CS4442/9542b
Artificial Intelligence II
prof. Olga Veksler

Lecture 6

Machine Learning

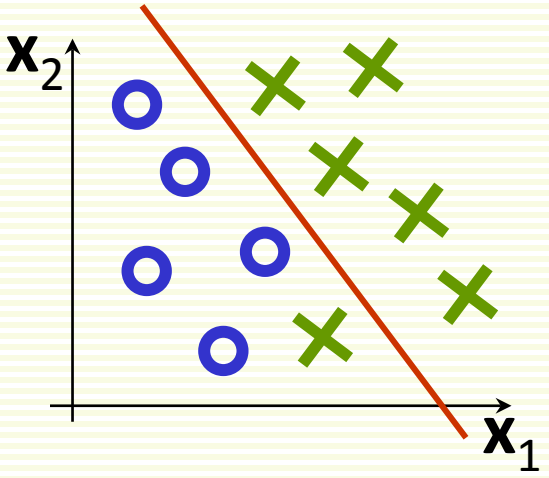
Neural Networks

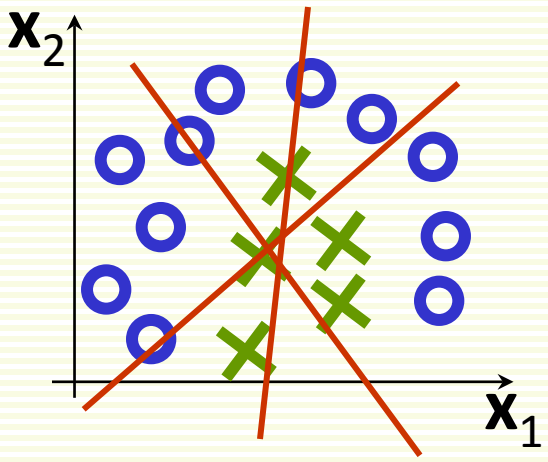
Many presentation Ideas are due to Andrew NG

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





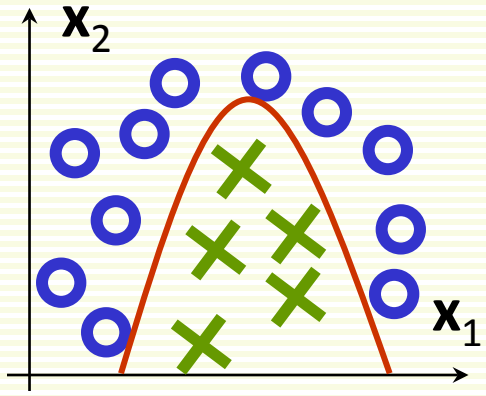
$$g(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2$$

- 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

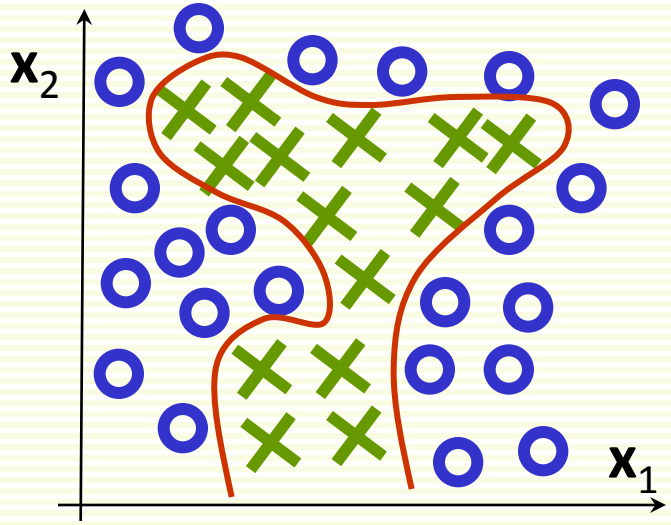
$$g(x) = w_0 + w_1 x_1 + w_2 x_2 + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2$$



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

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

Need for Non-Linear Discriminant



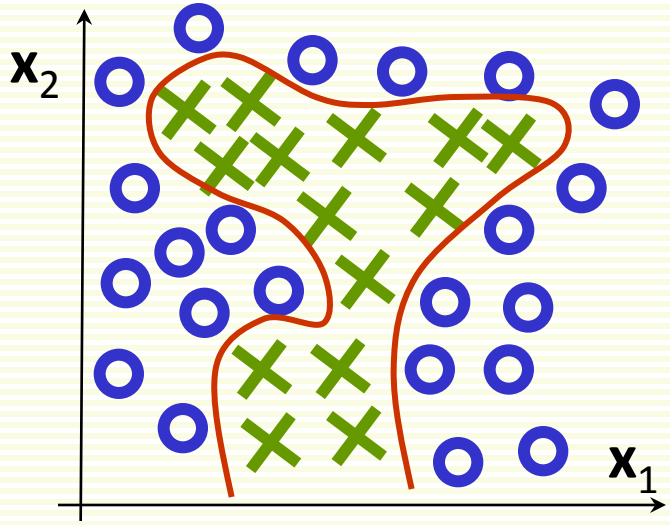
- May need highly non-linear decision boundaries
- This would require too many high order polynomial terms to fit

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

- 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^{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



Brain vs. Computer

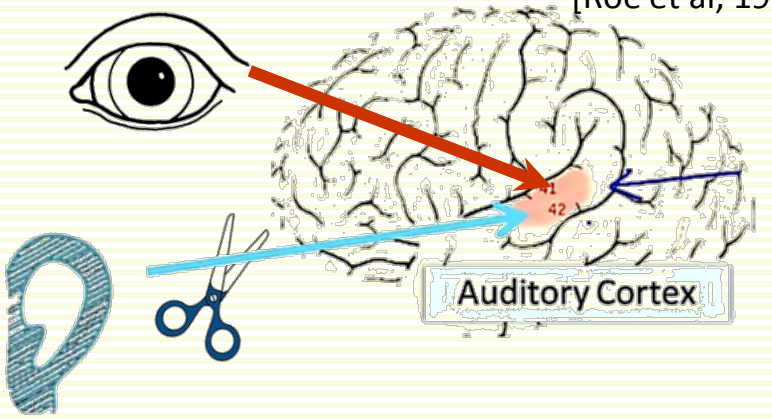
- usually one very fast processor
- high reliability
- designed to solve logic and arithmetic problems
- absolute precision
- can solve a gazillion arithmetic and logic problems in an hour

- 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

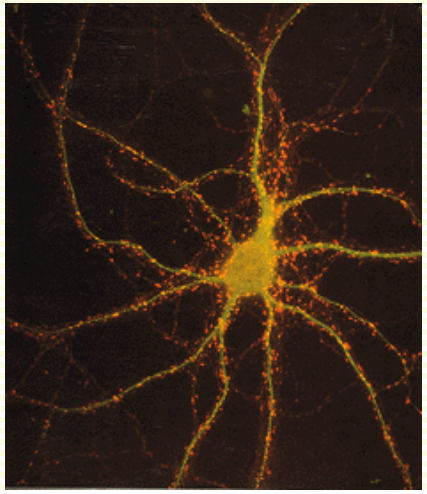
tongue array sensor

One Learning Algorithm Hypothesis

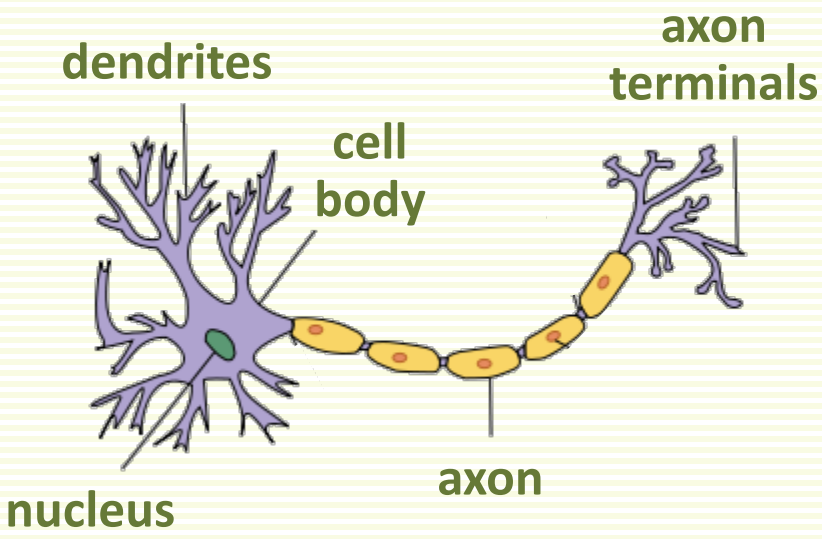
- Experimental evidence that we can plug any sensor to any part of the brain, and brain can learn how to deal with it
- Since the same physical piece of brain tissue can process sight, sound, etc.
- Maybe there is one learning algorithm can process sight, sound, etc.
- Maybe we need to figure out and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



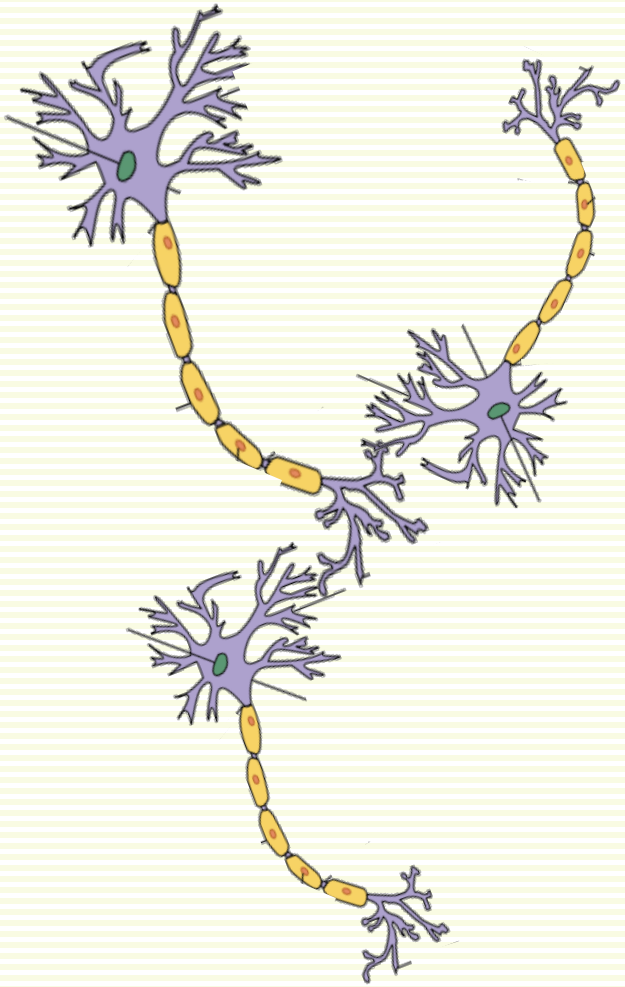
Neuron: Main Components



- **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

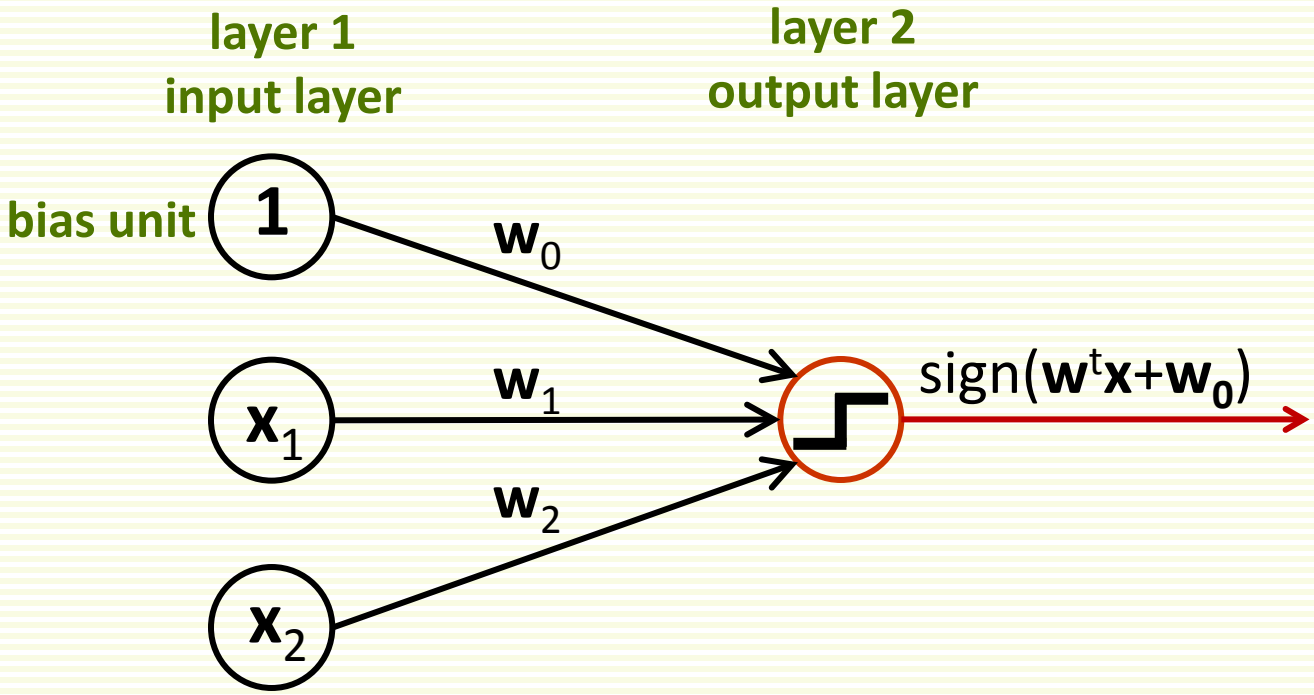
ANN History: Stagnation

- 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

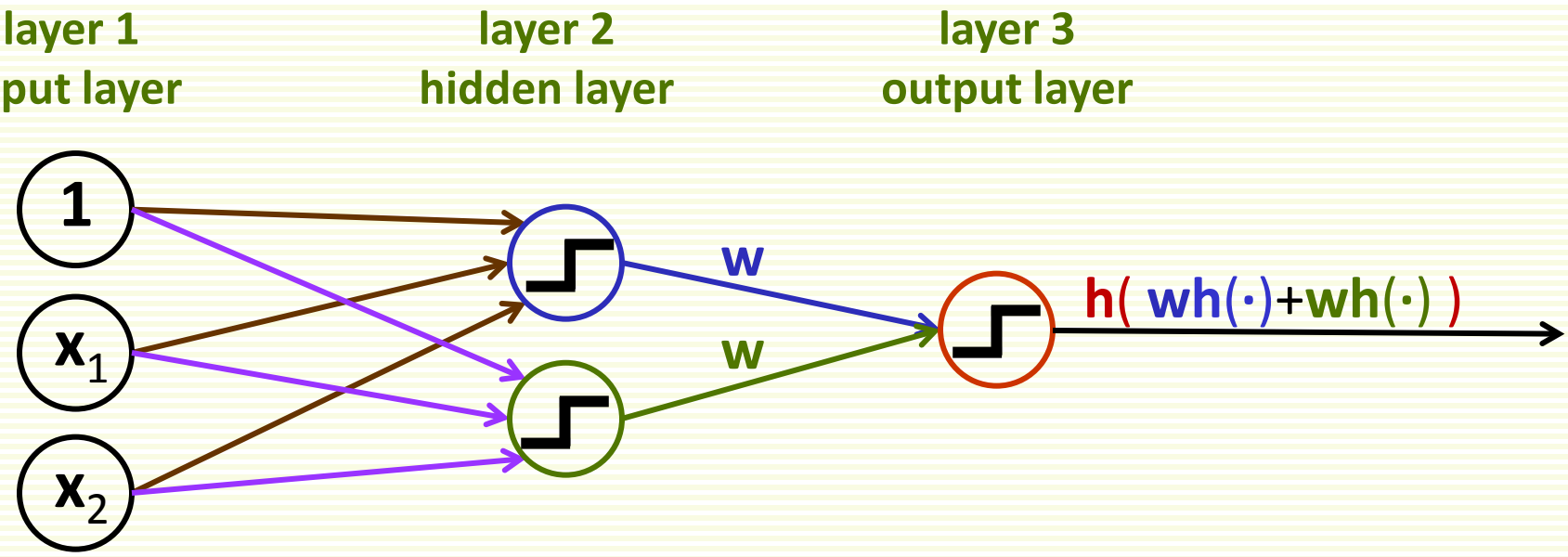
- Revival of ANN in 1980's
- 1982 joint US-Japanese conference on ANN
  - US worries that it will stay behind
- Many examples of mulitlayer NN appear
- 1986, re-discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows a network to learn not linearly separable 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

Perceptron: 1 Layer Neural Network (NN)



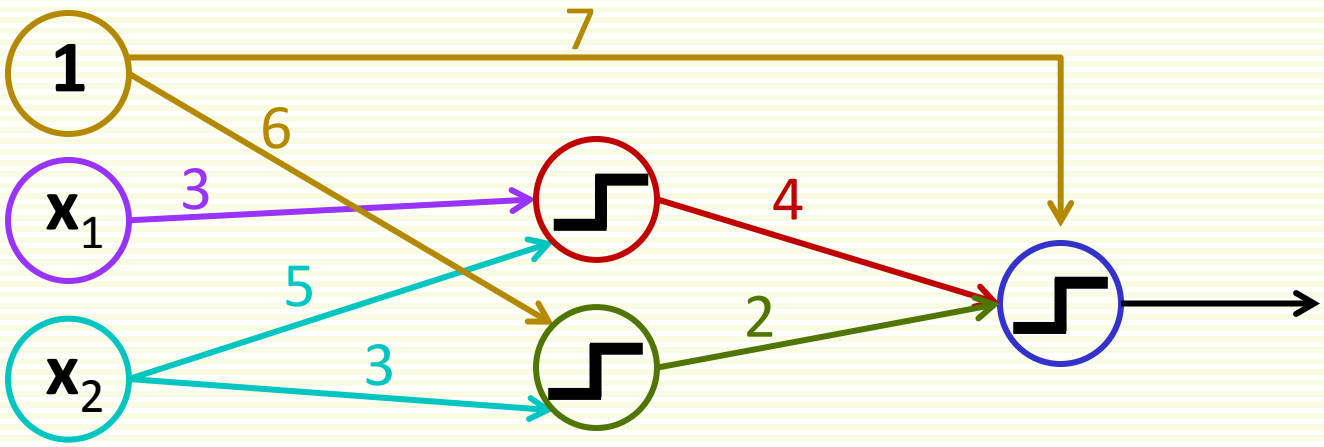
- Linear classifier **f**(**x**) = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w_0}$) is a single neuron "net"
- Input layer units emits features, except bias emits "1"
- Output layer unit applies **h**(t) = **sign**(t)
- **h**(t) is also called an *activation function*

Multilayer Perceptron (MLP)



- First hidden unit outputs $h(w_0+w_1x_1+w_2x_2)$
- Second hidden unit outputs $h(w_0+w_1x_1+w_2x_2)$
- Network implements classifier $f(x) = h(wh(\cdot)+wh(\cdot))$
- More complex boundaries than Perceptron

MLP Small Example



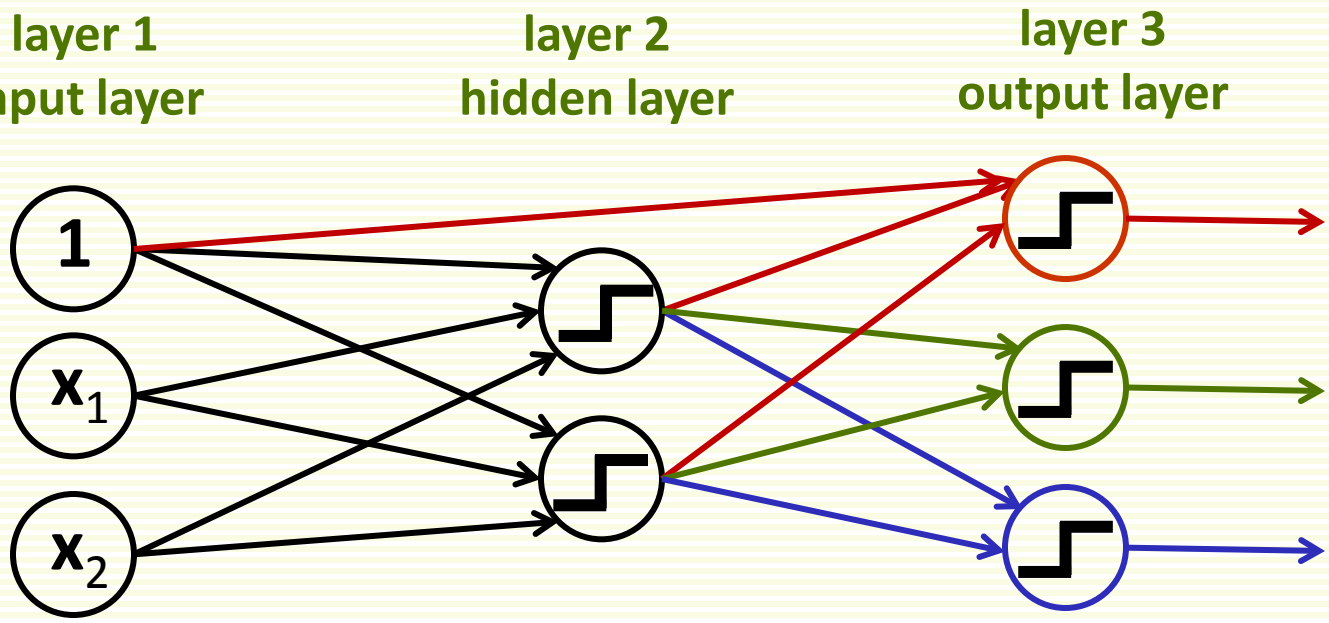
- Implements classifier

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

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

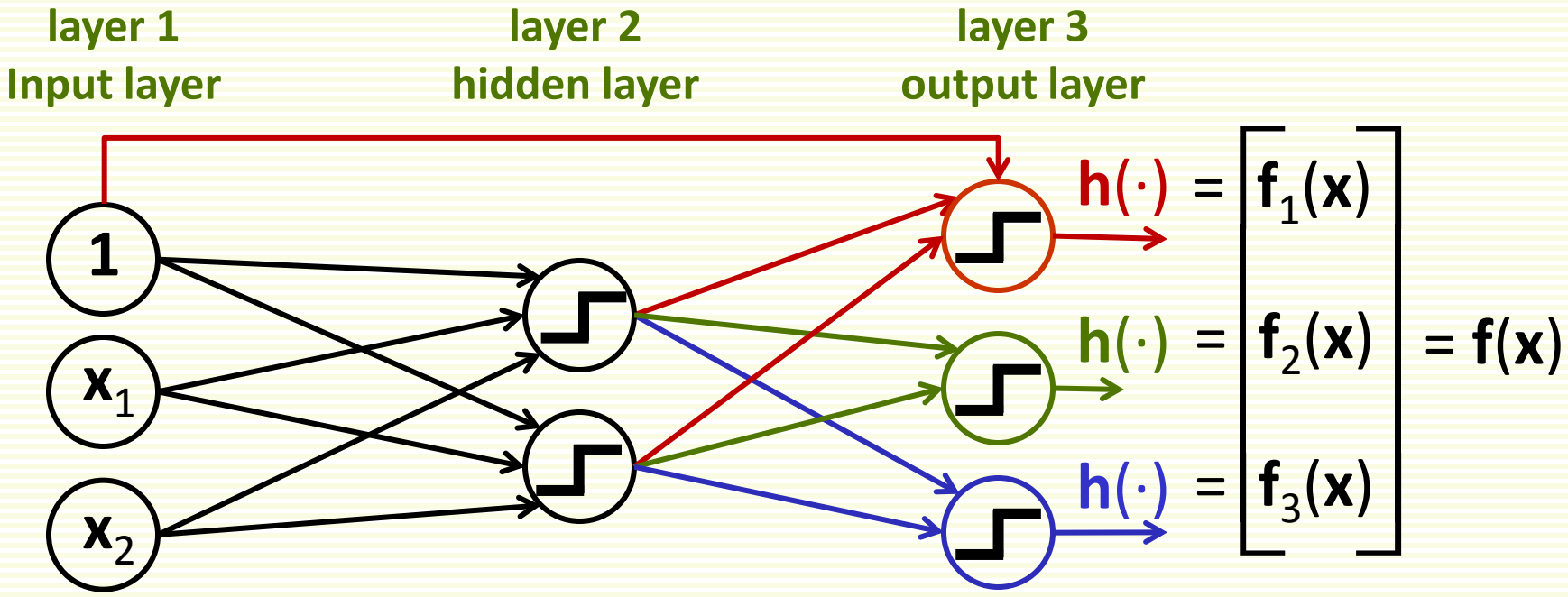
- Computing $\mathbf{f}(\mathbf{x})$ is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

MLP: Multiple Classes



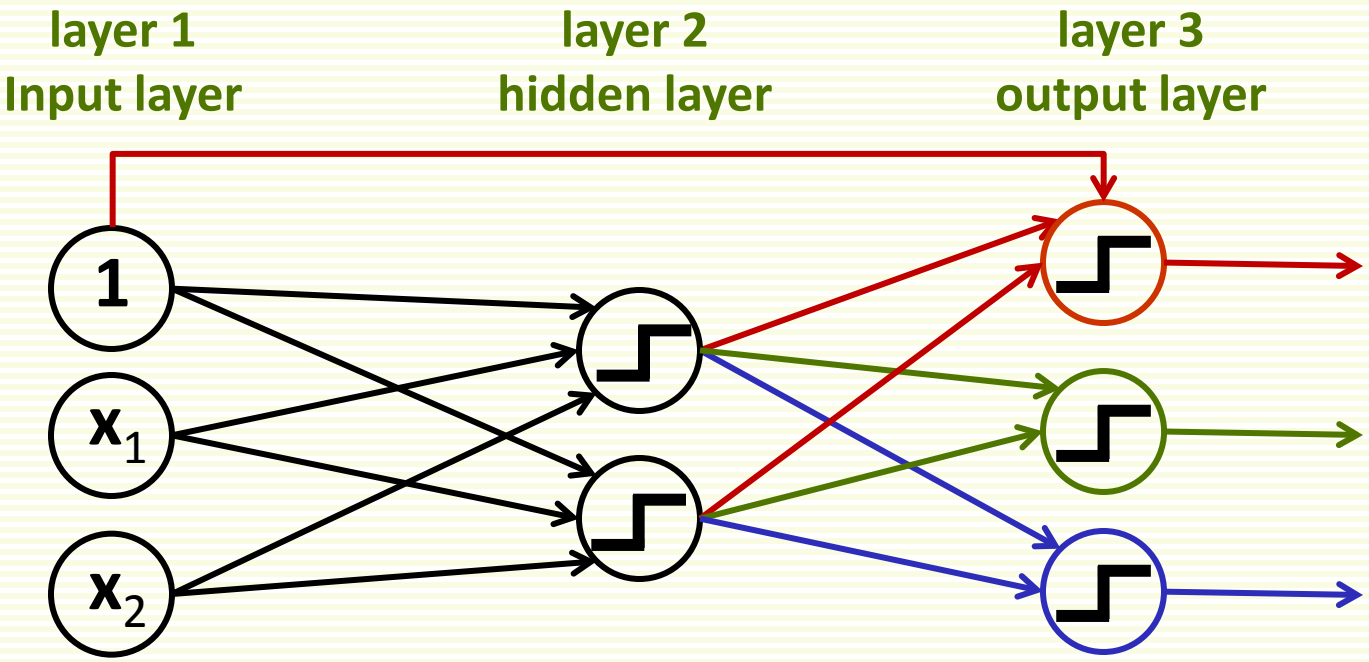
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, can draw in layer 1, or each layer has one

MLP: General Structure



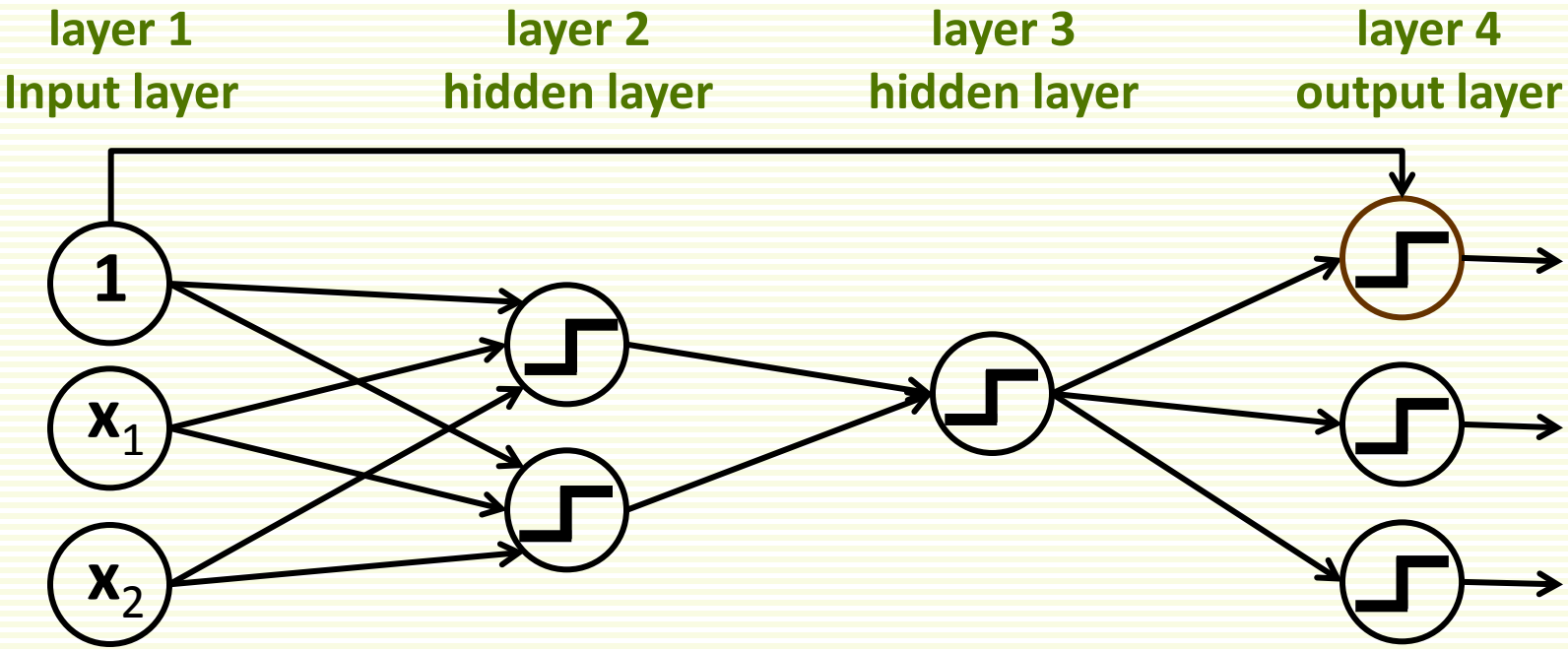
- **f** (**x**) is multi-dimensional
- Classification:
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$ is largest, decide class 3

MLP: General Structure



- Input layer: **d** features, **d** input units
- Output layer: **m** classes, **m** output units
- Hidden layer: how many units?
  - more units correspond to more complex classifiers

MLP: General Structure



- Can have many hidden layers
- *Feed forward* structure
  - **i**th layer connects to (**i+1**)th layer
  - except bias unit can connect to any layer
  - or, alternatively each layer can have its own bias unit

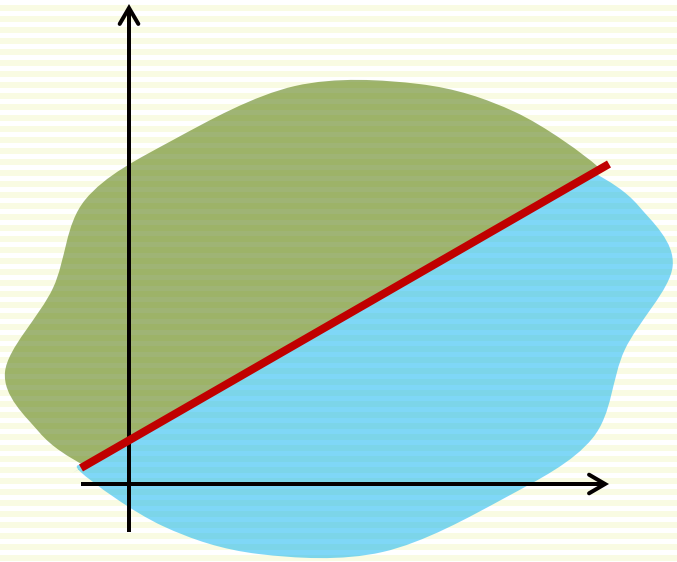
MLP: Overview

- MLP corresponds to rather complex classifier **f**(**x**,**w**)
  - 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 MLP, just as before
  - formulate an objective or *loss* function **L**(**w**)
  - optimize it with gradient descent
    - lots of notation due to gradient complexity
    - lots of tricks to get gradient descent work reasonably well

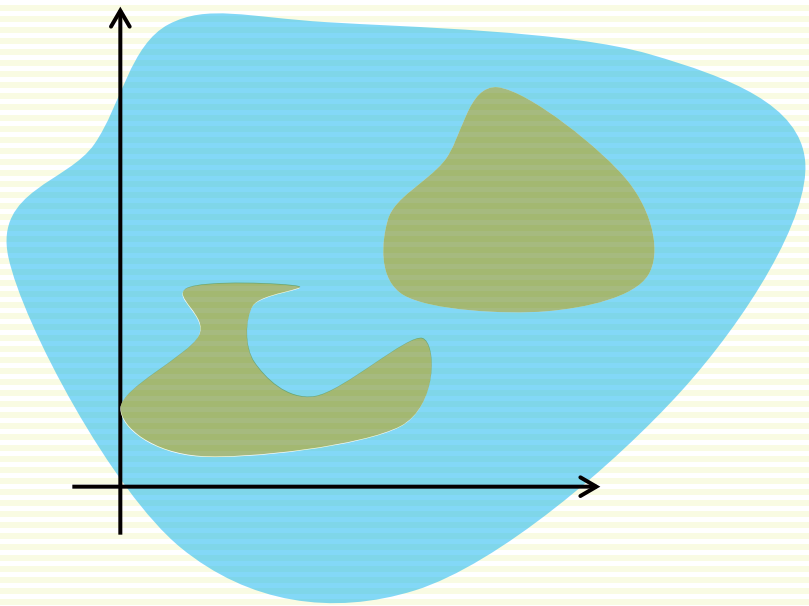
Expressive Power of MLP

- 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



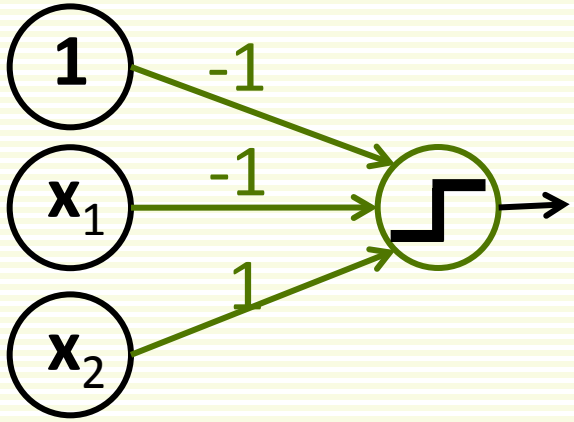
- Perceptron (single layer neural net)

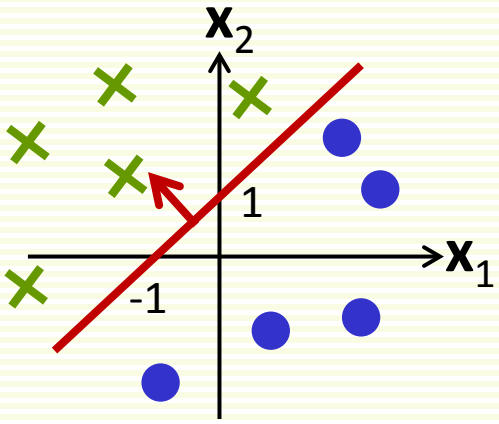


- Arbitrarily complex decision regions
- Even not contiguous

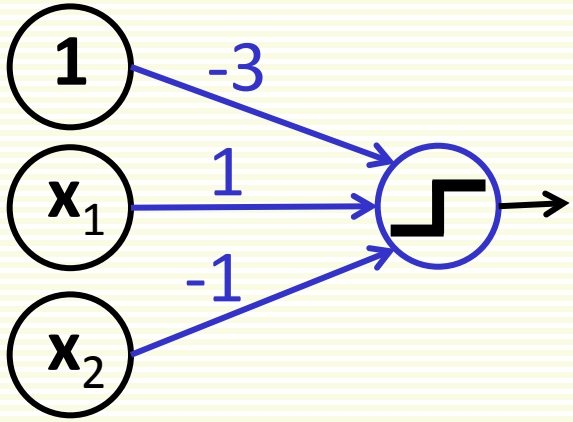
Nonlinear Decision Boundary: Example

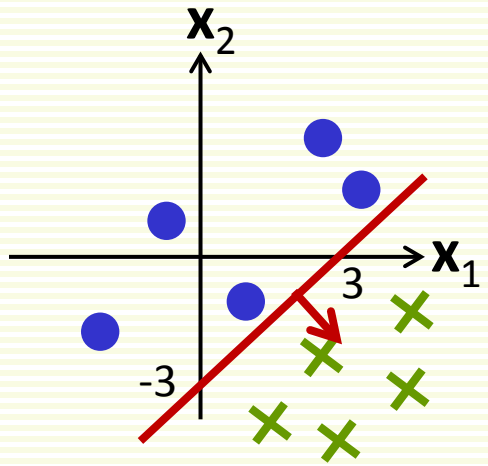
$-x_1 + x_2 - 1 > 0 \Rightarrow$ class 1





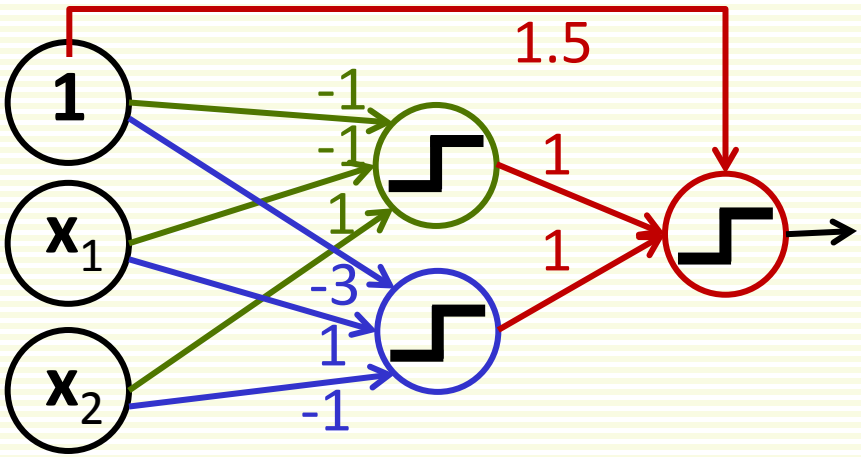
$x_1 - x_2 - 3 > 0 \Rightarrow$ class 1

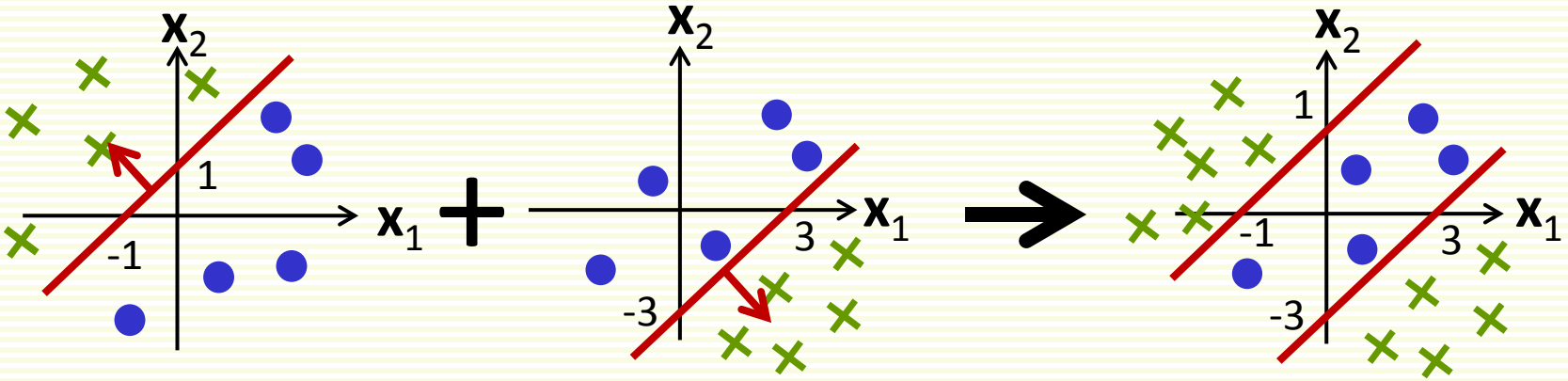




Nonlinear Decision Boundary: Example

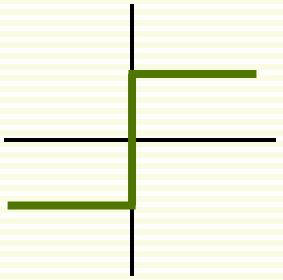
- Combine two Perceptrons into a 3 layer NN



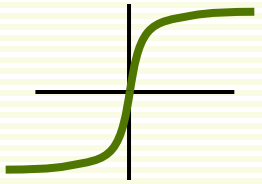


Multi-Layer Neural Networks: Activation Function

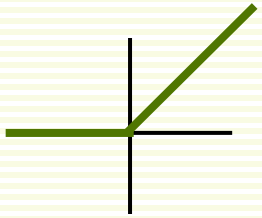
- **h**() = **sign**() does not work for gradient descent



- Can use sigmoid function



- Rectified Linear (RuLu) popular recently



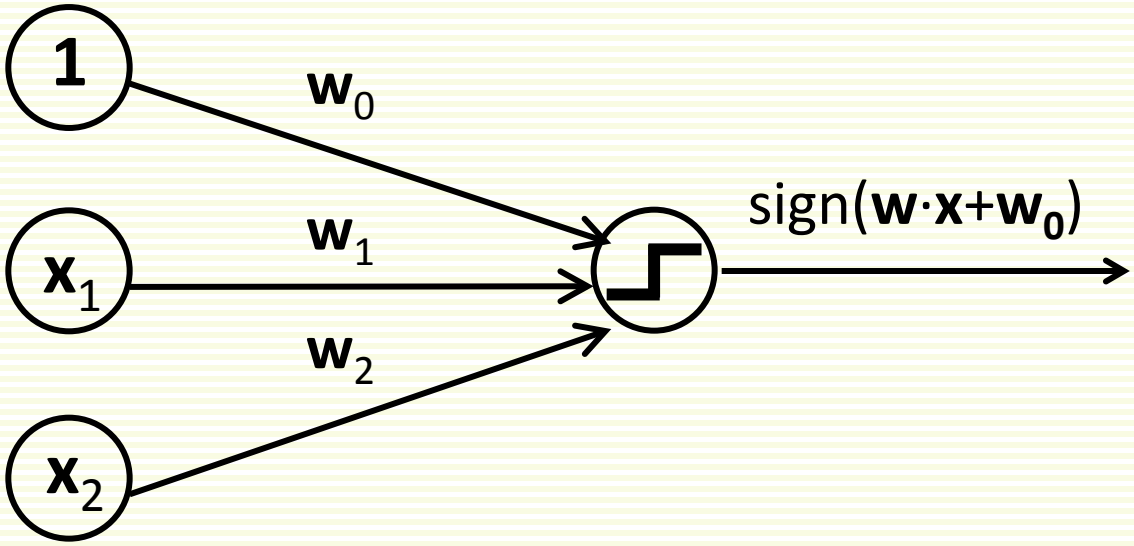
NN: Modes of Operation

- Due to historical reasons, training and testing stages have special names
  - **Backpropagation (or training)**

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

NN: Vector Notation

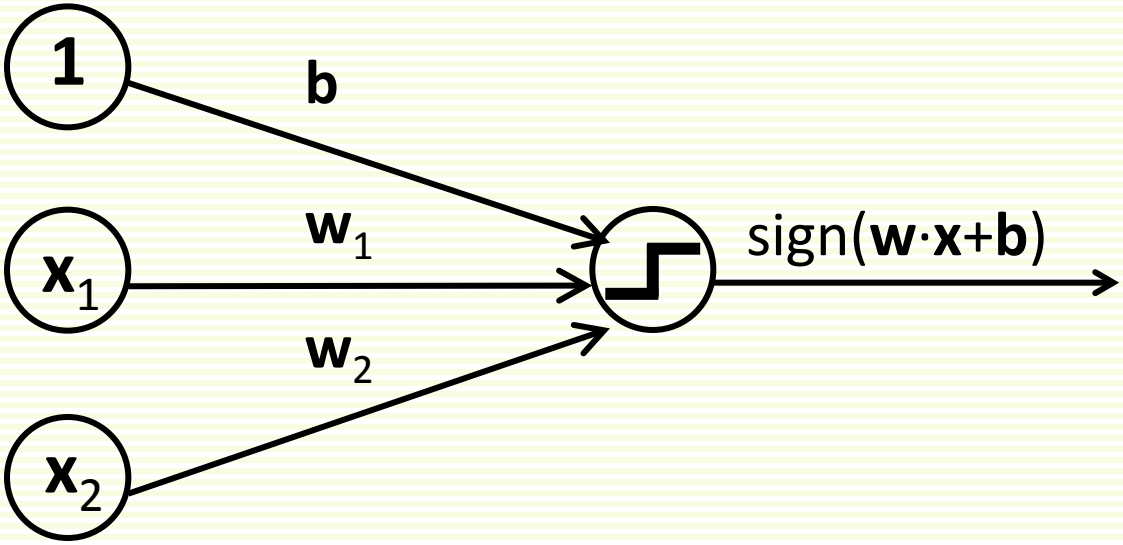
- Want more compact (vector) notation
- Compact notation for Perceptron



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

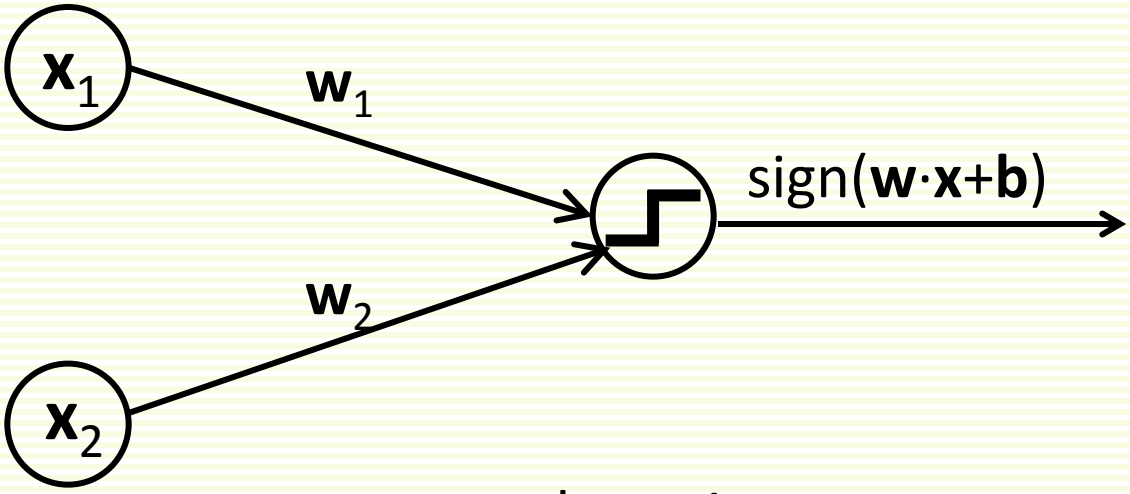
NN: Vector Notation

- Change notation a bit

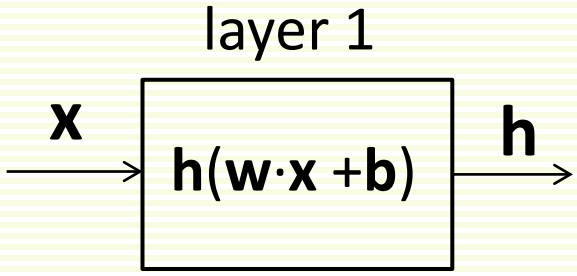


NN: Vector Notation

- Do not draw bias unit

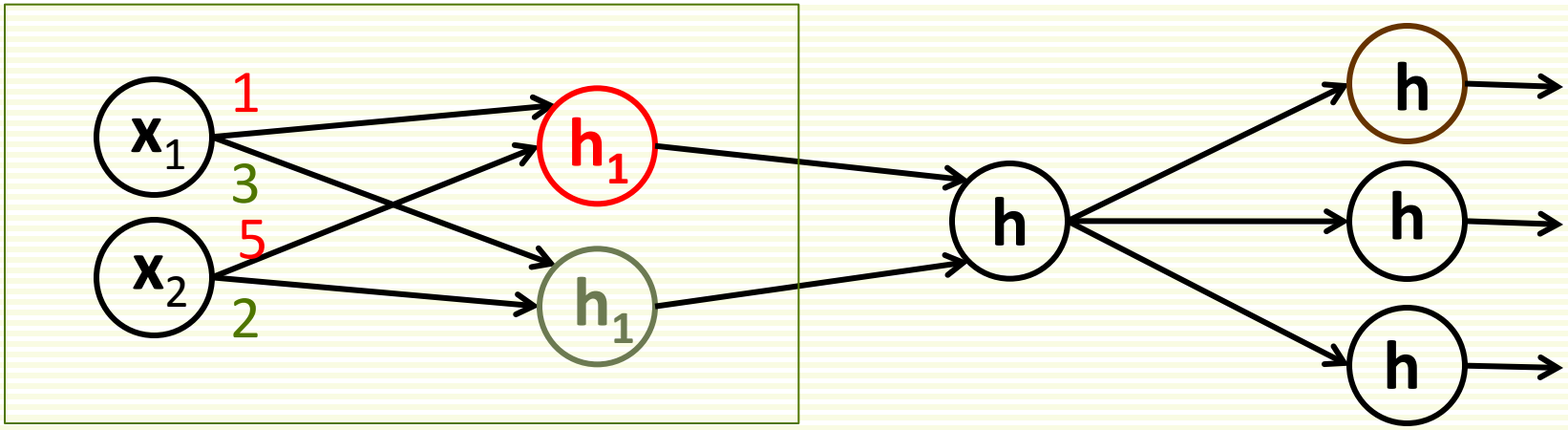


- Compact picture
  - **h**(t) = sign(t)

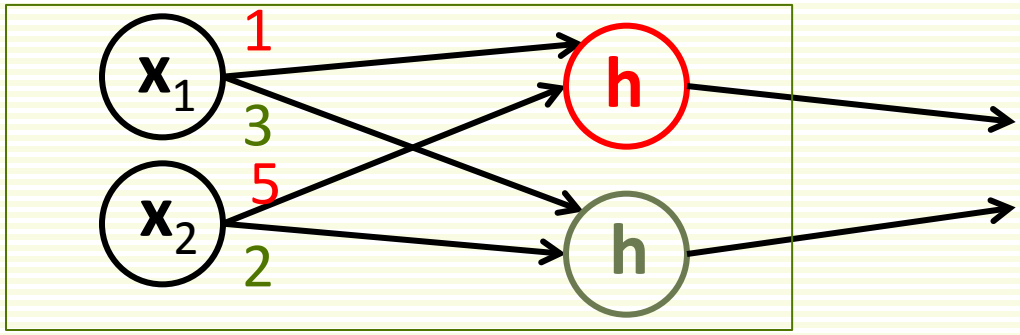


NN: Vector Notation

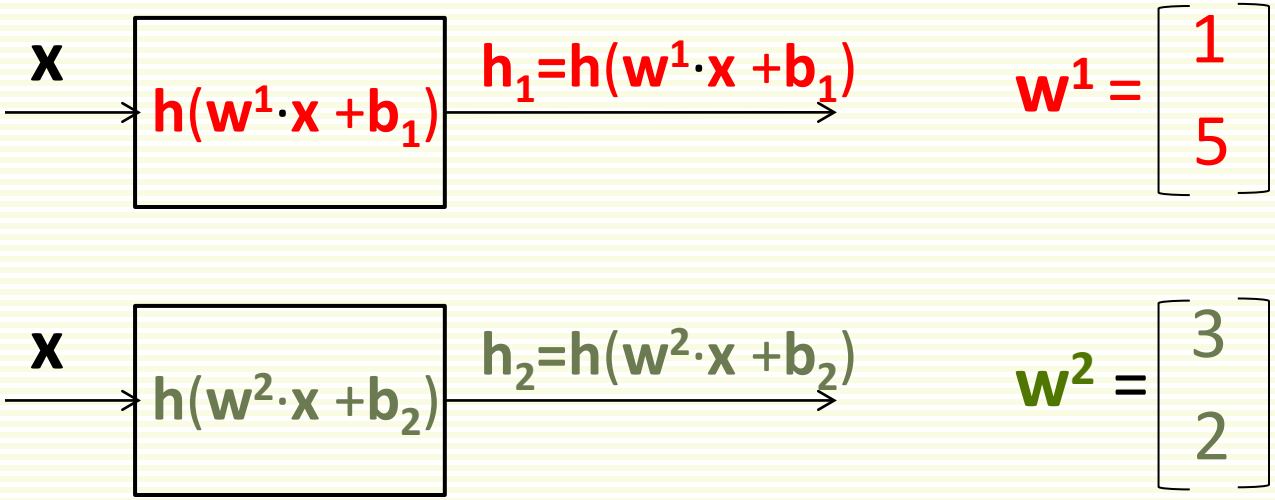
- For now, look just at the first layer (2 perceptrons)



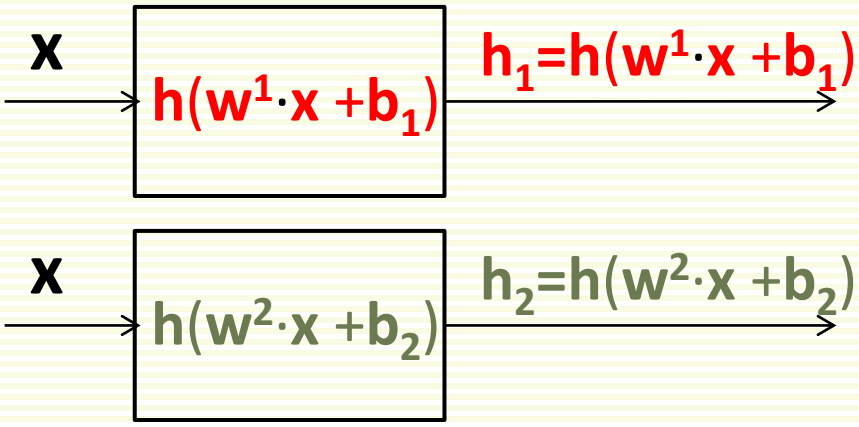
NN: Vector Notation, First Layer



- Red Perceptron has weights w^1 and bias b_1
- Green Perceptron has weights w^2 and bias b_2



NN: Vector Notation , First Layer

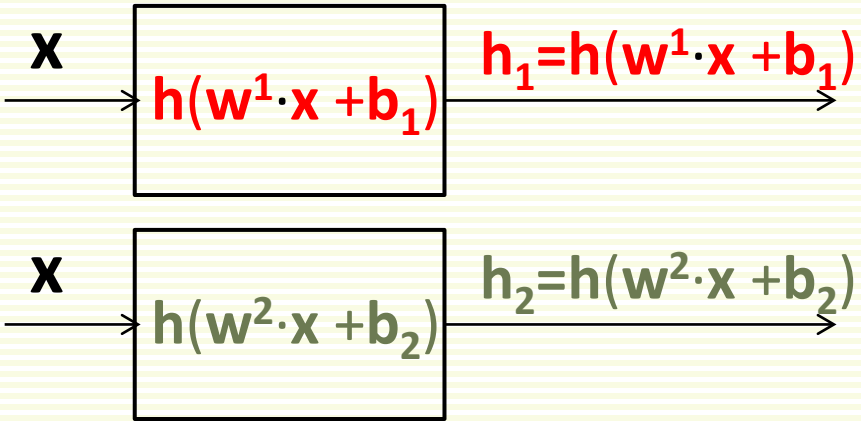


$$\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w^1 \cdot x \\ w^2 \cdot x \end{bmatrix}$$

$$W^1 \cdot x$$

$$w^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad w^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

NN: Vector Notation , First Layer

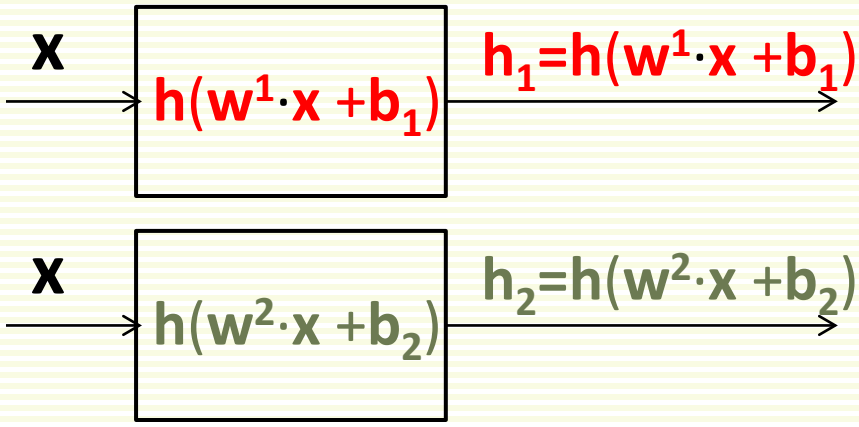


$$\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w^1 \cdot x + b_1 \\ w^2 \cdot x + b_2 \end{bmatrix}$$

$$W^1 \cdot x + b^1$$

$$w^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad w^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

NN: Vector Notation , First Layer



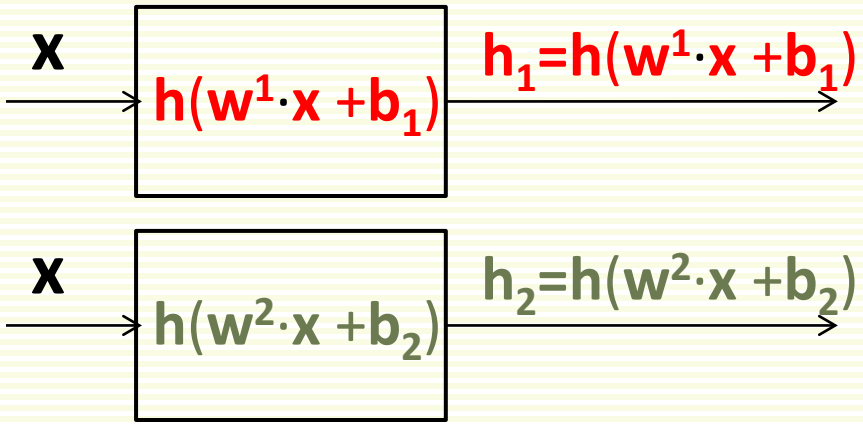
$$\mathbf{h}\left(\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

$$\mathbf{h}(\mathbf{W}^1 \cdot \mathbf{x} + \mathbf{b}^1)$$

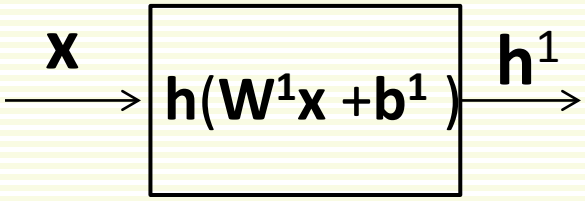
$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \quad \mathbf{w}^2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

- **h**(**v**) for vector **v** means apply **h** to each component of **v**

NN: Vector Notation , First Layer



more compact

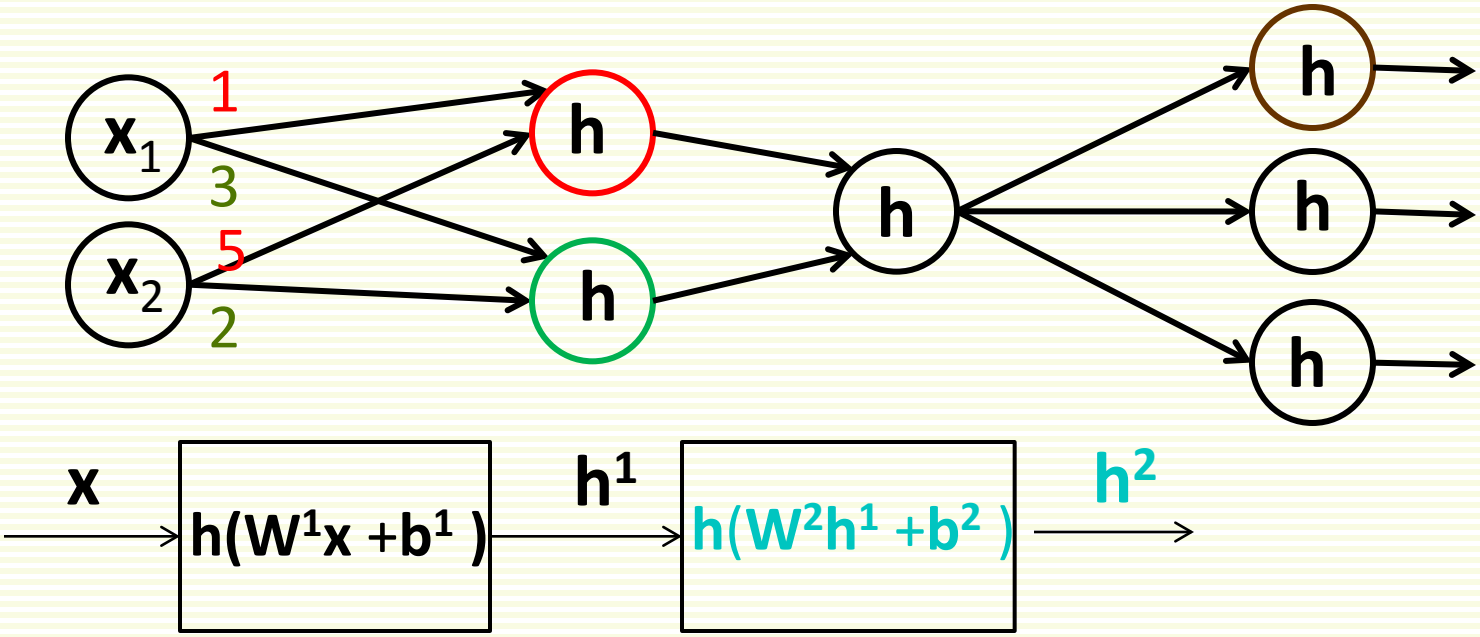


$$W^1 = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \quad b^1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$h^1 = h(W^1 x + b^1) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

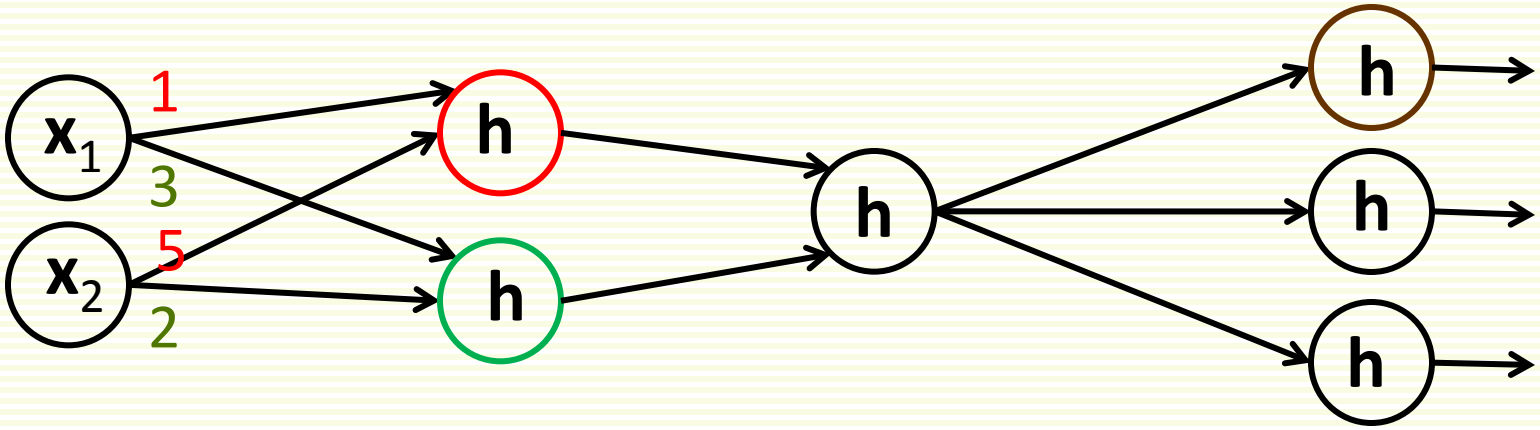
- **h**(**v**) for vector **v** means apply **h** to each component of **v**

NN: Vector Notation, Next Layer

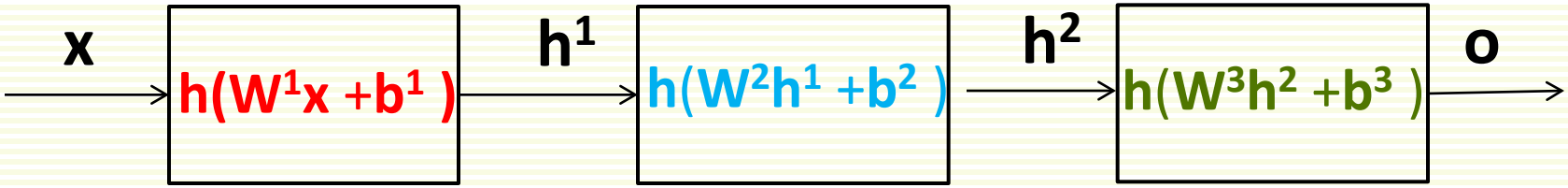


- $\mathbf{W^2}$ is a matrix of weights between hidden layer 1 and 2
  - $\mathbf{W^2(r,c)}$ is weight from unit **c** to unit **r**
- $\mathbf{b^2}$ is a vector of bias weights for second hidden layer
  - b^2_r is bias of unit **r** in second layer
- $\mathbf{h^2}$ is a vector of second layer outputs
  - h^2_r is output of unit **r** in second layer

NN: Vector Notation, all Layers



- Complete depiction



- **o** is the vector from the output layer
- **o** = $h(W^3h^2 + b^3)$
  = $h(W^3h(W^2h^1 + b^2) + b^3)$
  = $h(W^3h(W^2h(W^1x + b^1) + b^2) + b^3)$

NN: Output Representation

- Output of NN is a vector
- So label $\mathbf{y}^i$ of sample $\mathbf{x}^i$ should also be a vector
- Let $\mathbf{x}^i$ be sample of class **k**

$$\mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } \mathbf{k}$$

- Want output unit $\mathbf{o}_k = 1$
- Want other output units zero

$$\mathbf{f}(\mathbf{x}^i) = \mathbf{o} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } \mathbf{k}$$

Training NN: Loss Function

- Want to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i)$
- All network weights $\mathbf{W} = \{\mathbf{W}^1, \mathbf{W}^2, \ldots, \mathbf{W}^l, \mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^l\}$
- Minimum Squared Error (MSE) loss
- Loss on one example $\mathbf{x}^i$

$$\mathbf{L}\left(\mathbf{x}^i, \mathbf{y}^i; \mathbf{W}\right) = \frac{1}{2}\left\|\mathbf{f}\left(\mathbf{x}^i\right) - \mathbf{y}^i\right\|^2 = \frac{1}{2}\sum_{j=1}^{m}\left(\mathbf{f}_j\left(\mathbf{x}^i\right) - \mathbf{y}_j^i\right)^2$$

$$\mathbf{f}(\mathbf{x}^i) = \mathbf{o} = \begin{bmatrix} 0.5 \\ \vdots \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix} \qquad \mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{row } \mathbf{k}$$

- **f** depends on **W**, but too cumbersome to write **f**(**x**,**W**) everywhere
- Cross entropy loss works better than squared difference loss

Training NN: Loss Function

- Let $\mathbf{X} = \mathbf{x}^1, \ldots, \mathbf{x}^n$

  $\mathbf{Y} = \mathbf{y}^1, \ldots, \mathbf{y}^n$

- Loss on all examples: $\mathbf{L}(\mathbf{X}, \mathbf{Y}; \mathbf{W}) = \frac{1}{2} \sum_{i=1}^{n} \left\| \mathbf{f}(\mathbf{x}^i) - \mathbf{y}^i \right\|^2$

- Gradient descent

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

Training NN: Computing Gradient

- Need to find derivative of $\mathbf{L}(\mathbf{X},\mathbf{Y};\mathbf{W})$ wrt every network weight $\mathbf{w}_i$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{w}_i}$$

- Gradient descent tells us to add the following quantity to $\mathbf{w}_i$

$$\Delta \mathbf{w}_i = -\alpha \frac{\partial \mathbf{L}}{\partial \mathbf{w}_i}$$

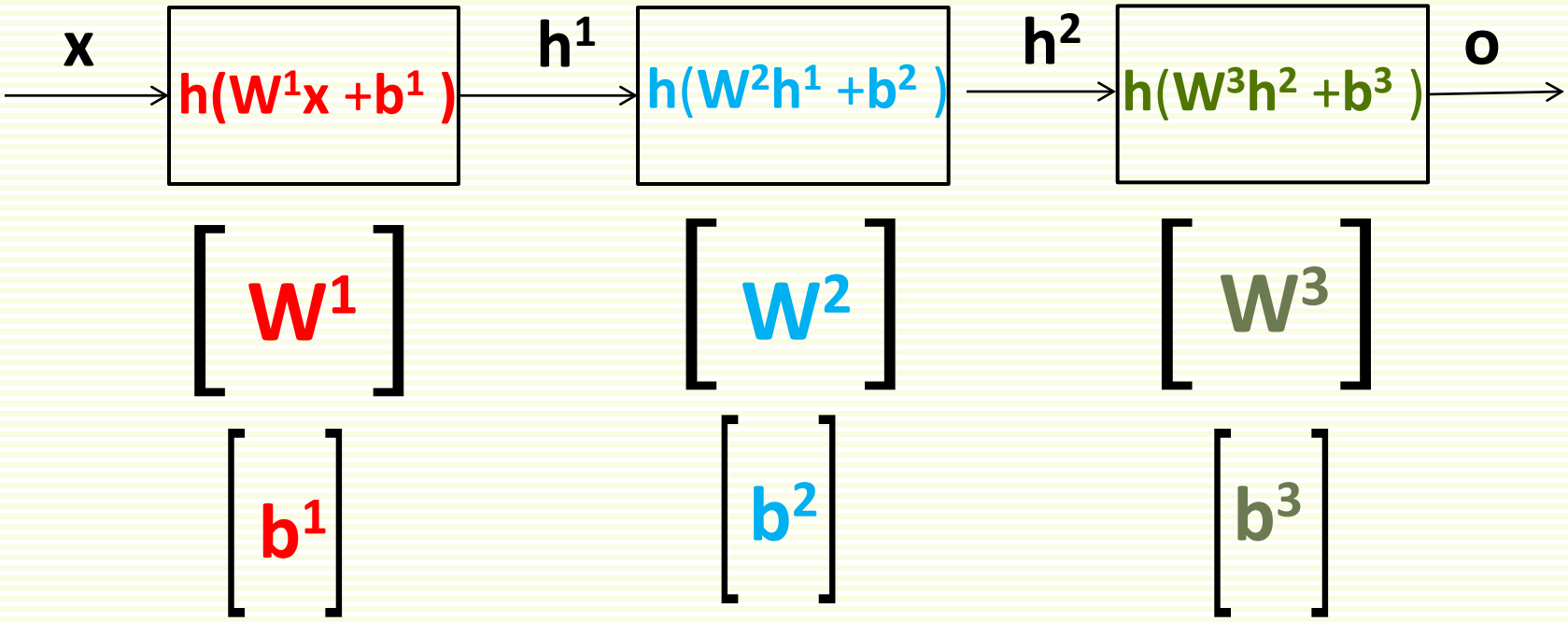
  - where α is the learning rate

- Perform weight update for all network weights

$$\mathbf{w}_i = \mathbf{w}_i + \Delta \mathbf{w}_i$$

Training NN: Computing Gradient

- How many weights do we have in our network?



- Weights are in matrices $\mathbf{W}^1, \mathbf{W}^2, \ldots, \mathbf{W}^l$
- And are in vectors $\mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^l$

Training NN: Computing Gradient

- Consider matrix $\mathbf{W}^1 = \begin{bmatrix} \mathbf{w}^1_{11} & \cdots & \mathbf{w}^1_{1k} \\ \vdots & \cdots & \vdots \\ \mathbf{w}^1_{d1} & \cdots & \mathbf{w}^1_{dk} \end{bmatrix}$
- Need to compute derivative wrt every $\mathbf{w}^1_{js}$
- Organize derivatives in matrix

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^1} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{11}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{1k}} \\ \vdots & \cdots & \vdots \\ \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{d1}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}^1_{dk}} \end{bmatrix}$$

Training NN: Computing Gradient

- Chain rule for derivatives of composed functions

$$\frac{\partial \mathbf{L}(\mathbf{h}(\mathbf{w}))}{\partial \mathbf{w}} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{x}}$$

- NN is a composition of compositions … of compositions of functions **h**(**h**(**h** ()))
- Have to apply the chain rule a lot

Training NN: Computing Gradient

- Consider derivative for one sample **x**, with true label **y**
  - dropped super-indexes for clarity

$$\mathbf{L}(\mathbf{x},\mathbf{y};\mathbf{W}) = \frac{1}{2}\sum_j \left(\mathbf{f}_j(\mathbf{x}) - \mathbf{y}_j\right)^2 = \frac{1}{2}\sum_j \left(\mathbf{o} - \mathbf{y}_j\right)^2$$

- First take derivatives wrt $\mathbf{o}_j$

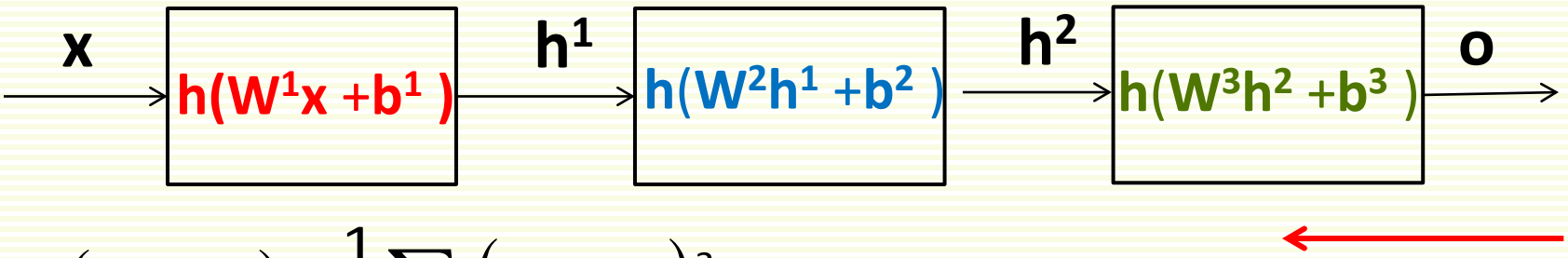
$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}_j} = \mathbf{f}_j(\mathbf{x}) - \mathbf{y}_j$$

- Vector of derivatives wrt **o**

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \cdots \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_m} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1(\mathbf{x}) - \mathbf{y}_1 \\ \cdots \\ \mathbf{f}_m(\mathbf{x}) - \mathbf{y}_m \end{bmatrix}$$

Training NN: Computing Gradient



$$L(x, y; w) = \frac{1}{2}\sum_j (y_j - o_j)^2$$

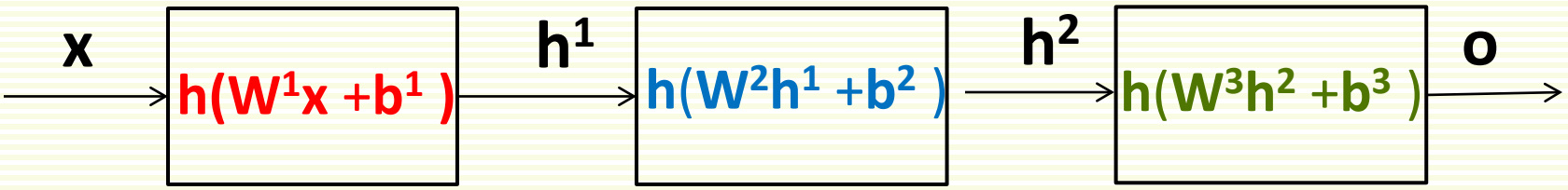
- Compute derivatives backwards, starting in last layer

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial W^3}$$

$$\frac{\partial L}{\partial h^2} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h^2}$$

$$\frac{\partial L}{\partial b^3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial b^3}$$

Training NN: Computing Gradient



- Let vector $\mathbf{a}^3 = \mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3$ $\quad \mathbf{a}^3 = \begin{bmatrix} a^3_1 \\ a^3_2 \end{bmatrix}$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \frac{\partial \mathbf{L}}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{W}^3} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^3\right)\right)\left(\mathbf{f}(\mathbf{x}) - \mathbf{y}\right)\left(\mathbf{h}^2\right)^T$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{h}^2} = \frac{\partial \mathbf{L}}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}^2} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^3\right)\right)\left(\mathbf{W}^3\right)^T\left(\mathbf{f}(\mathbf{x}) - \mathbf{y}\right)$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{b}^3} = \frac{\partial \mathbf{L}}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{b}^3} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^3\right)\right)\left(\mathbf{f}(\mathbf{x}) - \mathbf{y}\right)$$

Training NN: Computing Gradient

- Sketch of derivation for $\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3}$

$$\mathbf{W}^3 = \begin{bmatrix} \mathbf{w}_{11}^3 & \mathbf{w}_{12}^3 \\ \mathbf{w}_{21}^3 & \mathbf{w}_{22}^3 \end{bmatrix}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{22}^3} \end{bmatrix}$$

Training NN: Computing Gradient

- Recall $\mathbf{W}^3 = \begin{bmatrix} \mathbf{w}_{11}^3 & \mathbf{w}_{12}^3 \\ \mathbf{w}_{21}^3 & \mathbf{w}_{22}^3 \end{bmatrix} \quad \mathbf{h}^2 = \begin{bmatrix} \mathbf{h}_1^2 \\ \mathbf{h}_2^2 \end{bmatrix} \quad \mathbf{b}^2 = \begin{bmatrix} \mathbf{b}_1^2 \\ \mathbf{b}_2^2 \end{bmatrix}$

- Thus $\mathbf{W}^3\mathbf{h}^2 = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 \end{bmatrix}$

$$\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3 = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

$$\mathbf{o} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3\right) \\ \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3\right) \end{bmatrix}$$

Training NN: Computing Gradient

$$\mathbf{o} = \begin{bmatrix} \mathbf{o}_1 \\ \mathbf{o}_2 \end{bmatrix} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3 \\ \mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3 \end{bmatrix}$$

- Using chain rule

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{22}^3} \end{bmatrix}$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^3}$$

Training NN: Computing Gradient

- Need $\dfrac{\partial \mathbf{o}}{\partial \mathbf{W}^3}$

$$\mathbf{o} = \mathbf{h}\left(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3\right) = \begin{bmatrix} \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{11}^3 + \mathbf{h}_2^2\mathbf{w}_{12}^3 + \mathbf{b}_1^3\right) \\ \mathbf{h}\left(\mathbf{h}_1^2\mathbf{w}_{21}^3 + \mathbf{h}_2^2\mathbf{w}_{22}^3 + \mathbf{b}_2^3\right) \end{bmatrix} = \begin{bmatrix} \mathbf{h}\left(\mathbf{a}_1^3\right) \\ \mathbf{h}\left(\mathbf{a}_1^3\right) \end{bmatrix}$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^3} = \begin{bmatrix} \dfrac{\partial \mathbf{o}}{\partial \mathbf{w}_{11}^3} & \dfrac{\partial \mathbf{o}}{\partial \mathbf{w}_{12}^3} \\ \dfrac{\partial \mathbf{o}}{\partial \mathbf{w}_{21}^3} & \dfrac{\partial \mathbf{o}}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_1^2 & \mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_2^2 \\ \mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_1^2 & \mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_2^2 \end{bmatrix}$$

Training NN: Computing Gradient

- Continue

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{o}}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_1^2 & \mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_2^2 \\ \mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_1^2 & \mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_2^2 \end{bmatrix}$$

- Plug into

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{11}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\frac{\partial \mathbf{o}_1}{\partial \mathbf{w}_{12}^3} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{21}^3} & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\frac{\partial \mathbf{o}_2}{\partial \mathbf{w}_{22}^3} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1}\mathbf{h}'\left(\mathbf{a}_1^3\right)\mathbf{h}_2^2 \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2}\mathbf{h}'\left(\mathbf{a}_2^3\right)\mathbf{h}_2^2 \end{bmatrix}$$

Training NN: Computing Gradient

- Rewrite

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \mathbf{h}'\left(\mathbf{a}_1^3\right) \mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \mathbf{h}'\left(\mathbf{a}_1^3\right) \mathbf{h}_2^2 \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \mathbf{h}'\left(\mathbf{a}_2^3\right) \mathbf{h}_1^2 & \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \mathbf{h}'\left(\mathbf{a}_2^3\right) \mathbf{h}_2^2 \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \mathbf{h}'\left(\mathbf{a}_1^3\right) \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \mathbf{h}'\left(\mathbf{a}_2^3\right) \end{bmatrix} \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{h}'\left(\mathbf{a}_1^3\right) & 0 \\ 0 & \mathbf{h}'\left(\mathbf{a}_2^3\right) \end{bmatrix} \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \end{bmatrix} \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

$$= \mathbf{diag} \begin{bmatrix} \mathbf{h}'\left(\mathbf{a}_1^3\right) \\ \mathbf{h}'\left(\mathbf{a}_2^3\right) \end{bmatrix} \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \end{bmatrix} \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

Training NN: Computing Gradient

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \mathbf{diag}\begin{bmatrix} \mathbf{h}'\left(\mathbf{a}_1^3\right) \\ \mathbf{h}'\left(\mathbf{a}_2^3\right) \end{bmatrix} \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{o}_1} \\ \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \end{bmatrix} \begin{bmatrix} \mathbf{h}_1^2 & \mathbf{h}_2^2 \end{bmatrix}$$

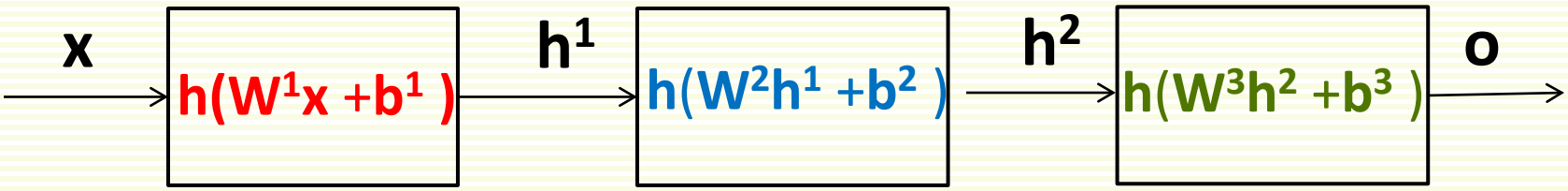
- Recall

$$\frac{\partial \mathbf{L}}{\partial \mathbf{o}} = \mathbf{f}(\mathbf{x}) - \mathbf{y} = \begin{bmatrix} \mathbf{f}_1(\mathbf{x}) - \mathbf{y}_1 \\ \vdots \\ \mathbf{f}_m(\mathbf{x}) - \mathbf{y}_m \end{bmatrix}$$

- We get what we want

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^3} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^3\right)\right)\left(\mathbf{f}(\mathbf{x}) - \mathbf{y}\right)\left(\mathbf{h}^2\right)^{\mathbf{T}}$$

Training NN: Computing Gradient



$$\frac{\partial L}{\partial h^2}$$

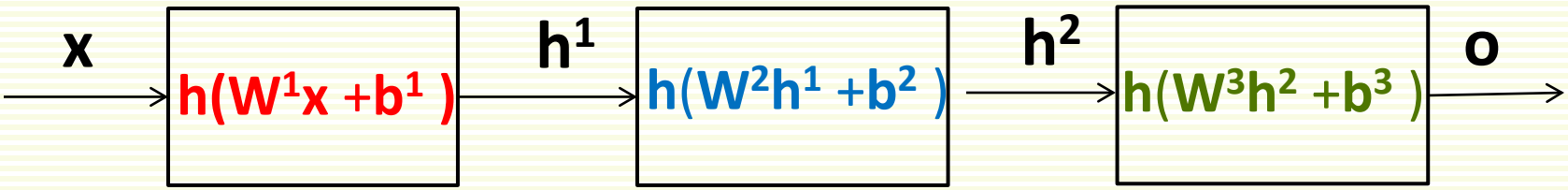
- Continue computing backwards
- Let vector $\mathbf{a^2} = \mathbf{W^2 h^1 + b^2}$

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial W^2}$$

$$\frac{\partial L}{\partial h^1} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial h^1}$$

$$\frac{\partial L}{\partial b^2} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial b^2}$$

Training NN: Computing Gradient



$$\frac{\partial L}{\partial h^2}$$

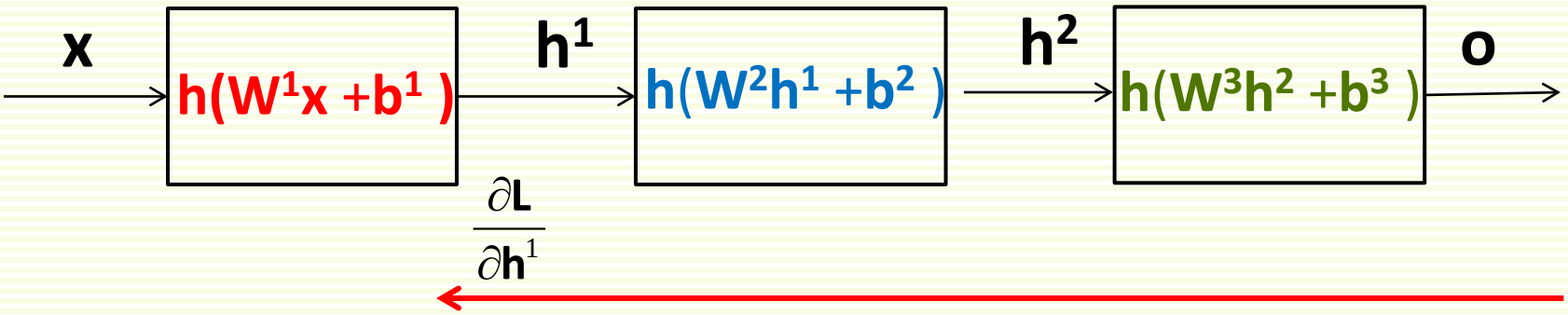
- Continue computing backwards
- Let vector $a^2 = W^2h^1 + b^2$

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial W^2} = \mathbf{diag}\left(h'\left(a^2\right)\right)\frac{\partial L}{\partial h^2}\left(h^1\right)^T$$

$$\frac{\partial L}{\partial h^1} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial h^1} = \mathbf{diag}\left(h'\left(a^2\right)\right)\left(W^2\right)^T\frac{\partial L}{\partial h^2}$$

$$\frac{\partial L}{\partial b^2} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial b^2} = \mathbf{diag}\left(h'\left(a^2\right)\right)\frac{\partial L}{\partial h^2}$$

Training NN: Computing Gradient



- Continue computing backwards
- Let vector $\mathbf{a^1} = \mathbf{W^1 x^1} + \mathbf{b^1}$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^1} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}\frac{\partial \mathbf{h}^1}{\partial \mathbf{W}^1} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^1\right)\right)\frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}\mathbf{x}^{\mathsf{T}}$$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{b}^1} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}\frac{\partial \mathbf{h}^1}{\partial \mathbf{b}^1} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^1\right)\right)\frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}$$

Training Protocols

- Batch Protocol
  - full gradient descent
  - weights are updated only after all examples are processed
  - might be very slow to train
- Single Sample Protocol
  - examples are chosen randomly from the training set
  - weights are updated after every example
  - weighs get changed faster than batch, less stable
  - One iteration over all samples (in random order) is called an **epoch**
- Mini Batch
  - Divide data in batches, and update weights after processing each batch
  - Middle ground between single sample and batch protocols
  - Helps to prevent over-fitting in practice, think of it as “noisy” gradient
  - allows CPU/GPU memory hierarchy to be exploited so that it trains much faster than single-sample in terms of wall-clock time
  - One iteration over all mini-batches is called an **epoch**

Training DNN: Initialization

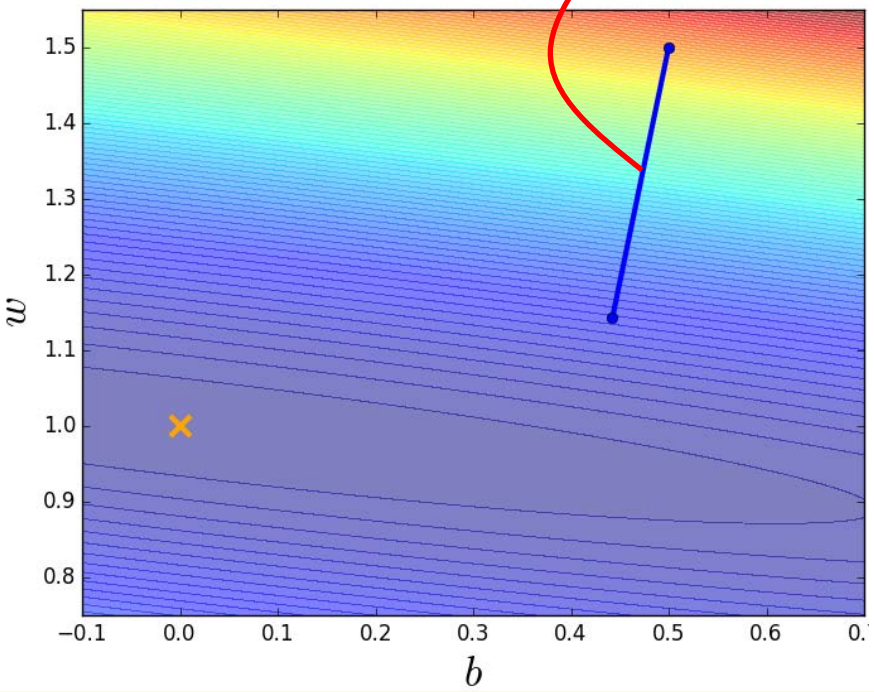
- For gradient descent, need to pick initialization parameters **w**
  - do not set all the parameters **w** equal
  - set the parameters in **w** randomly

Training DNN: Learning Rate

- Can adjust α at the training time
- The loss function $\mathbf{L}(\mathbf{w})$ should decrease during gradient descent
  - if $\mathbf{L}(\mathbf{w})$ oscillates, α is too large, decrease it
  - if $\mathbf{L}(\mathbf{w})$ goes down but very slowly, α is too small, increase it

Training DNN: Gradient descent

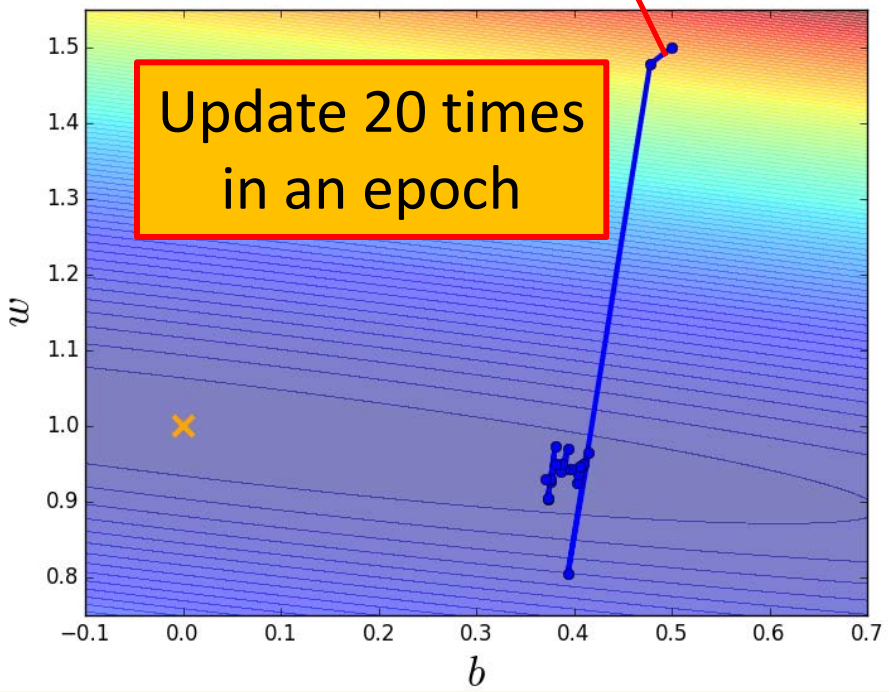
see all examples



Gradient descent

see only one example

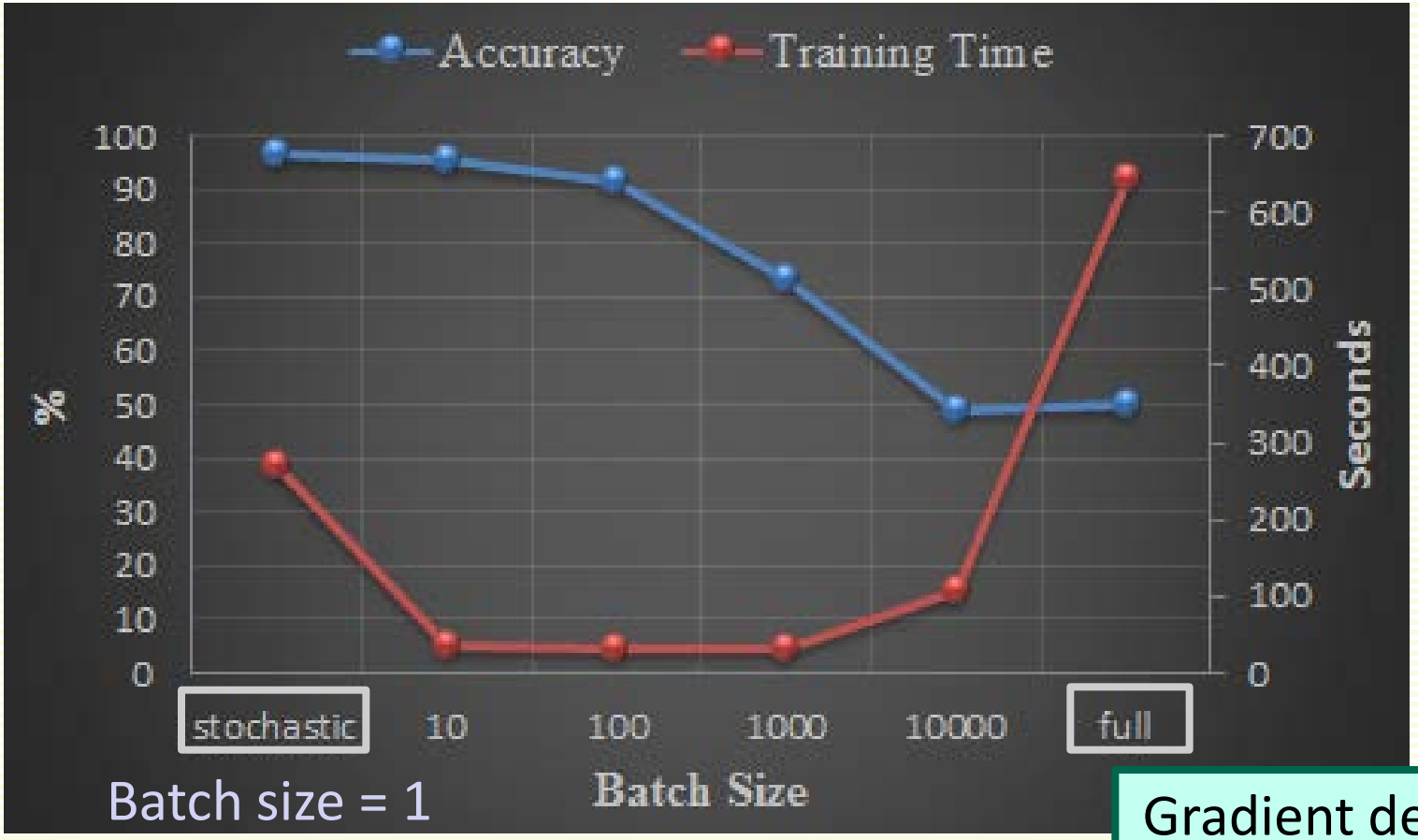
Update 20 times in an epoch



Stochastic gradient descent, 1 epoch

Training DNN: Gradient descent

- Real Example: Handwriting Digit Classification



Training DNN: 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:

$$\mathbf{w}^{t+1} = \mathbf{w}^t + (1-\beta)\nabla\mathbf{L}(\mathbf{w}^t) + \beta\Delta\mathbf{w}^{t-1}$$

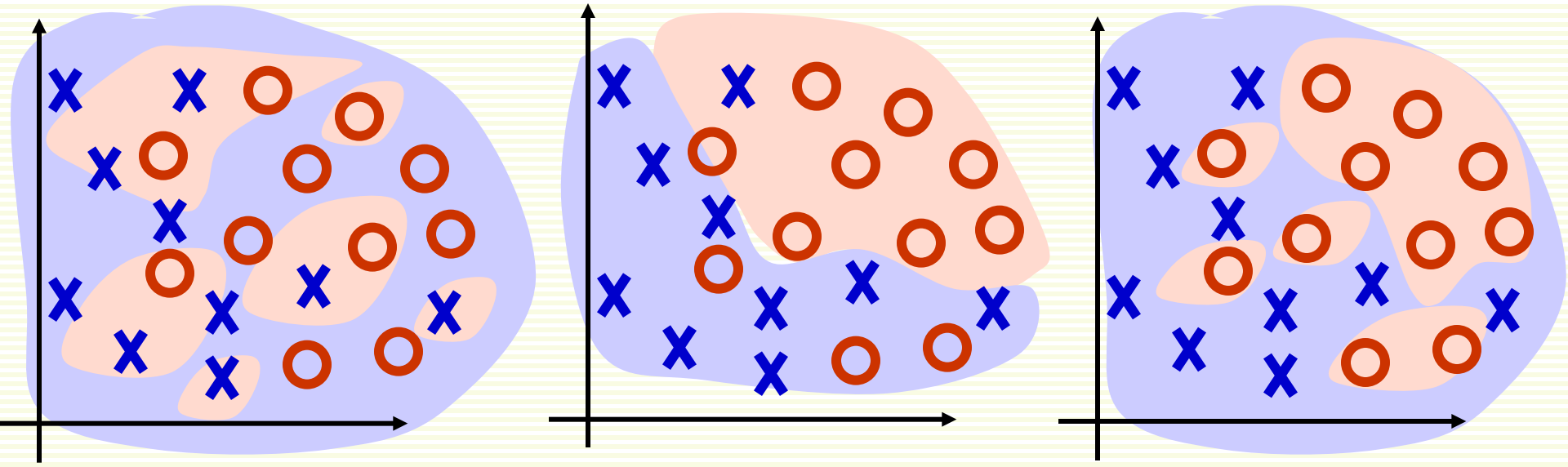
steepest descent direction

previous direction

Training DNN: Normalization

- Features should be normalized for faster convergence
- Suppose fish length is 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 will do
  - test samples should be normalized exactly as training samples

MLP Training: How long to Train?



training time

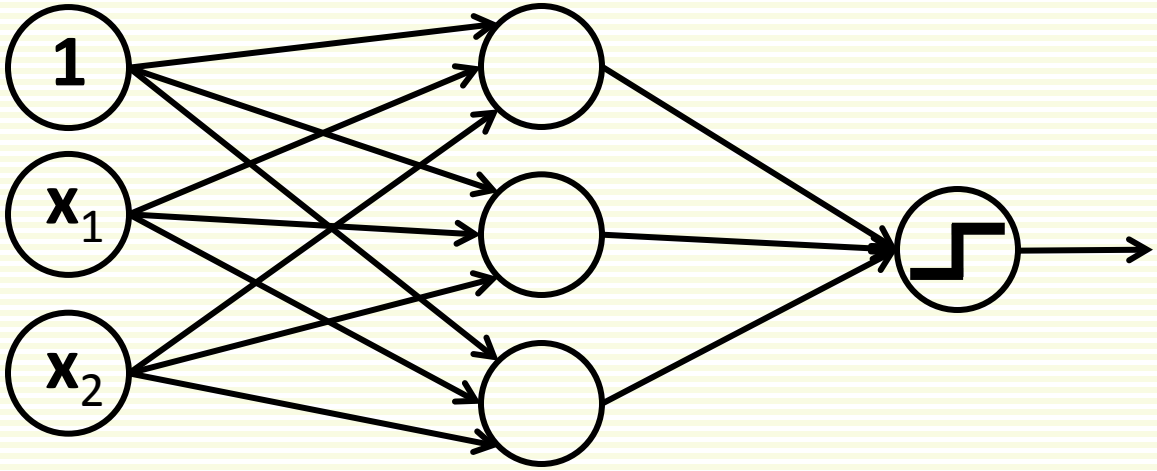
Large training error: random decision regions in the beginning - underfit

Small training error: decision regions improve with time

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

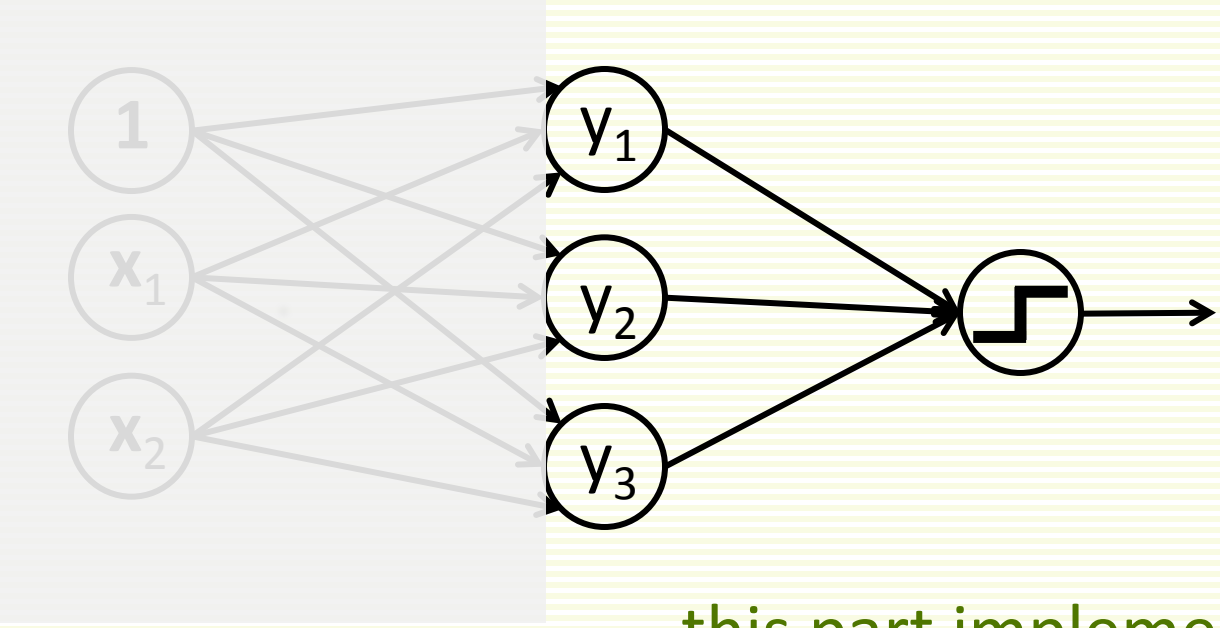
can learn when to stop training through validation

MLP as Non-Linear Feature Mapping



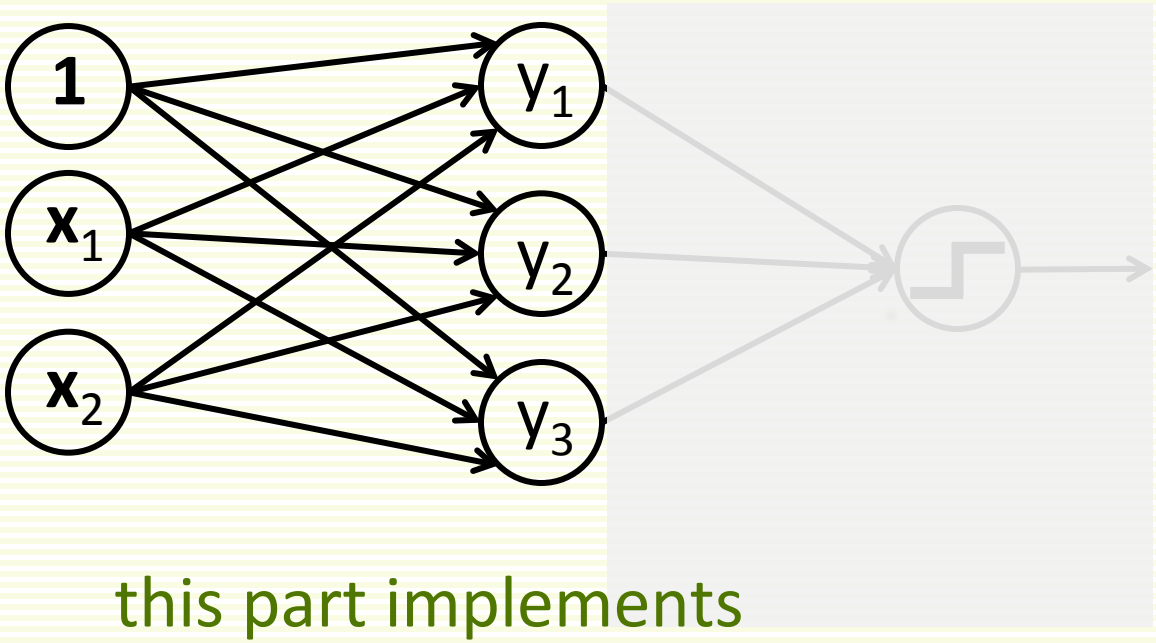
- MLP can be interpreted as first mapping input features to new features
- Then applying Perceptron (linear classifier) to the new features

MLP as Non-Linear Feature Mapping



this part implements
Perceptron (liner classifier)

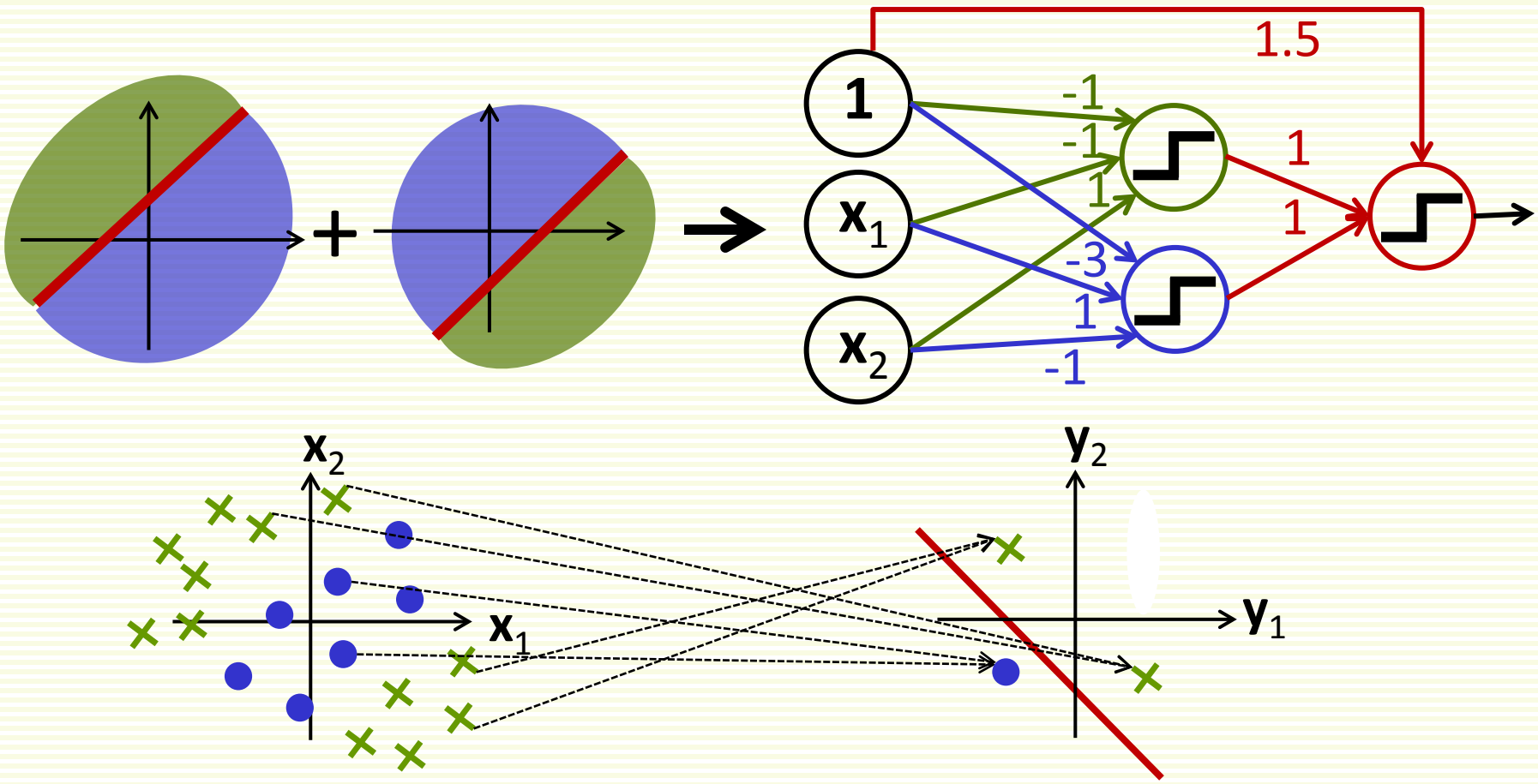
MLP as Non-Linear Feature Mapping



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

MLP as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:



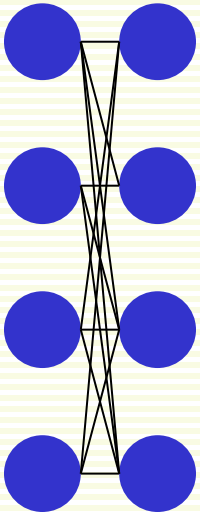
non linearly separable in the original feature space

linearly separable in the new feature space

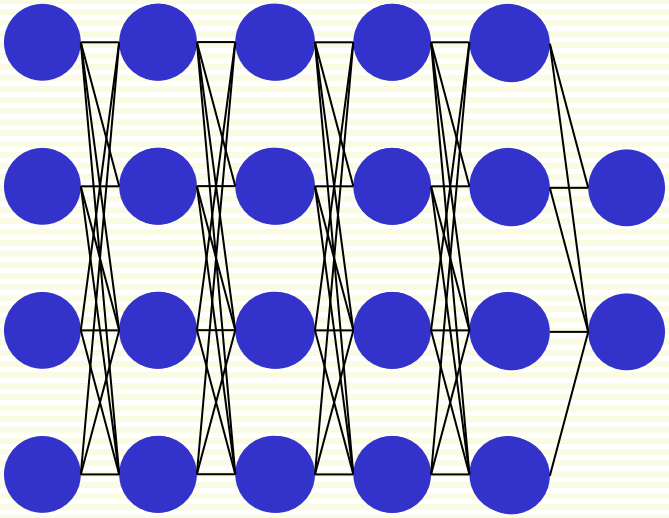
Shallow vs. Deep Architecture

- How many layers should we choose?

Shallow network



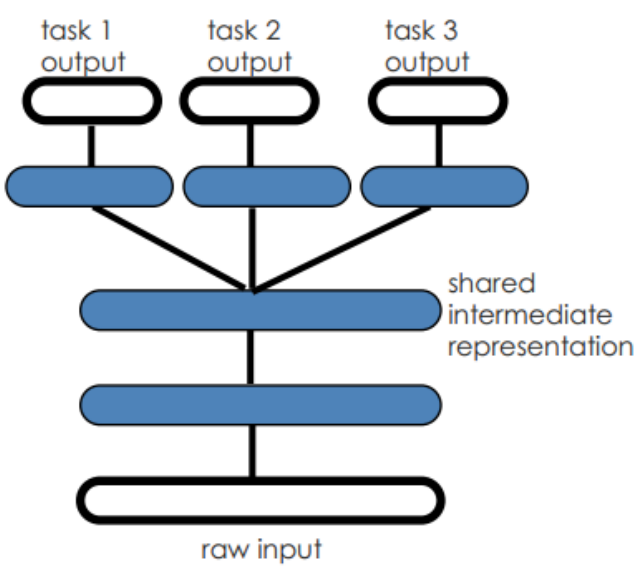
Deep network



- 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

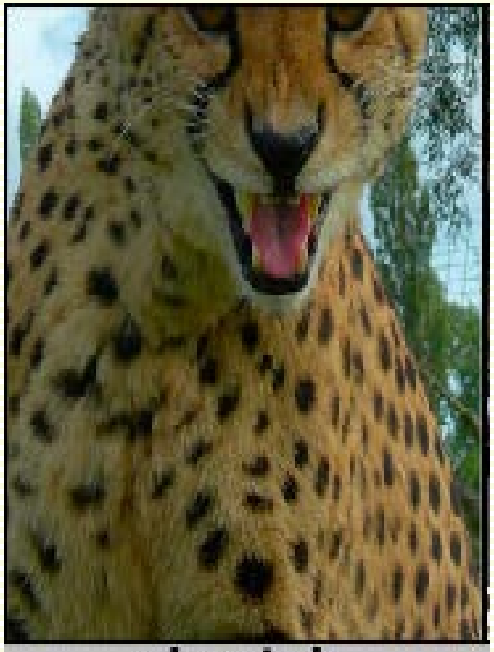
- Greedy layer-wise training avoids many of the problems of trying to train a deep net in a supervised fashion
  - Each layer gets full learning focus in its turn since it is the only current "top" layer
  - Can take advantage of the unlabeled data
  - When you finally tune the entire network with supervised 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

Neural Nets: Character Recognition

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

Yann LeCun et. al.

ConvNet on Image Classification



cheetah

cheetah
leopard
snow leopard
Egyptian cat

bullet train

bullet train
passenger car
subway train
electric locomotive

hand glass

scissors
hand glass
frying pan
stethoscope

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