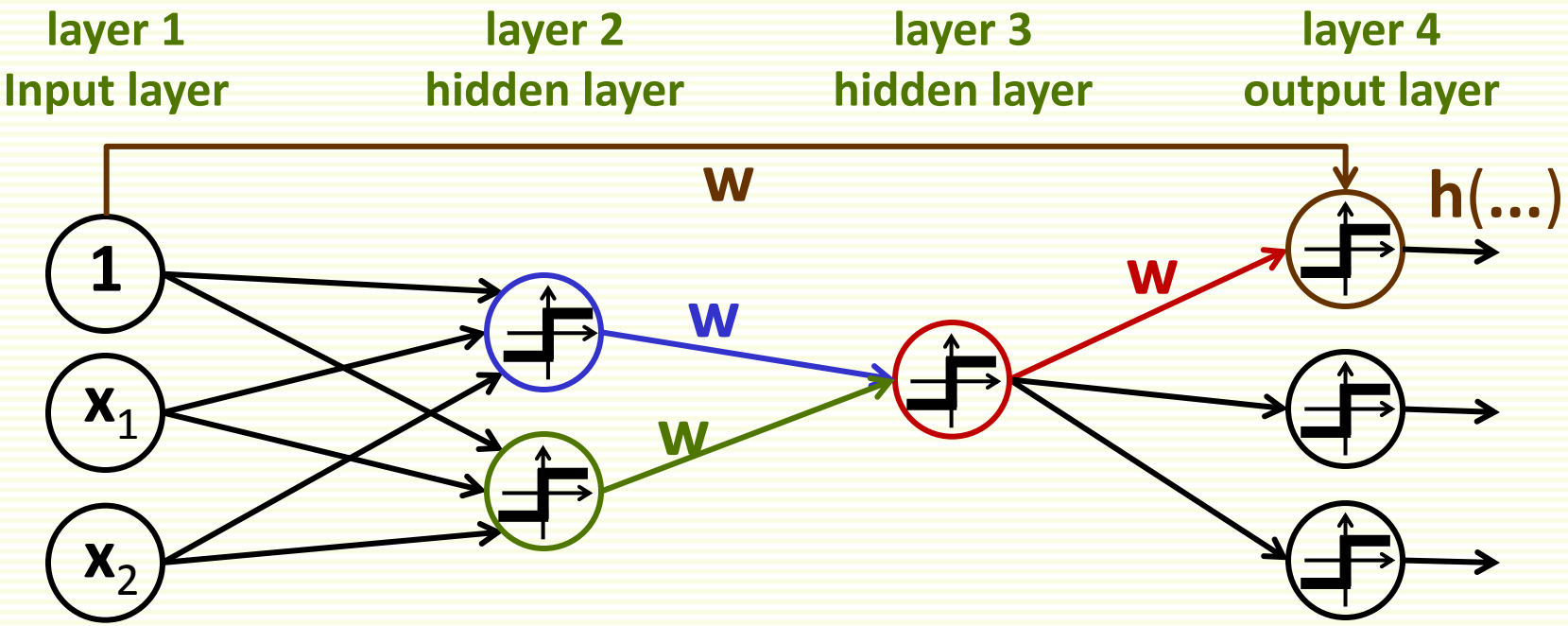
- $\mathbf{f}(\mathbf{x}) = [\mathbf{f}_1(\mathbf{x}), \mathbf{f}_2(\mathbf{x}), \mathbf{f}_3(\mathbf{x})]$ is multi-dimensional
- Classification:
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$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

MLP: General Structure



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

MLP: 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:

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

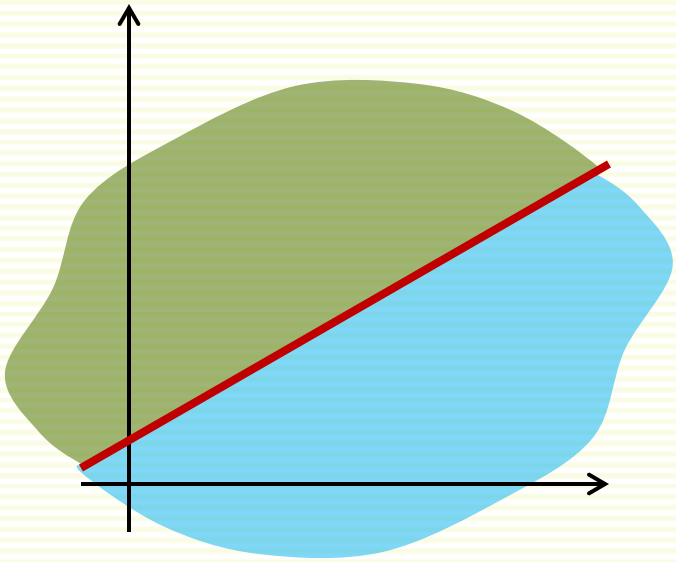
MLP: Overview

- A neural network corresponds to a classifier $\mathbf{f}(\mathbf{x},\mathbf{w})$ 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 $\mathbf{J}(\mathbf{w})$
  - optimize it with gradient descent
  - That's all!
  - Except we need quite a few slides to write down details due to complexity of $\mathbf{f}(\mathbf{x},\mathbf{w})$

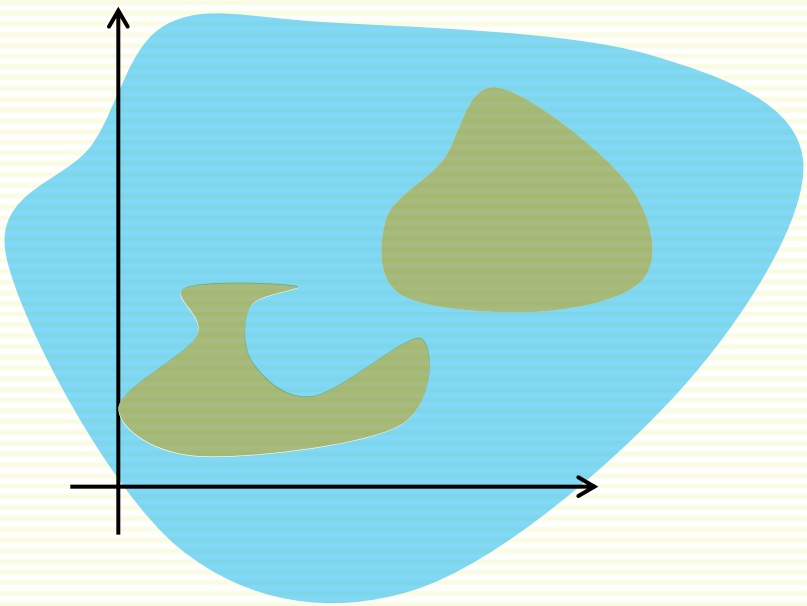
Expressive Power of MLP

- 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 MLP)
  - 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
    - MLP is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron

Decision Boundaries



- Perceptron (single layer neural net)

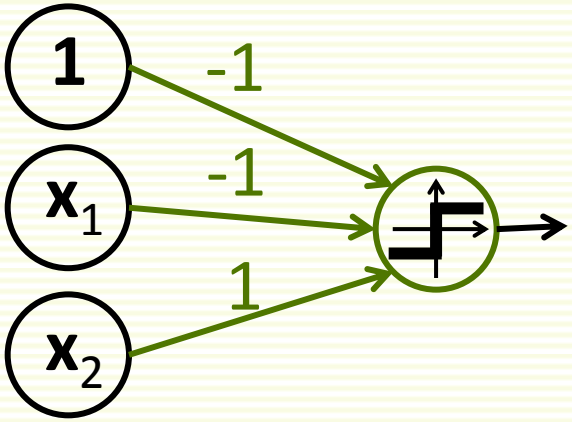


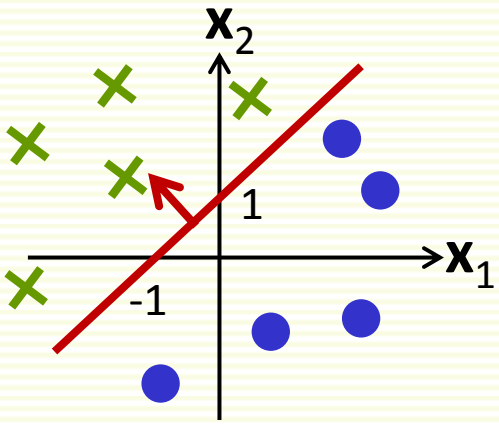
- Arbitrarily complex decision regions
- Even not contiguous

Nonlinear Decision Boundary: Example

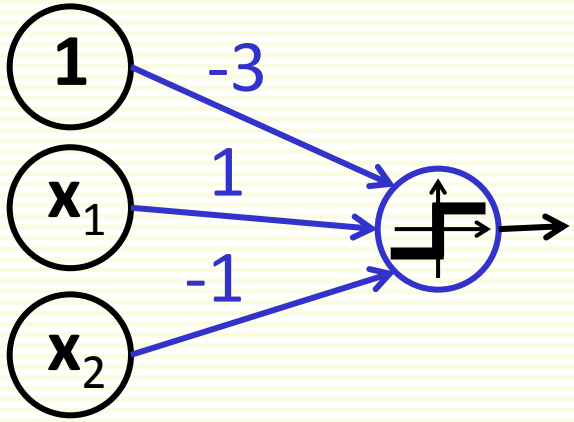
- Start with two Perceptrons, **h**() = sign()

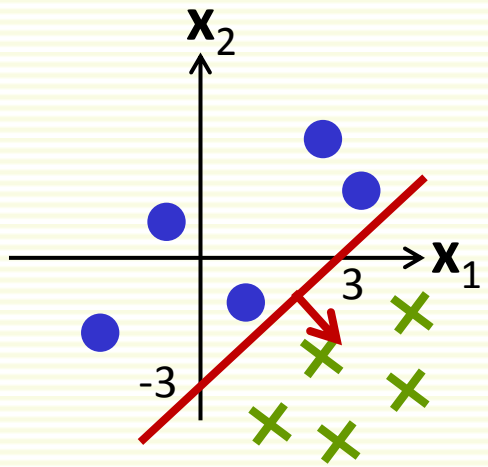
$-\mathbf{x}_1 + \mathbf{x}_2 - 1 > 0 \Rightarrow$ class 1





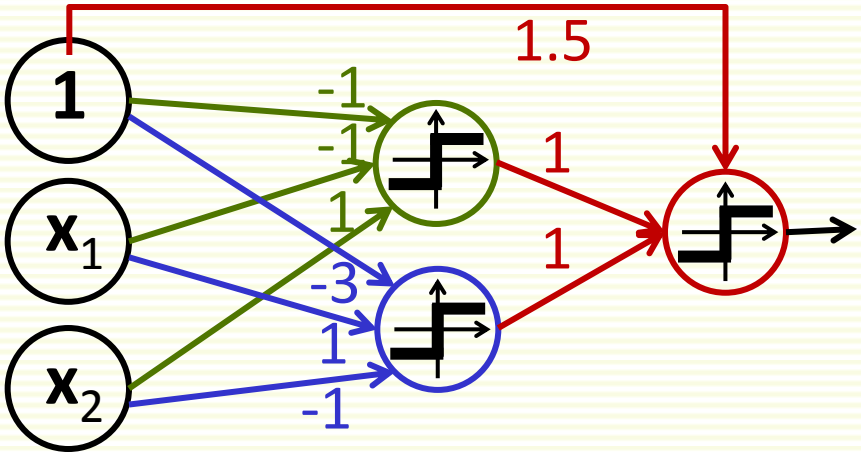
$\mathbf{x}_1 - \mathbf{x}_2 - 3 > 0 \Rightarrow$ class 1

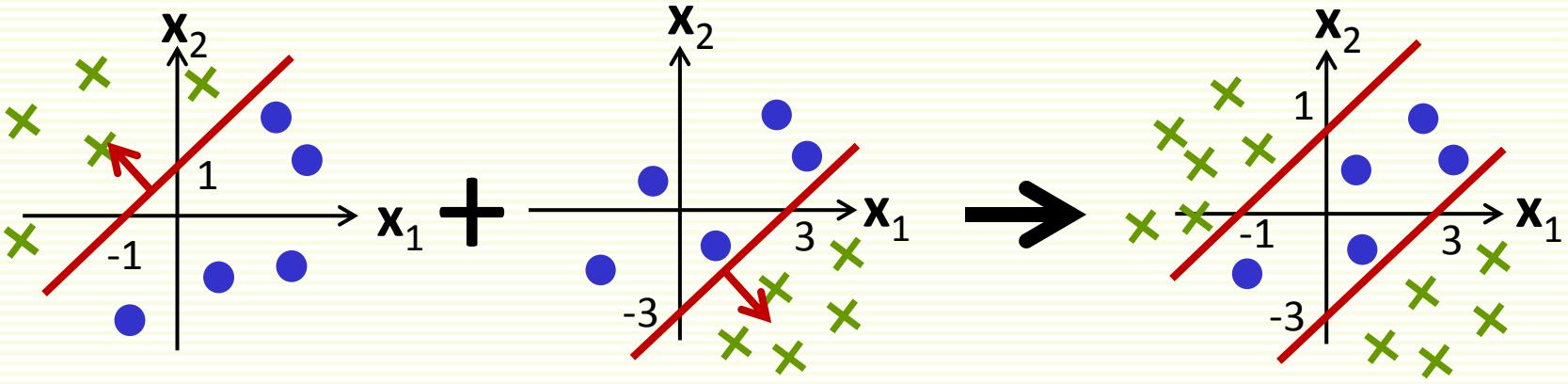




Nonlinear Decision Boundary: Example

- Now combine them into a 3 layer NN



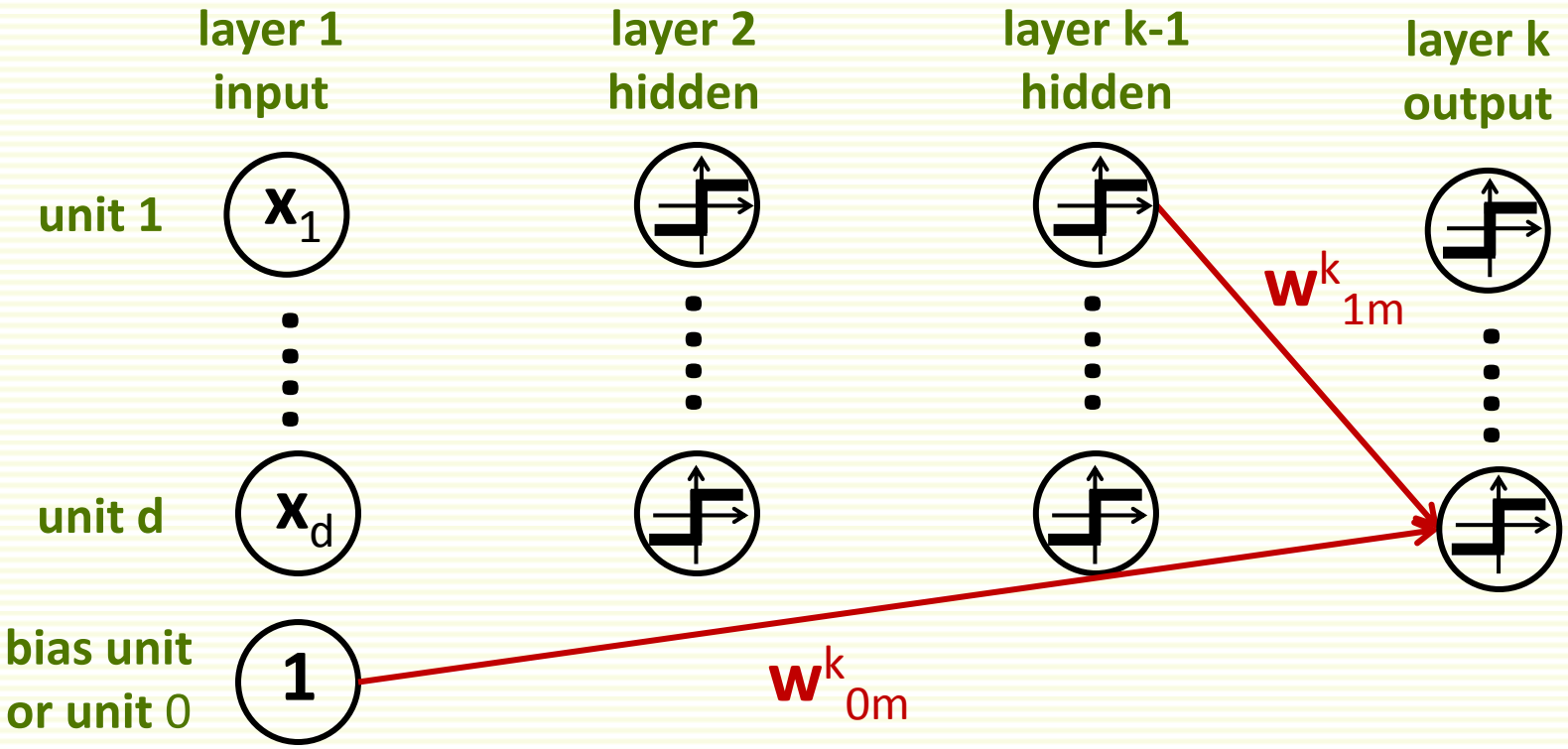


MLP: Modes of Operation

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

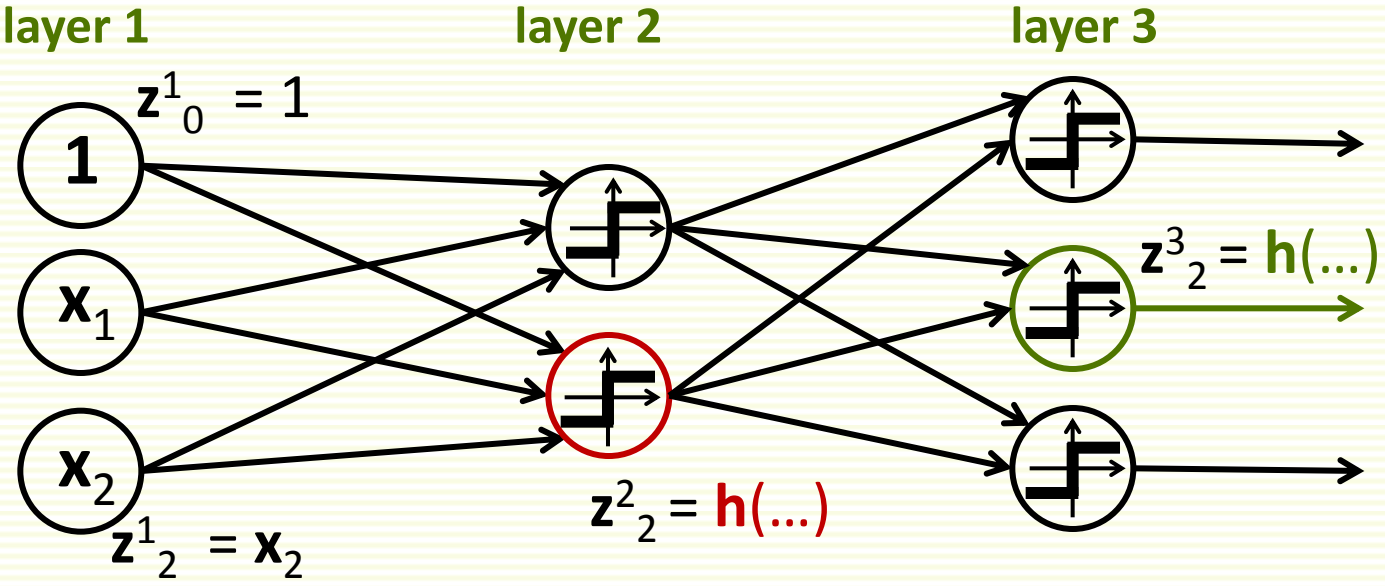
    Minimize objective function with gradient descent
  - **Feedforward (or testing)**

MLP: Notation for Edge Weights



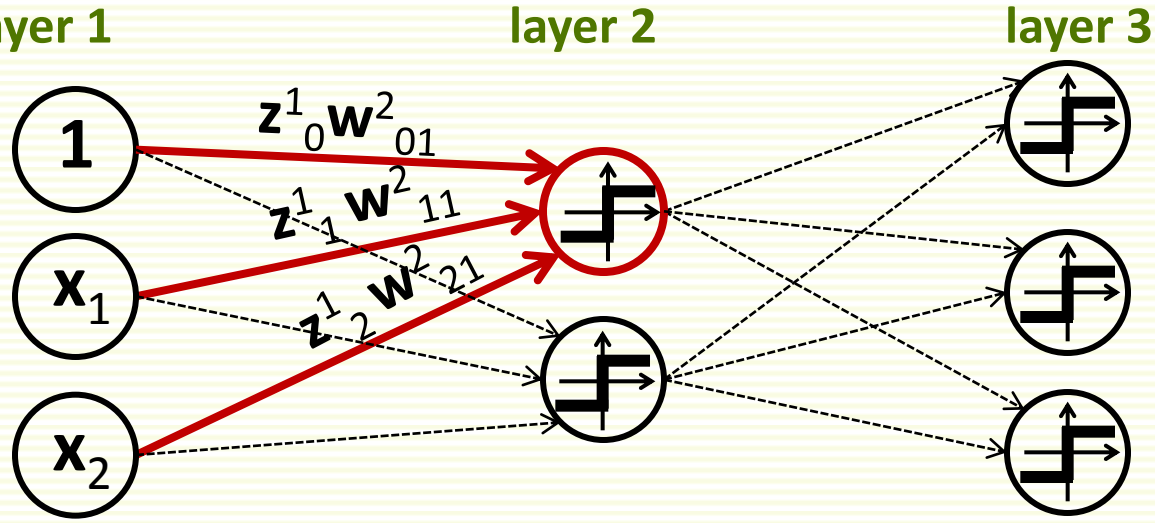
- $\mathbf{w}^k_{pj}$ is edge weight from unit **p** in layer **k**-1 to unit **j** in layer **k**
- $\mathbf{w}^k_{0j}$ is edge weight from bias unit to unit **j** in layer **k**
- $\mathbf{w}^k_{j}$ is all weights to unit **j** in layer **k**, i.e. $\mathbf{w}^k_{0j}$, $\mathbf{w}^k_{1j}$, …, $\mathbf{w}^k_{N(k-1)j}$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

MLP: More Notation



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

MLP: More Notation



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

$$a_j^k = \sum_{p=1}^{N_{k-1}} z_p^{k-1} w_{pj}^k + w_{0j}^k = \sum_{p=0}^{N_{k-1}} z_p^{k-1} w_{pj}^k = z^{k-1} \cdot w_j^k$$

$$a_1^2 = z_0^1 w_{01}^2 + z_1^1 w_{11}^2 + z_2^1 w_{21}^2$$

- For **k** > 1, $z^k_j = h(a^k_j)$

MLP: Class Representation

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

Training MLP: Objective Function

- Want to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i) = \mathbf{z}^t$
- Use squared difference
- Let **w** be all edge weights in MLP collected in one vector
- Error on one example $\mathbf{x}^i$, of class **s**

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

$$\mathbf{z}^t = \mathbf{f}(\mathbf{x}_i) = \begin{bmatrix} \mathbf{f}(\mathbf{x}_1^i) \\ \vdots \\ \mathbf{f}(\mathbf{x}_s^i) \\ \vdots \\ \mathbf{f}(\mathbf{x}_m^i) \end{bmatrix} \quad \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } s$$

Training MLP: Objective Function

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

initialize **w** to random
choose ε, α
while $\alpha||\nabla J(\mathbf{w})|| > \varepsilon$
    $\mathbf{w} = \mathbf{w} - \alpha\nabla J(\mathbf{w})$

Training MLP: Single Sample

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

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

  - $\mathbf{f}_c(\mathbf{x}^i)$ 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

$$\mathbf{f}_c\left(\mathbf{x}^i\right) = \mathbf{z}_c^t = \mathbf{h}\left(\mathbf{a}_c^t\right) = \mathbf{h}\left(\mathbf{z}^{t-1} \cdot \mathbf{w}_c^t\right)$$

Training MLP: Single Sample

$$J_i(\mathbf{w}) = \frac{1}{2}\sum_{c=1}^{m}\left(f_c(x^i) - y_c^i\right)^2 = \frac{1}{2}\left(\left(f_1(x^i) - y_1^i\right)^2 + \ldots + \left(f_p(x^i) - y_p^i\right)^2 + \ldots + \left(f_m(x^i) - y_m^i\right)^2\right)$$

$$f_p(x^i) = h(a_p^t) = h(z^{t-1} \cdot w_p^t) = h(z_0^{t-1} w_{1j}^t + \ldots + z_p^{t-1} w_{pj}^t + \ldots + z_m^{t-1} w_{mj}^t)$$

- For weights w^t_{pj} to the output layer **t**:

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

- $$\frac{\partial}{\partial w_{pj}^t}\left(f_j(x^i) - y_j^i\right) = h'(a_j^t)\, z_p^{t-1}$$
- Therefore, $$\frac{\partial}{\partial w_{pj}^t} J_i(\mathbf{w}) = \left(f_j(x^i) - y_j^i\right) h'(a_j^t)\, z_p^{t-1}$$
  - both $h'(a_j^t)$ and z_p^{t-1} depend on x^i

Training MLP: 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): $\mathbf{e}^t_j = \left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}^i_j\right)$
- For layers **k** < **t**: $\mathbf{e}^k_j = \sum_{c=1}^{N(k+1)} \mathbf{e}^{k+1}_c \mathbf{h}'\left(\mathbf{a}^{k+1}_c\right) \mathbf{w}^{k+1}_{jc}$
- And for $2 \le \mathbf{k} \le \mathbf{t}$: $\frac{\partial}{\partial \mathbf{w}^k_{pj}} \mathbf{J}_i(\mathbf{w}) = \mathbf{e}^k_j \mathbf{h}'\left(\mathbf{a}^k_j\right) \mathbf{z}^{k-1}_p$

MLP Training: Multiple Samples

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

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

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

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

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 much faster than batch
- Mini-Batch protocol
  - In between batch and single sample protocols
  - choose sets (batches) of examples
  - Update weights after each batch

MLP 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

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

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

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

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

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

for i = 1 to **n**

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

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

for k = t to 2

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

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

first last layer errors computed

then errors computed backwards

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

MLP Training

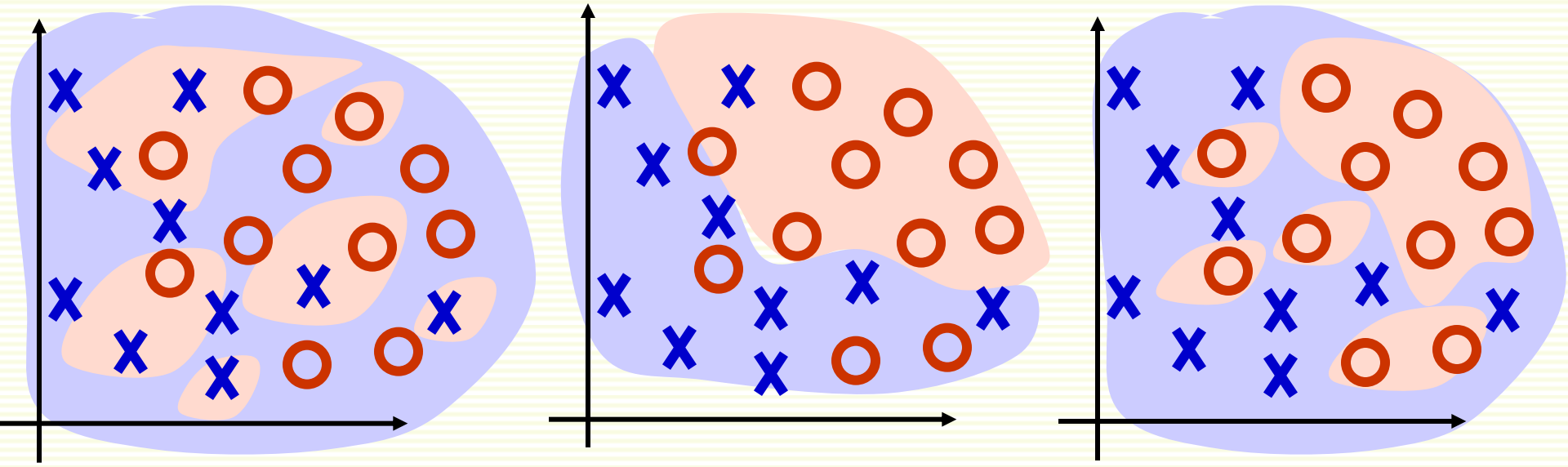
- Important: weights should be initialized to random nonzero numbers

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

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

- 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

MLP Training: How long to Train?



training time

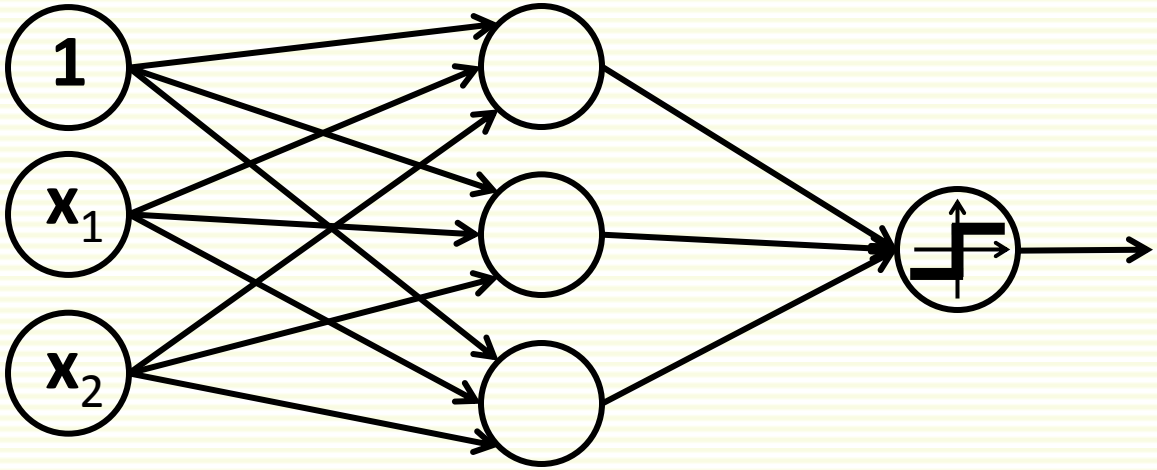
Large training error: random decision regions in the beginning - underfit

Small training error: decision regions improve with time

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

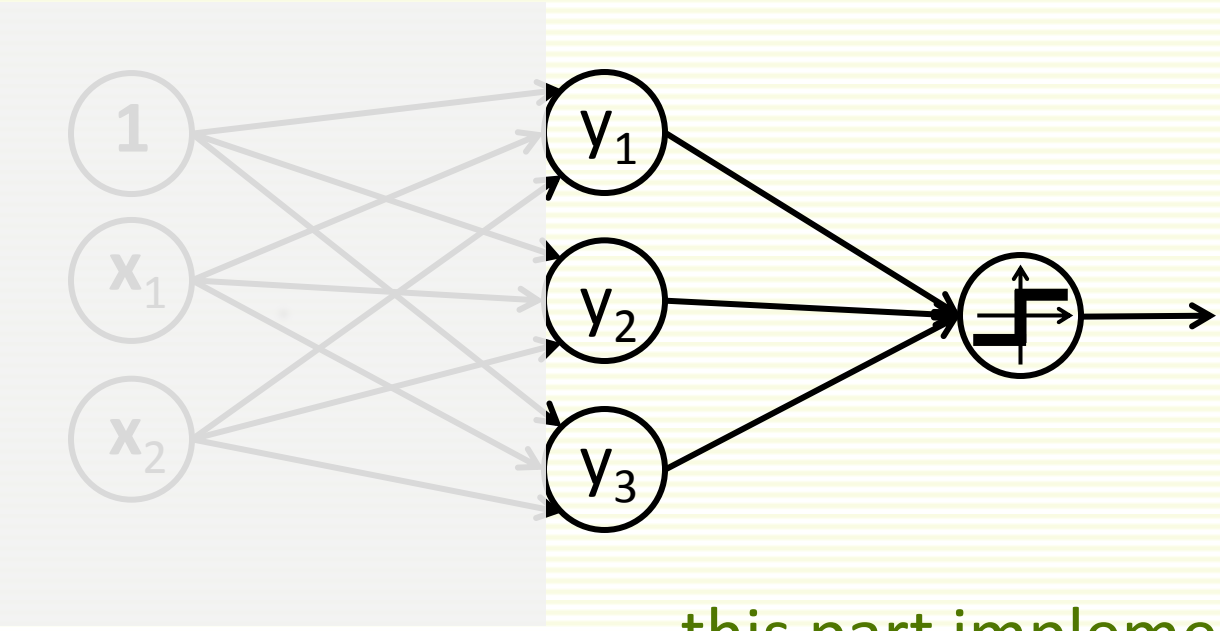
can learn when to stop training through validation

MLP as Non-Linear Feature Mapping



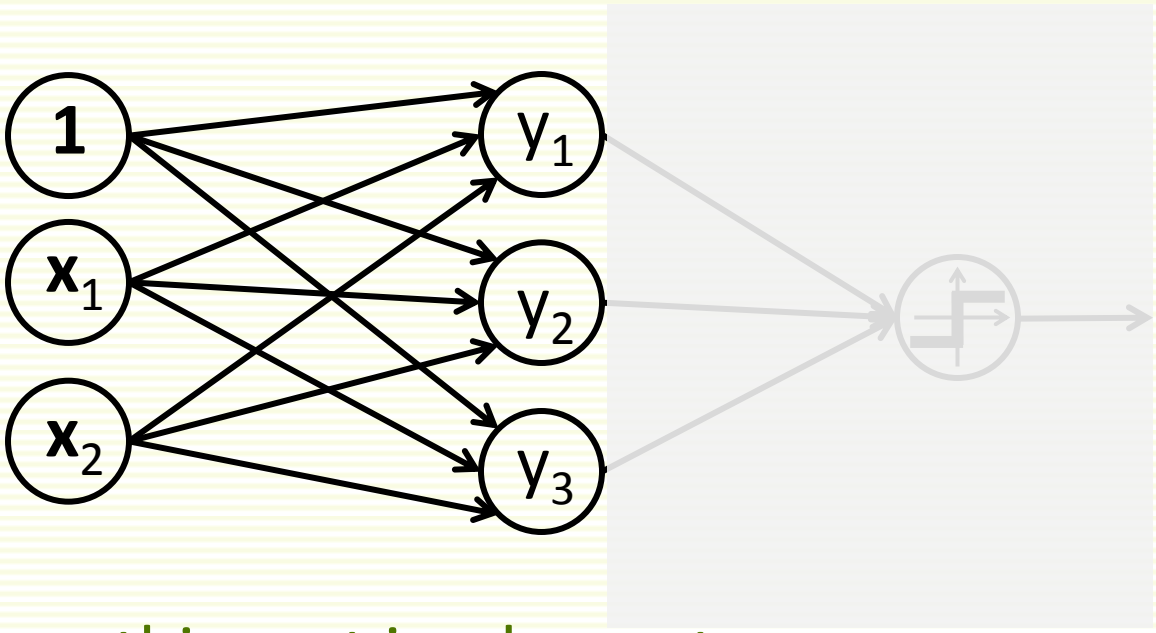
- MLP can be interpreted as first mapping input features to new features
- Then applying Perceptron (linear classifier) to the new features

MLP as Non-Linear Feature Mapping



this part implements
Perceptron (liner classifier)

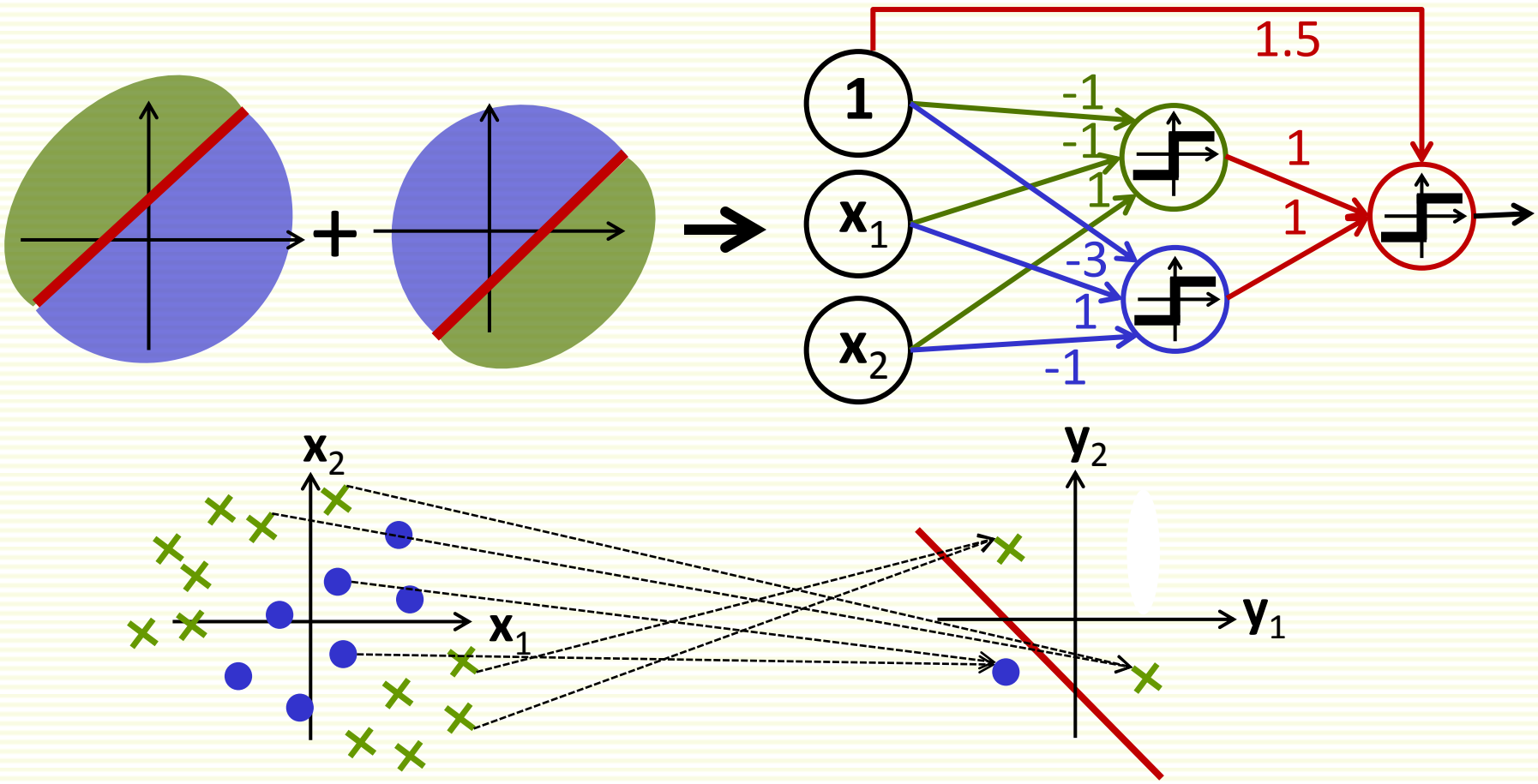
MLP as Non-Linear Feature Mapping



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

MLP as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:



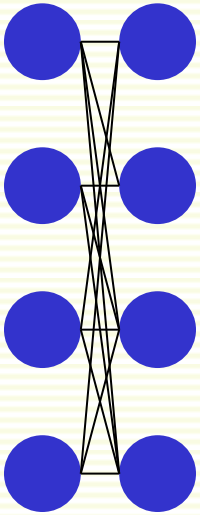
non linearly separable in the original feature space

linearly separable in the new feature space

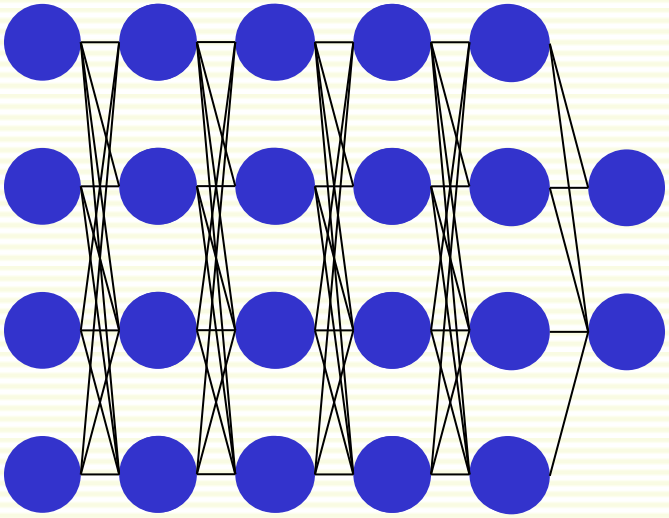
Shallow vs. Deep Architecture

- How many layers should we choose?

Shallow network



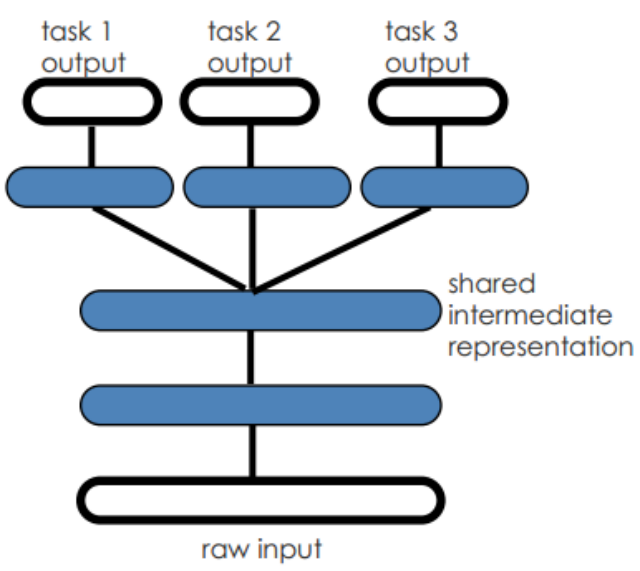
Deep network



- Deep network lead to many successful applications recently

Why Deep Networks

- 2 layer networks can represent any function
- But deep architectures are more efficient for representing some classes of functions
  - problems which can be represented with a polynomial number of nodes with *k* layers, may require an exponential number of nodes with *k*-1 layers
  - thus with deep architecture, less units might be needed overall
    - less weights, less parameter updates
  - maybe especially in image processing, with structure being mainly local
- Sub-features created in deep architecture can potentially be shared between multiple tasks



Training Deep Networks

- Difficulties of supervised training of deep networks
  - Early layers of MLN do not get trained well
    - Diffusion of Gradient – error attenuates as it propagates to earlier layers
    - Exacerbated since top couple layers can usually learn any task "pretty well" and thus the error to earlier layers drops quickly as the top layers "mostly" solve the task– lower layers never get the opportunity to use their capacity to improve results, they just do a random feature map
    - Need a way for early layers to do effective work
  - Often not enough labeled data available while there may be lots of unlabeled data
    - Can we use unsupervised/semi-supervised approaches to take advantage of the unlabeled data
  - Deep networks tend to have more local minima problems than shallow networks during supervised training

Greedy Layer-Wise Training

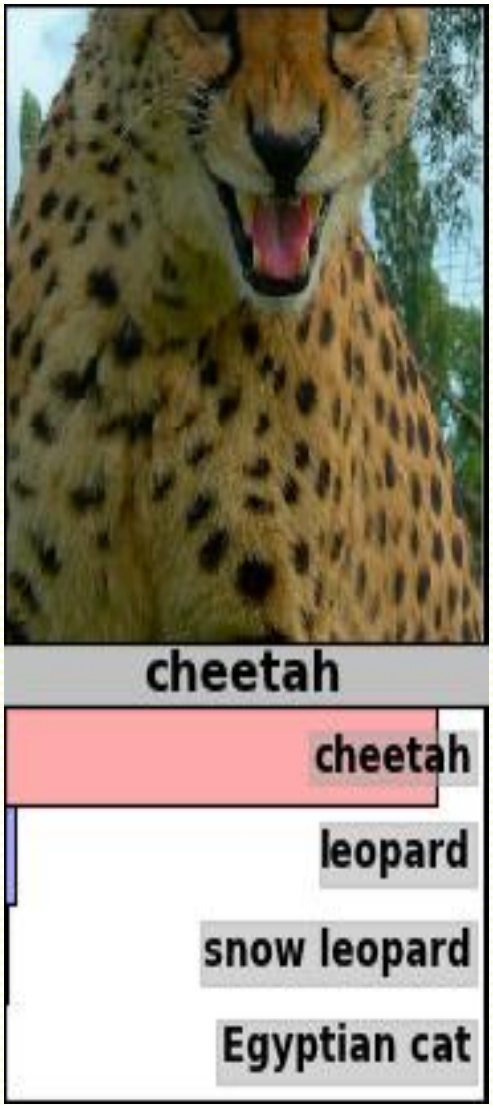
- Greedy layer-wise training to insure lower layers learn

1. Train first layer using your data without the labels (unsupervised)
   - we do not know targets at this level anyway
   - can use the more abundant unlabeled data which is not part of the training set
2. Freeze the first layer parameters and start training the second layer using the output of the first layer as the unsupervised input to the second layer
3. Repeat this for as many layers as desired
   - This builds our set of robust features
4. Use the outputs of the final layer as inputs to a supervised layer/model and train the last supervised layer(s)
   - leave early weights frozen
5. Unfreeze all weights and fine tune the full network by training with a supervised approach, given the pre-processed weight settings

Greedy Layer-Wise Training

- Greedy layer-wise training avoids many of the problems of trying to train a deep net in a supervised fashion
  - Each layer gets full learning focus in its turn since it is the only current "top" layer
  - Can take advantage of the unlabeled data
  - When you finally tune the entire network with supervised training the network weights have already been adjusted so that you are in a good error basin and just need fine tuning This helps with problems of
    - Ineffective early layer learning
    - Deep network local minima

ConvNet on Image Classification



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:

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

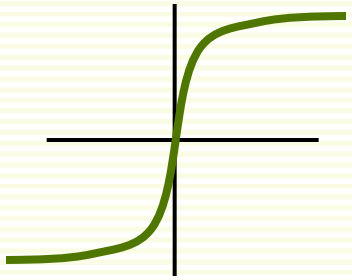
Practical Tips for BP: Activation Function

- 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 ± 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 $-1 < \mathbf{q} < 1$
- Rectified linear is popular recently: **h**(q) = max(0,**q**)

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

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

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 $\mathbf{J}(\mathbf{w})$ should decrease during gradient descent
  - If $\mathbf{J}(\mathbf{w})$ oscillates, α is too large, decrease it
  - If $\mathbf{J}(\mathbf{w})$ goes down but very slowly, α is too small, increase it

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
  - MLP can learn complex mappings from inputs to outputs, based only on the training samples
  - Easy to incorporate a lot of heuristics
  - Many competitions won recently
- Disadvantages
  - May be 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 successful implementation