CS4442/9542b
Artificial Intelligence II
prof. Olga Veksler

Lecture 7

Machine Learning

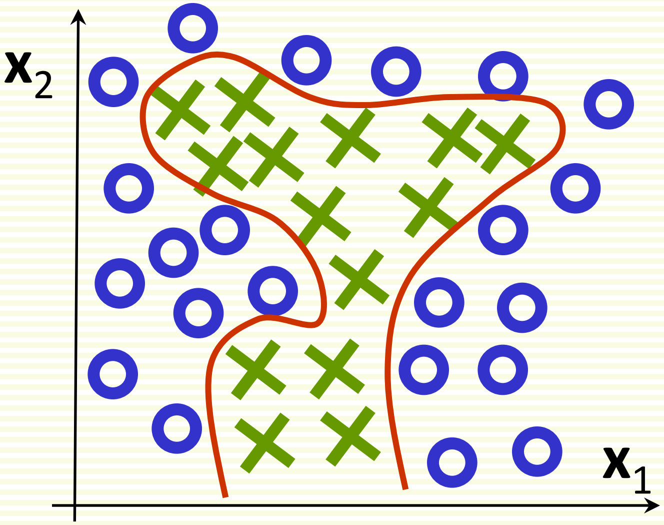
Neural Networks

Many presentation Ideas are due to Andrew NG

Outline

- Motivation
  - Non linear discriminant functions
- Introduction to Neural Networks
    - Inspiration from Biology
    - History
- Perceptron: 1 layer Neural Network
- Multilayer Neural Networks
  - also called Artificial Neural Network (ANN), ,perceptron (MLP), Feedforward Neural Network
- Training Neural Networks
  - backpropagation algorithm
  - practical tips for training

Need for Non-Linear Discriminant



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

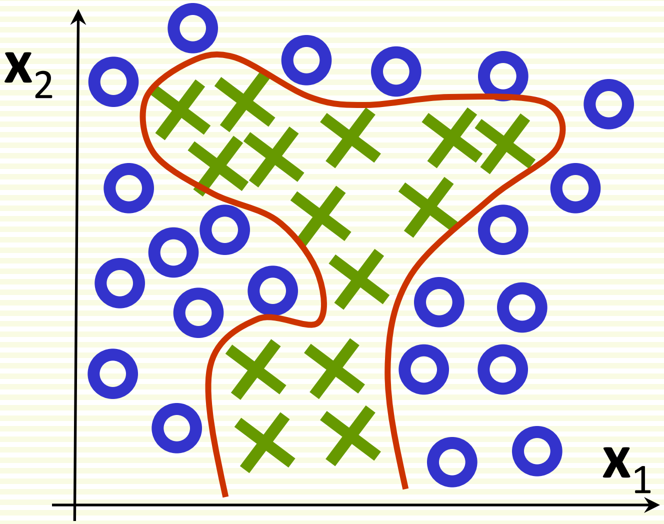
$$\mathbf{g}(\mathbf{x}) = \mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2+$$
$$+ \mathbf{w}_{12}\mathbf{x}_1\mathbf{x}_2 + \mathbf{w}_{11}\mathbf{x}_1{}^2 +\mathbf{w}_{22}\mathbf{x}_2{}^2 +$$
$$+ \mathbf{w}_{111}\mathbf{x}_1{}^3+ \mathbf{w}_{112}\mathbf{x}_1{}^2\mathbf{x}_2 +\mathbf{w}_{122}\mathbf{x}_1\mathbf{x}_2{}^2 + \mathbf{w}_{222}\mathbf{x}_2{}^3 +$$

+ even more terms of degree **4**

+ super many terms of degree **k**

- For **n** features, there O($\mathbf{n}^k$) 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 $\mathbf{g}_{NN}(\mathbf{x})$
- Can carve out arbitrarily complex decision boundaries without requiring so many terms as polynomial functions
- Neural Nets were inspired by research in how human brain works
- But also proved to be quite successful in practice
- Are used nowadays successfully for a wide variety of applications
  - took some time to get them to work

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

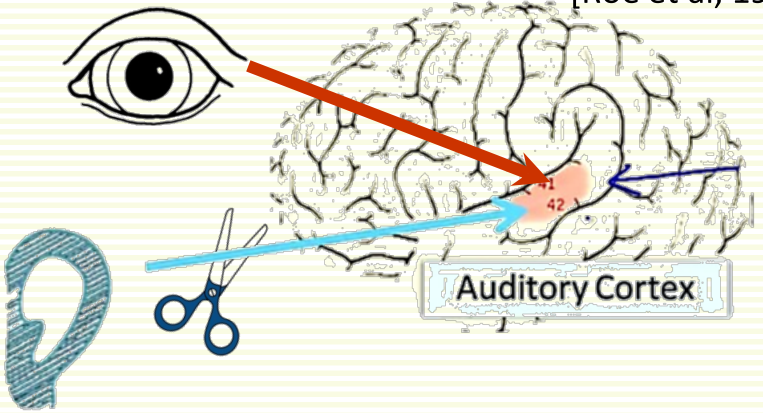
- huge number of parallel but relatively slow and unreliable processors
- not perfectly precise, not perfectly reliable
- evolved (in a large part) for pattern recognition
- learns to solve various PR problems

seek inspiration for classification from human brain

One Learning Algorithm Hypothesis

[Roe et al, 1992]

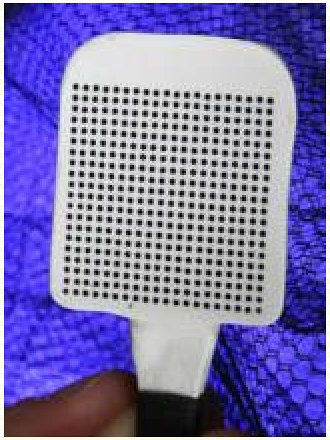
- Brain does many different things
- Seems like it runs many different "programs"
- Seems we have to write tons of different programs to mimic brain



- 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



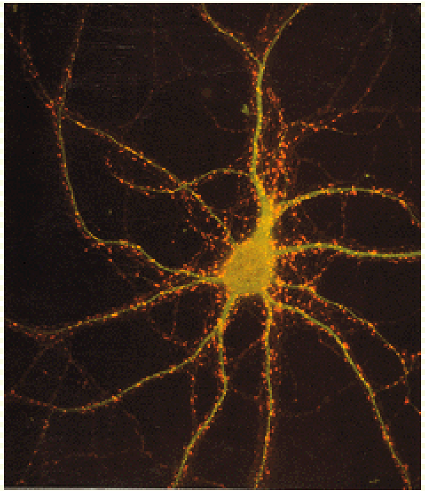
tongue array sensor

One Learning Algorithm Hypothesis

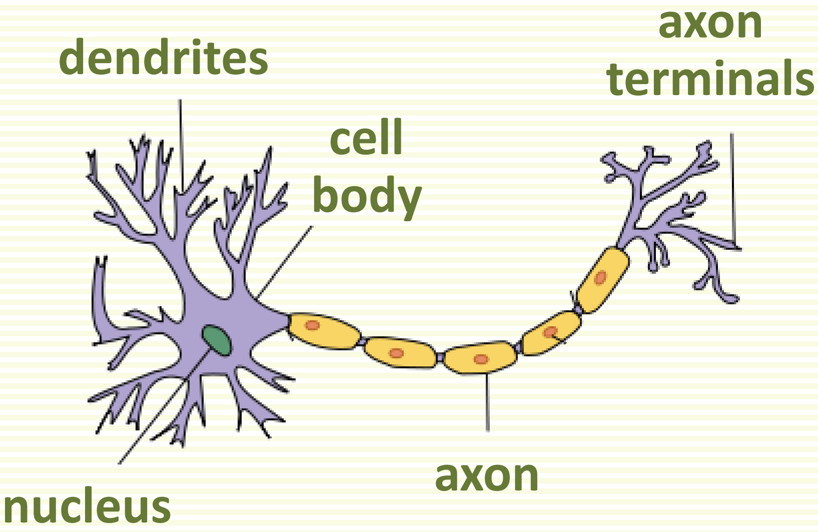
- Experimental evidence that we can plug any sensor to any part of the brain, and brain can learn how to deal with it
- Since the same physical piece of brain tissue can process sight, sound, etc.
- Maybe there is one learning algorithm can process sight, sound, etc.
- Maybe we need to figure out and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



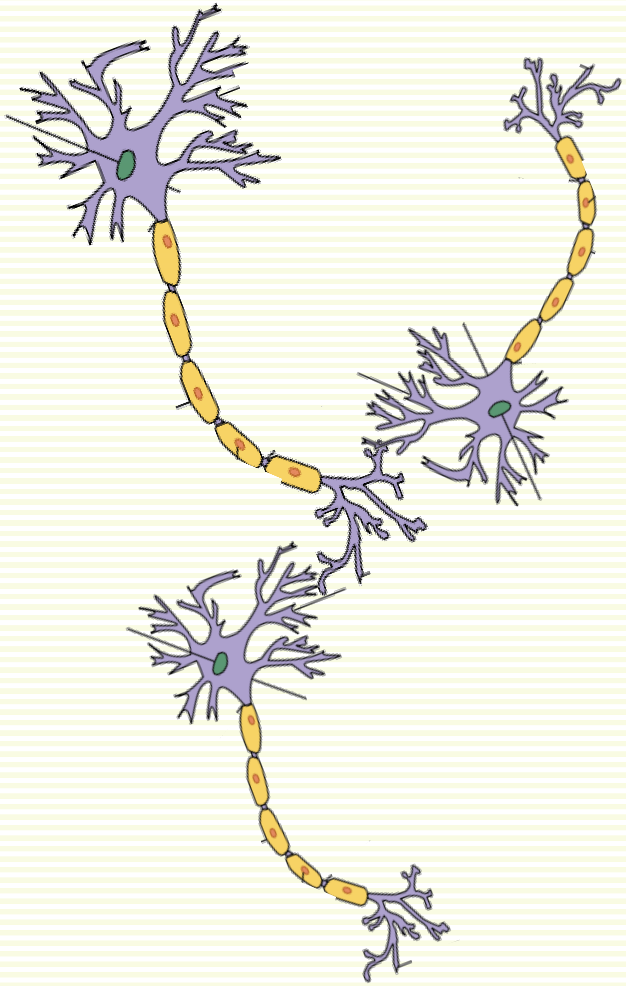
Neuron: Main Components



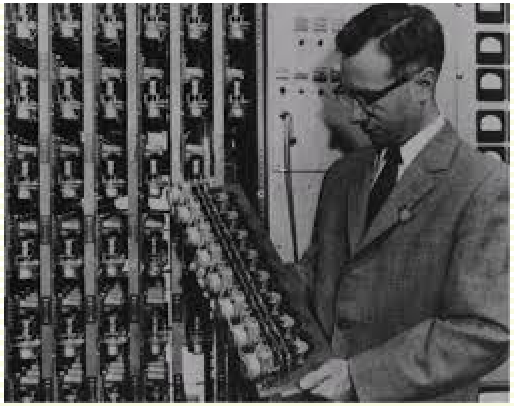
- **cell body**
  - computational unit
- **dendrites**
  - “input wires”, receive inputs from other neurons
  - a neuron may have thousands of dendrites, usually short
- **axon**
  - “output wire”, sends signal to other neurons
  - single long structure (up to 1 meter)
  - splits in possibly thousands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



ANN History: First Successes



- 1958, F. Rosenblatt, Cornell University
  - Perceptron, oldest neural network
    - studied in lecture on linear classifiers
  - Algorithm to train the Perceptron
  - Built in hardware to recognize digits images
  - Proved convergence in linearly separable case
  - Early success lead to a lot of claims which were not fulfilled
  - New York Times reports that perceptron is "*the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence*."

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

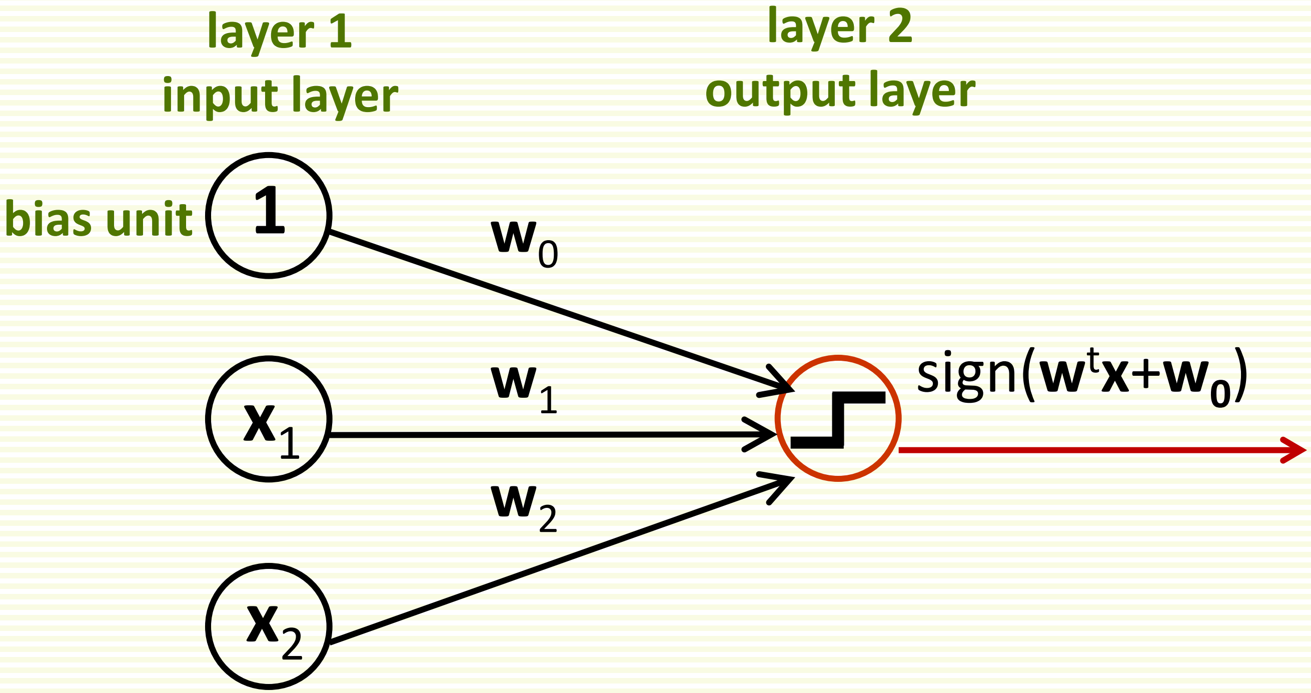
ANN History: Revival & Stagnation (Again)

- Revival of ANN in early 1980
- 1986, (re)discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows training a MLP
- Many examples of mulitlayer Neural Networks appear
- 1998, Convolutional network (convnet) by Y. Lecun for digit recognition, very successful
- 1990's: research in NN move slowly again
  - Networks with multiple layers are hard to train well (except convnet for digit recognition)
  - SVM becomes popular, works better

ANN History: Deep Learning Age

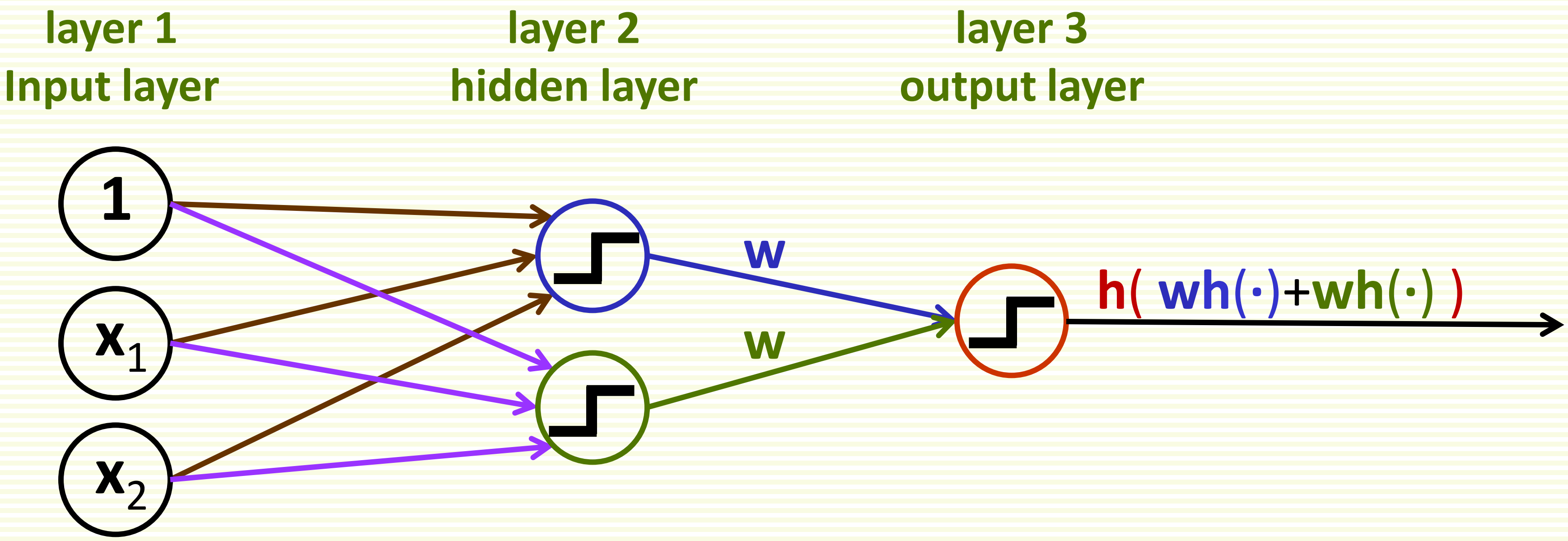
- Deep networks are inspired by brain architecture
- Until now, no success at training them, except convnet
- 2006-now: deep networks are trained successfully
  - massive training data becomes available
  - better hardware: fast training on GPU
  - better training algorithms for network training when there are many hidden layers
    - unsupervised learning of features, helps when training data is limited
- Break through papers
  - Hinton, G. E, Osindero, S., and Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. Neural Computation, 18:1527-1554.
  - Bengio, Y., Lamblin, P., Popovici, P., Larochelle, H. (2007). Greedy Layer-Wise Training of Deep Networks, Advances in Neural Information Processing Systems 19
- Industry: Facebook, Google, Microsoft, etc.

Perceptron: 1 Layer Neural Network



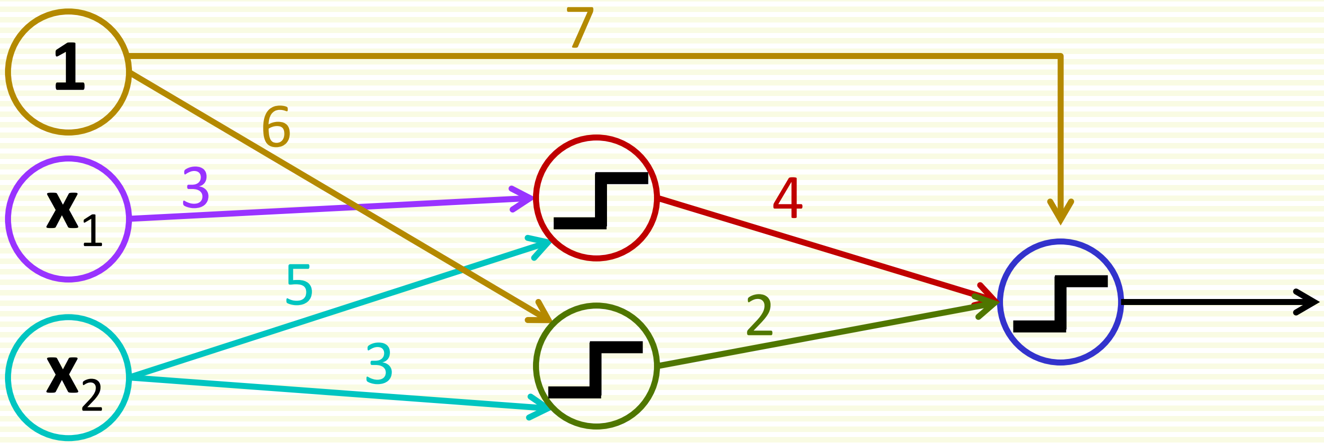
- Linear classifier $\mathbf{f}(\mathbf{x})$ = sign($\mathbf{w}^t\mathbf{x}+\mathbf{w}_0$) is a single neuron "net"
- Input layer units emits features, except bias emits "1"
- Output layer unit applies **h**(t) = **sign**(t)
- **h**(t) is also called an *activation function*

Multilayer Neural Network



- First hidden unit outputs $\mathbf{h}(\mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2)$
- Second hidden unit outputs $\mathbf{h}(\mathbf{w}_0+\mathbf{w}_1\mathbf{x}_1+\mathbf{w}_2\mathbf{x}_2)$
- Network implements classifier $\mathbf{f}(\mathbf{x}) = \mathbf{h}(\mathbf{wh}(\cdot)+\mathbf{wh}(\cdot))$
- More complex boundaries than Perceptron

Multilayer Neural Network: Small Example



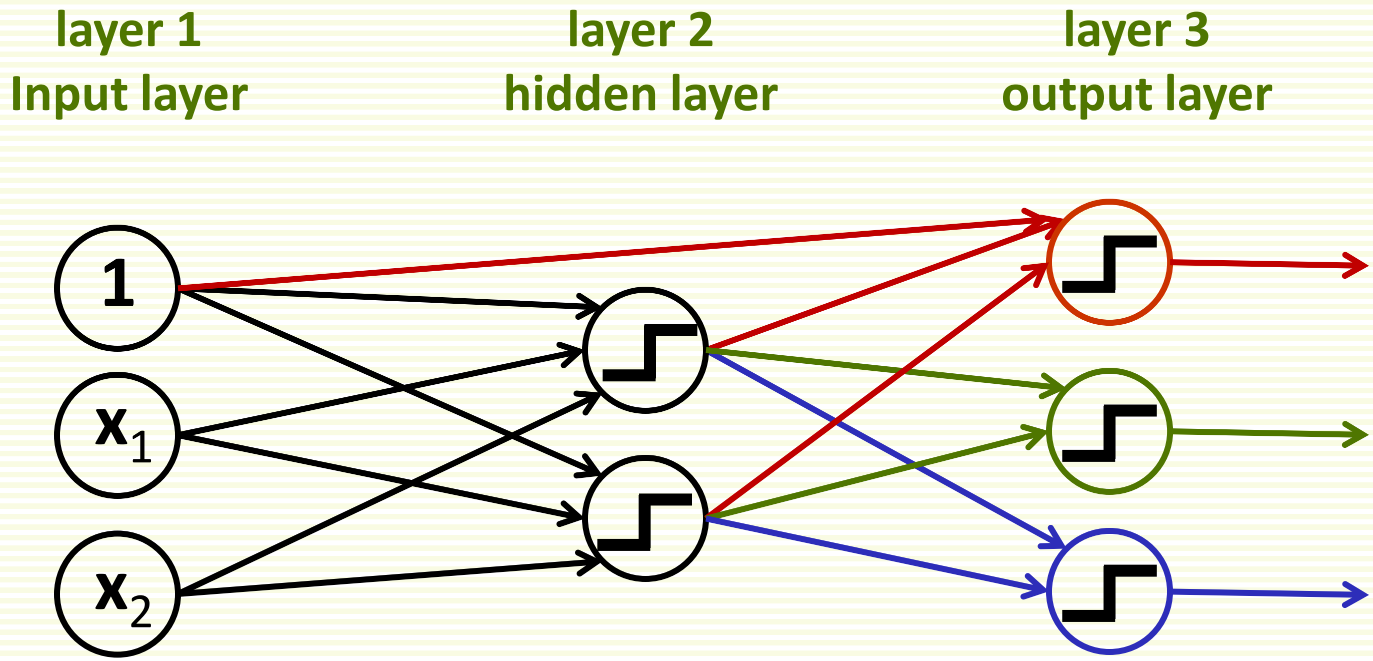
- Implements classifier

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

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

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

Multilayer NN : Multiple Classes



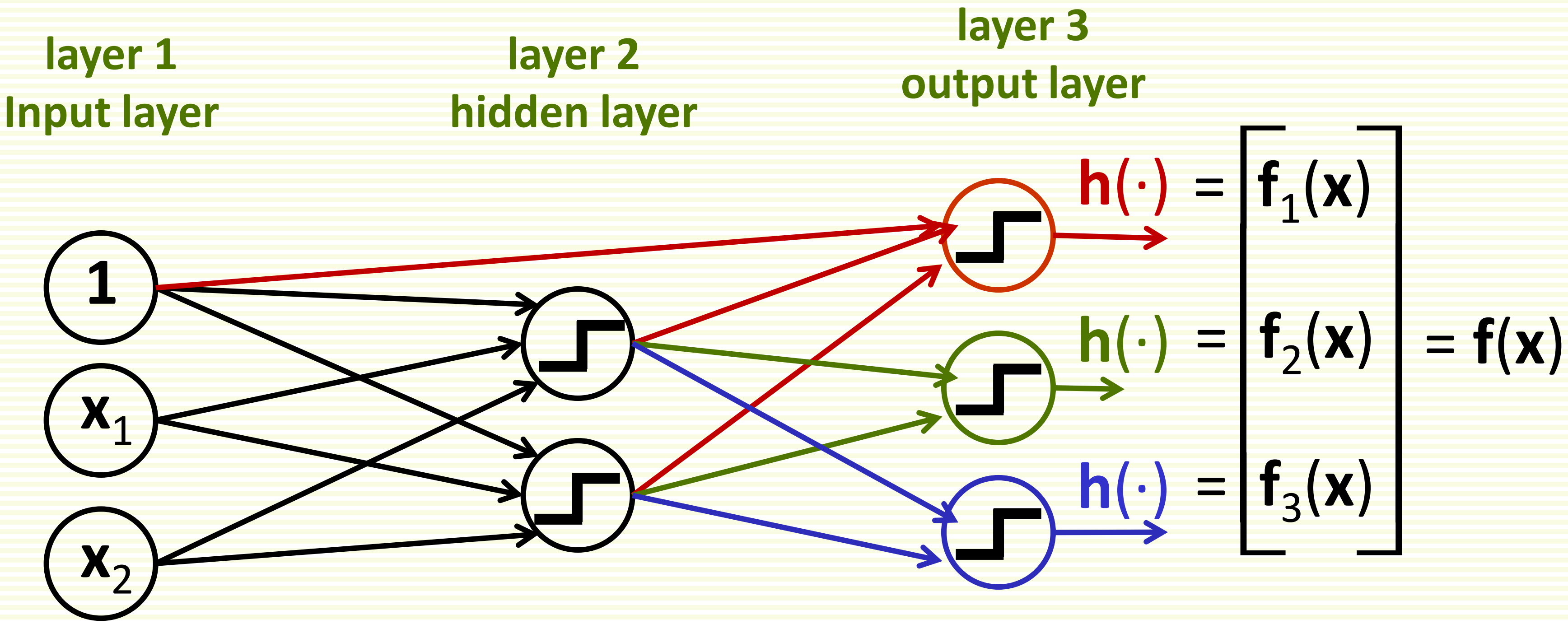
layer 1
Input layer

layer 2
hidden layer

layer 3
output layer

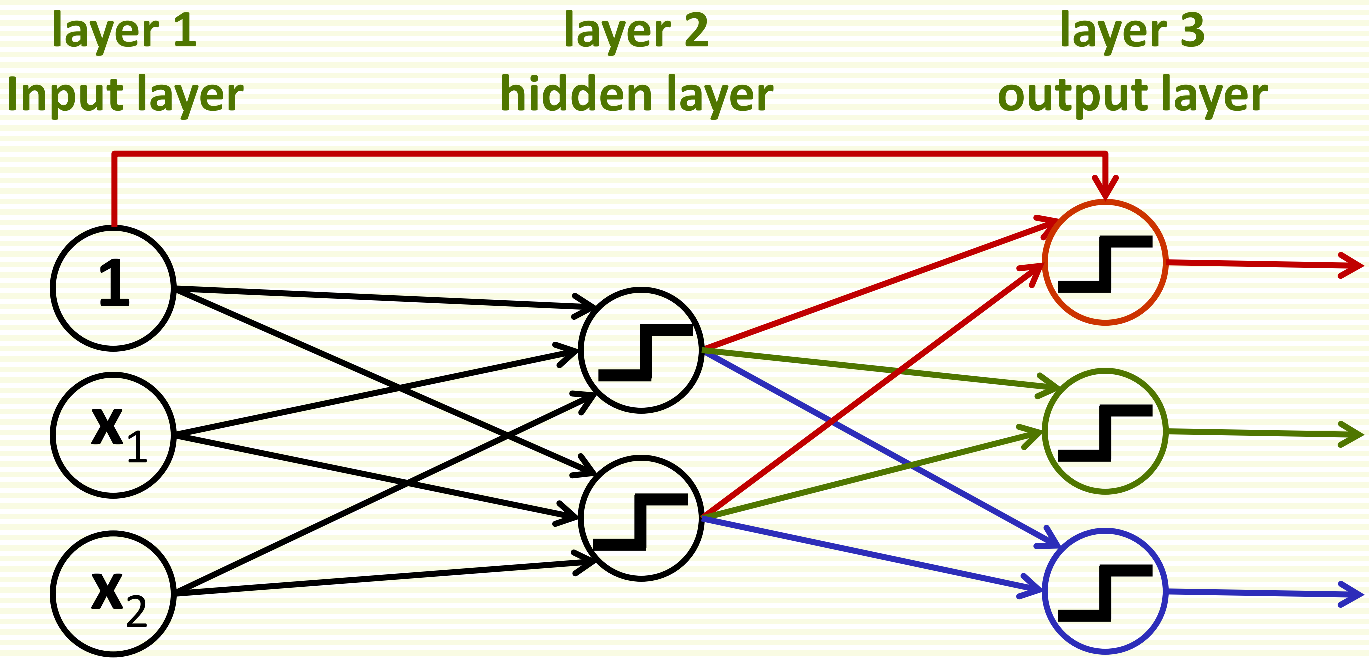
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, can draw in layer 1, or each layer has one

Multilayer NN: General Structure


layer 1
Input layer
layer 2
hidden layer
layer 3
output layer
1
x_1
x_2
$h(\cdot) = f_1(\mathbf{x})$
$h(\cdot) = f_2(\mathbf{x})$
$h(\cdot) = f_3(\mathbf{x})$
$= \mathbf{f}(\mathbf{x})$

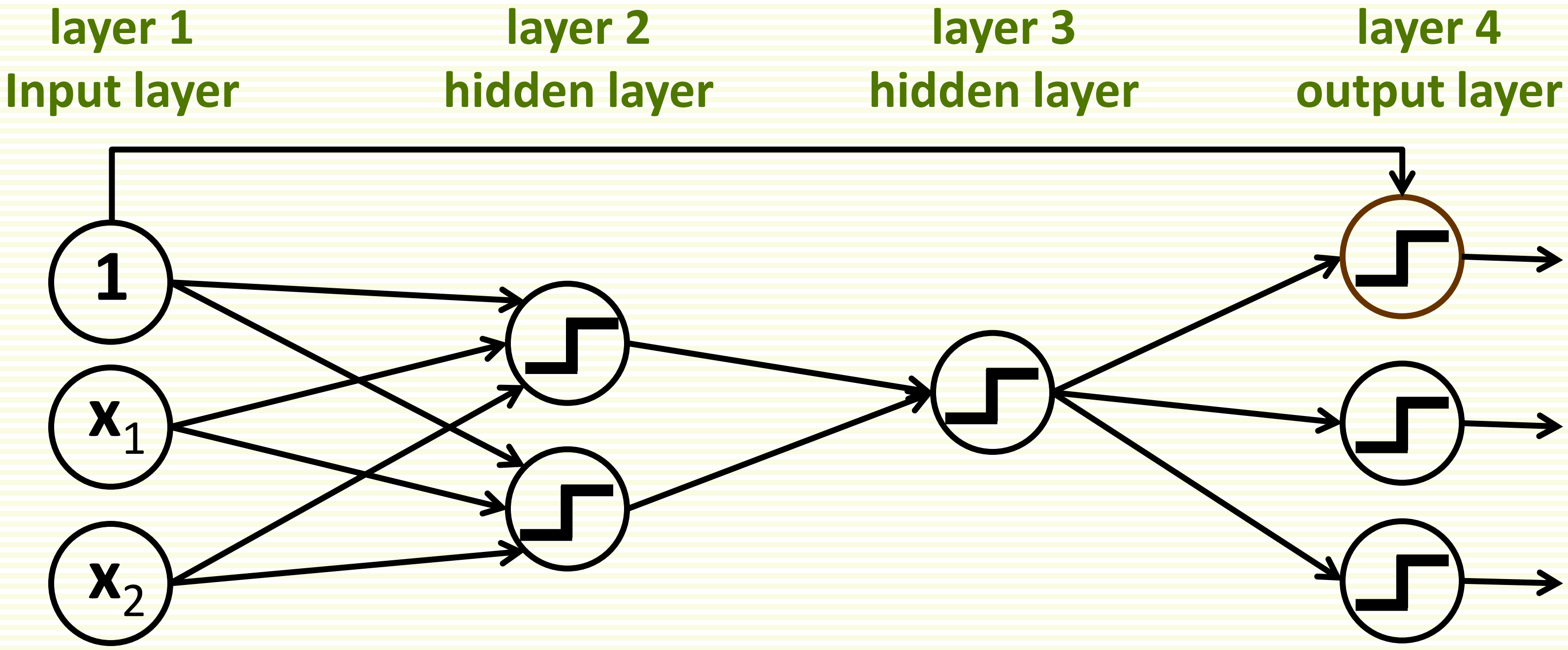
- $\mathbf{f}(\mathbf{x})$ is multi-dimensional
- Classification
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$ is largest, decide class 3

Multilayer NN : 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

Multilayer NN : 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

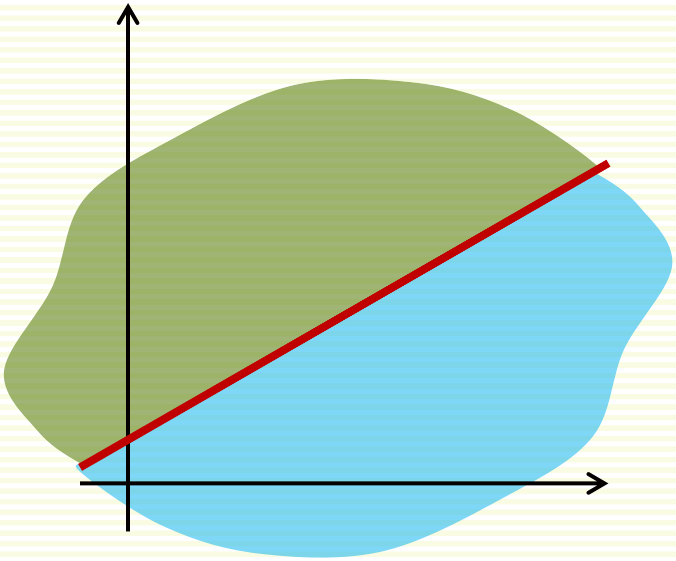
Multilayer NN : Overview

- NN 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 rather than write out the functions
- To train NN, just as before
    - formulate per-sample loss function **L**(**w**)
    - optimize it with gradient descent
        - lots of heuristics to get gradient descent work well enough

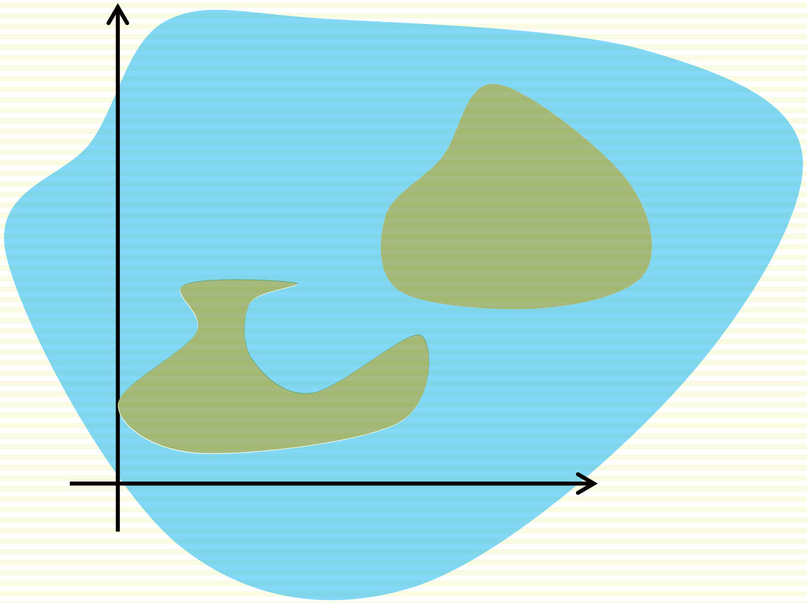
Multilayer NN : Expressive Power

- 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
- More of theoretical than practical interest
  - do not know the desired function in the first place, our goal is to learn it through the samples
  - but this result gives confidence that we are on the right track
    - multilayer NN is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron

Multilayer NN: Decision Boundaries



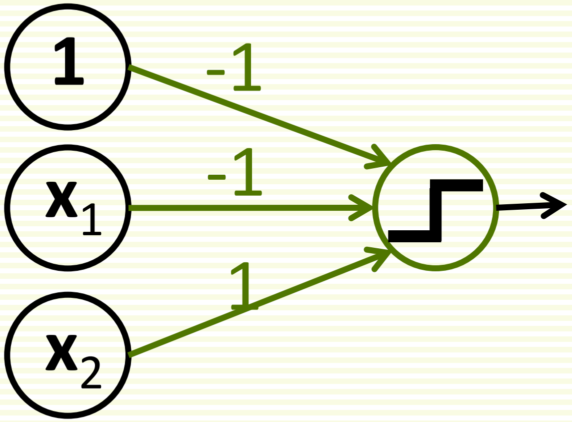
- Perceptron (single layer neural net)

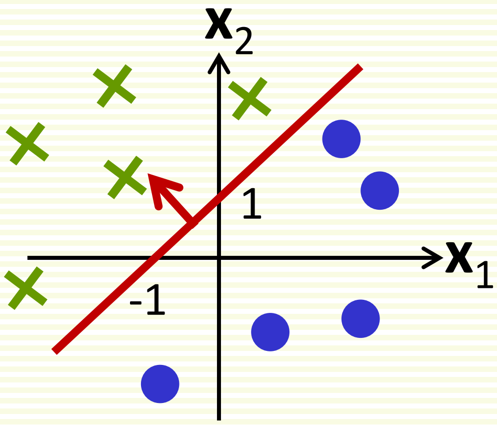


- Multilayer NN
- Arbitrarily complex decision regions
- Even not contiguous

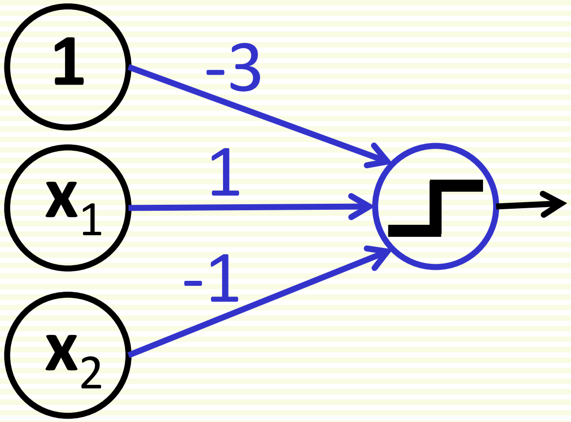
Multilayer NN : Nonlinear Boundary Example

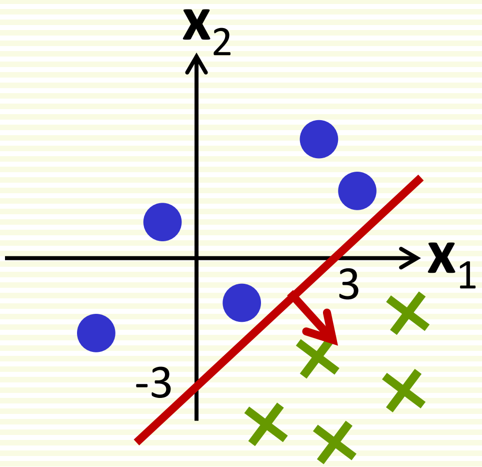
$-\mathbf{x}_1 + \mathbf{x}_2 - 1 > 0 \Rightarrow \text{class } 1$





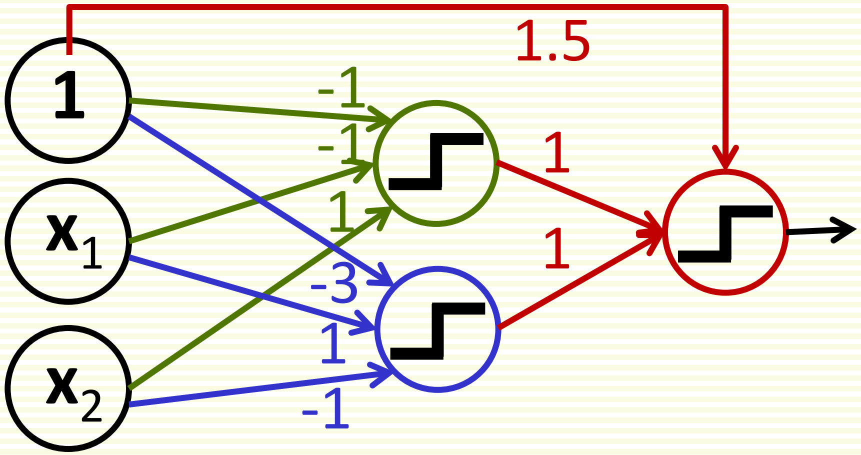
$\mathbf{x}_1 - \mathbf{x}_2 - 3 > 0 \Rightarrow \text{class } 1$

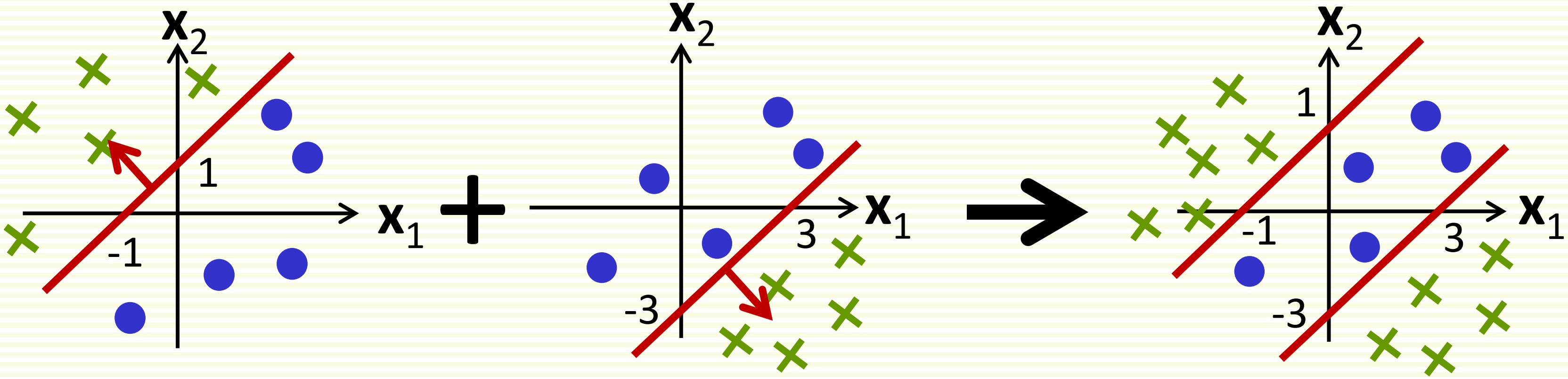




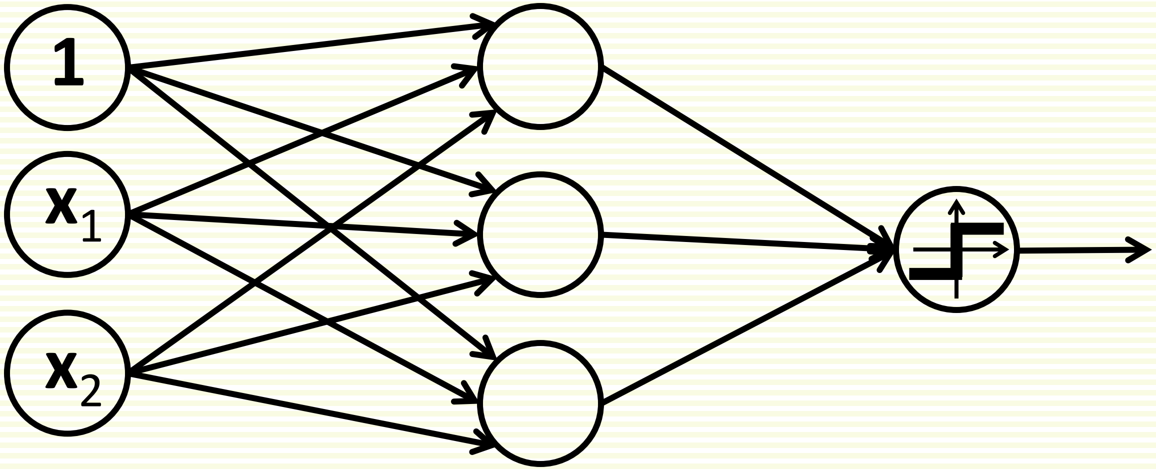
Multilayer NN : Nonlinear Boundary Example

- Combine two Perceptrons into a 3 layer NN



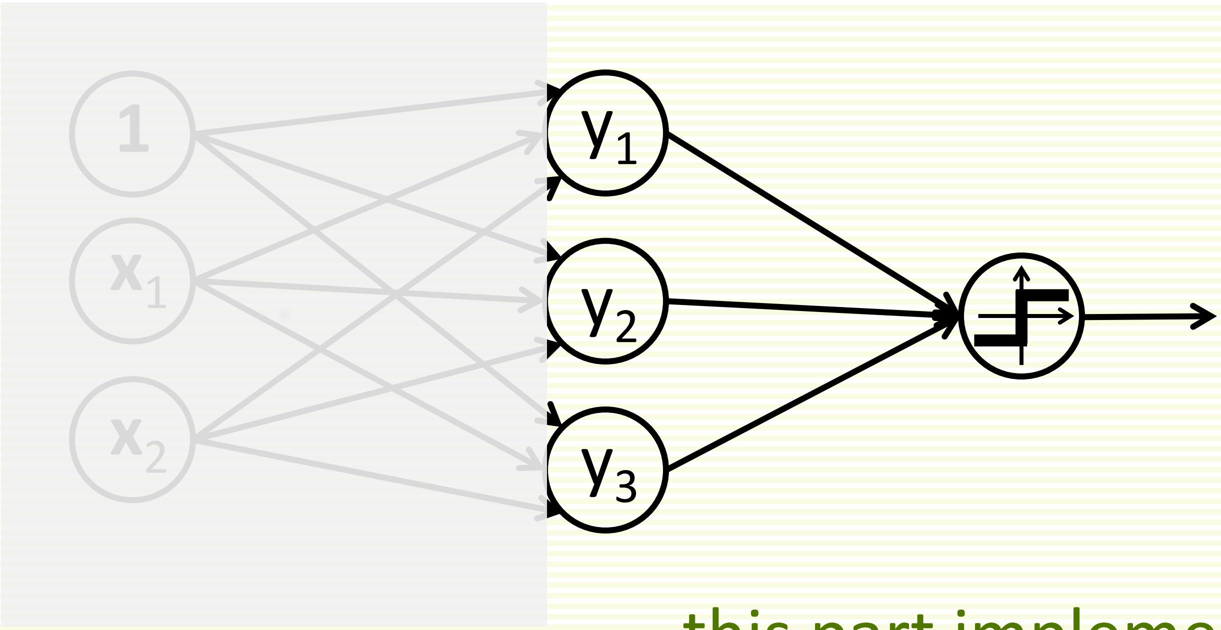


Multilayer NN as Non-Linear Feature Mapping



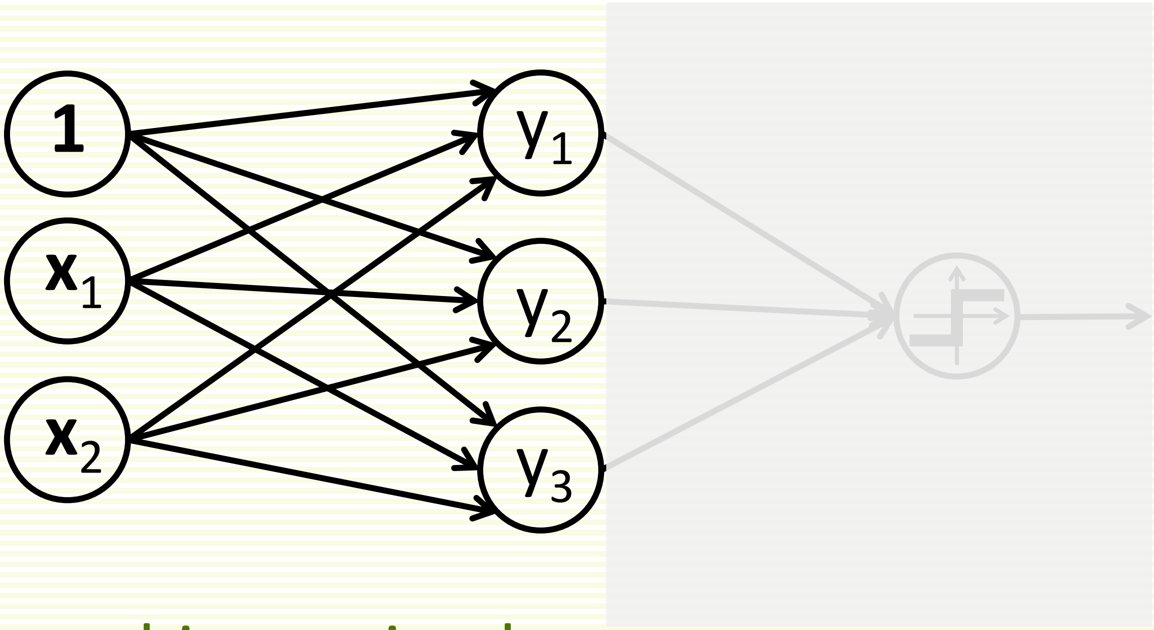
- Interpretation
  - 1 hidden layer maps input features to new features
  - next layer then applies linear classifier to the new features

Multilayer NN as Non-Linear Feature Mapping



this part implements Perceptron (liner classifier)

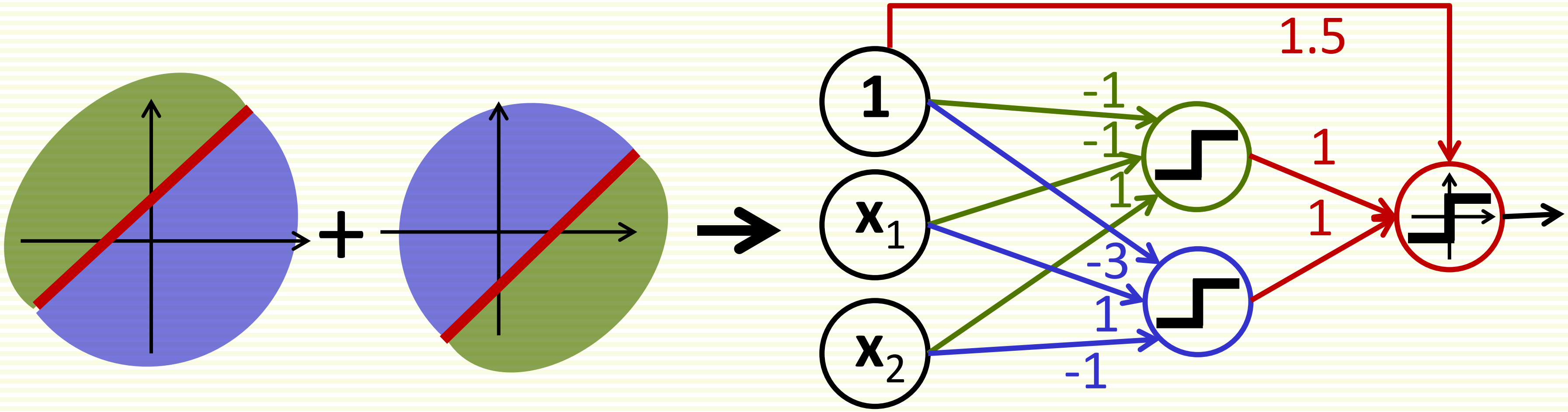
Multilayer NN as Non-Linear Feature Mapping

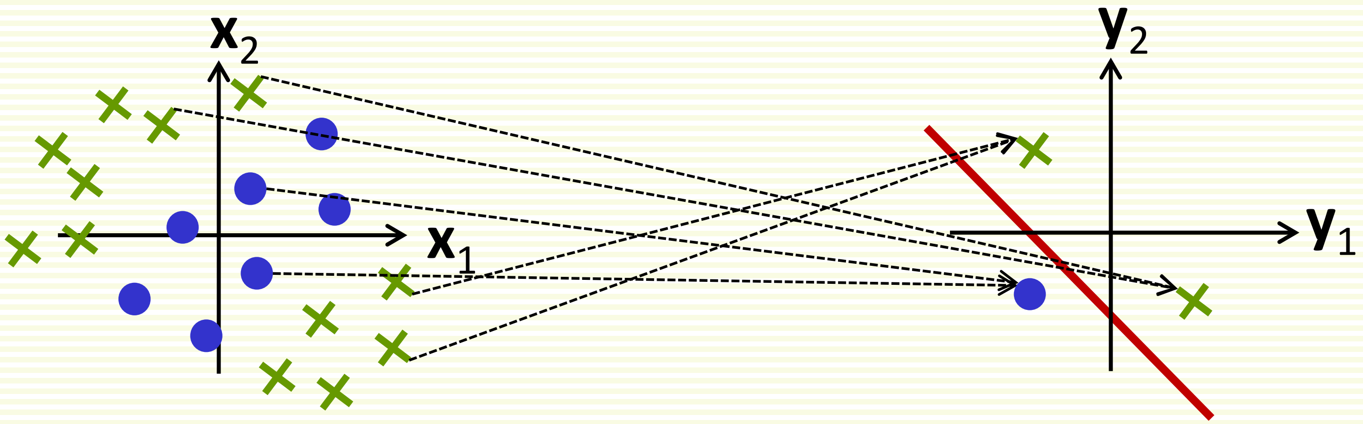


this part implements mapping to new features **y**

Multilayer NN as Non-Linear Feature Mapping

- Consider 3 layer NN example we saw previously:



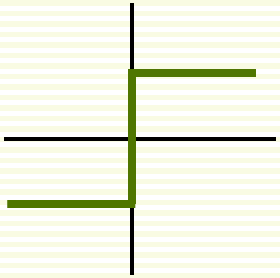


non linearly separable in the original feature space

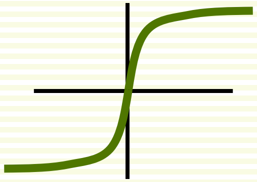
linearly separable in the new feature space

Multi Layer NN: Activation Function

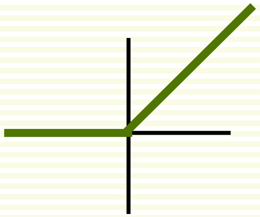
- **h**() = **sign**() does not work for gradient descent



- Can use tanh or sigmoid function



- Rectified Linear (ReLu) popular recently
  - gradients do not saturate for positive half-interval
  - but have to be careful with learning rate, otherwise many units can become “dead”, i.e. always output 0



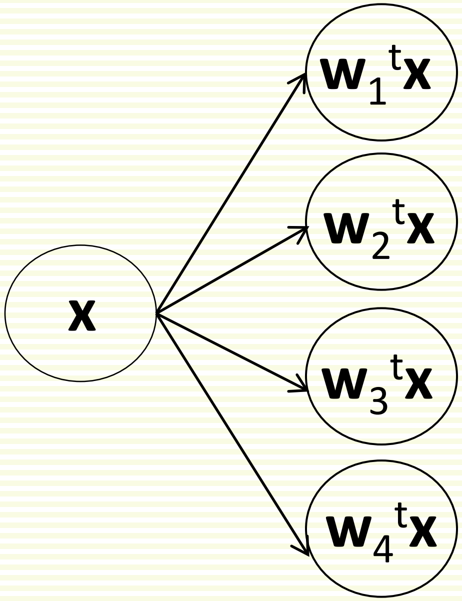
Multilayer NN: Modes of Operation

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

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

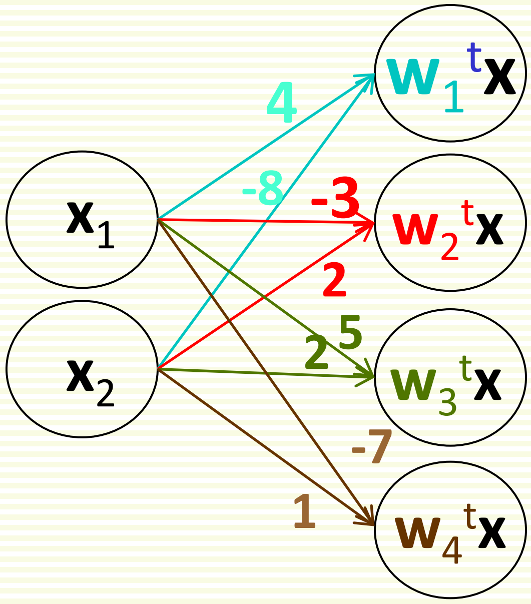
Multilayer NN: Matrix Notation

- Recall matrix notation for linear classifier



$$\begin{array}{c} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \mathbf{w}_3 \\ \mathbf{w}_4 \end{array} \underset{\mathbf{W}}{\begin{bmatrix} 2 & 4 & -7 \\ 9 & -3 & 2 \\ 4 & 5 & 2 \\ 2 & -7 & 1 \end{bmatrix}} \underset{\mathbf{x}}{\begin{bmatrix} 1 \\ 7 \\ 4 \end{bmatrix}} = \underset{\mathbf{Wx}}{\begin{bmatrix} 2 \\ -4 \\ 47 \\ -43 \end{bmatrix}}$$

- Full picture, ignoring bias weights

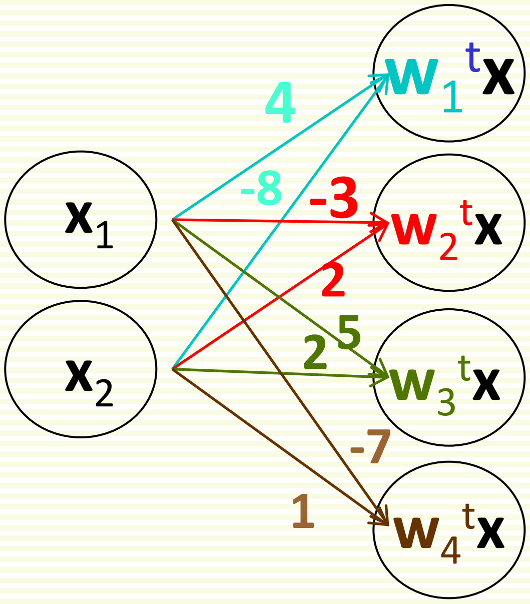


Wx

- This is subpart of neural network
- Need to add activation function

Multilayer NN: Matrix Notation

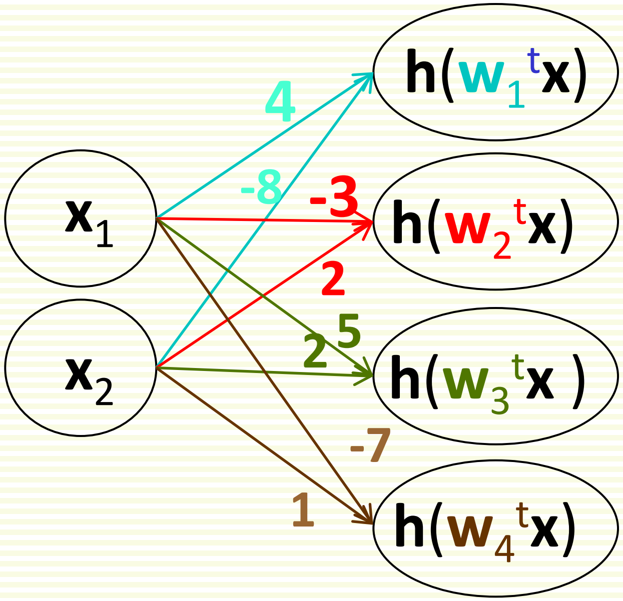
- Full picture



$\mathbf{Wx}$

- This is subpart of neural network

- Need activation function **h** in Neural Network

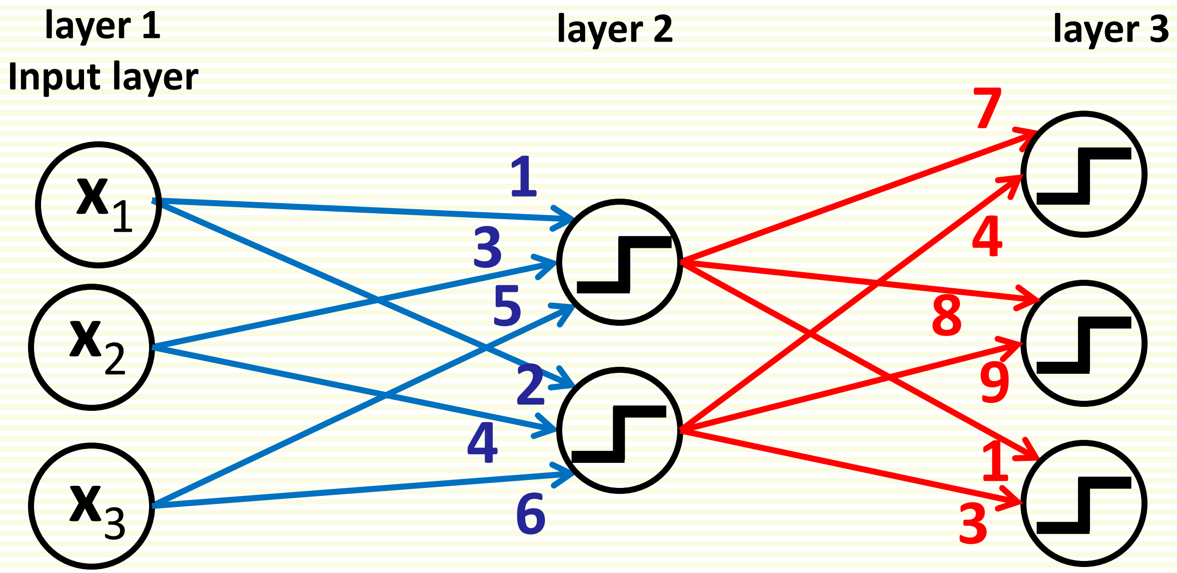


$\mathbf{h}(\mathbf{Wx})$

$$\mathbf{h} = \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix}$$

Multilayer NN: Matrix Notation

- Use similar notation for NN



$$\mathbf{h} = \mathbf{h}(\mathbf{W}\mathbf{x})$$

$$\mathbf{h} = \begin{bmatrix} * \\ * \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

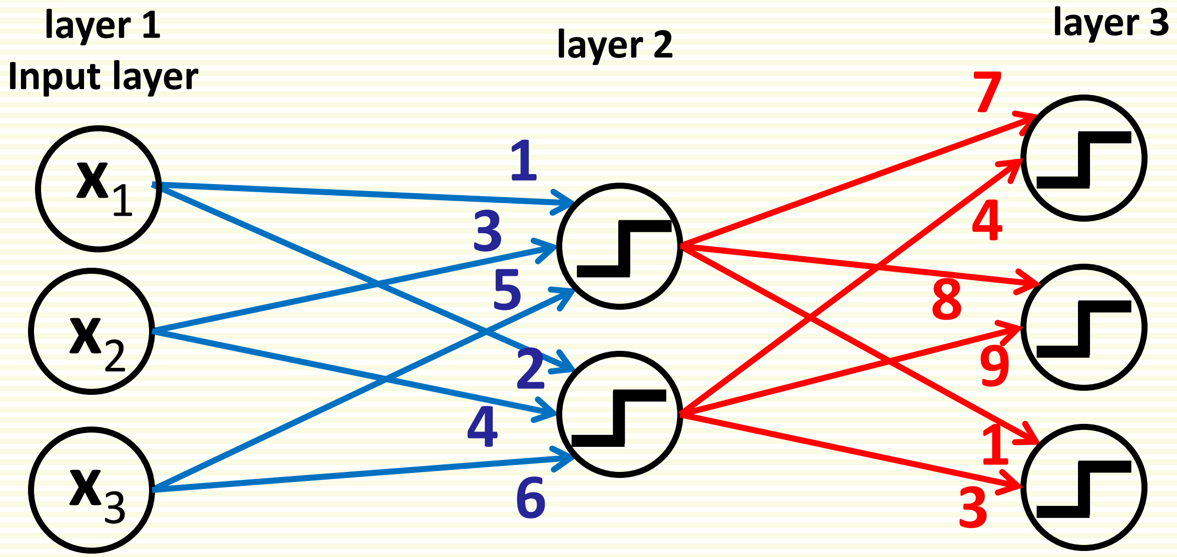
$$\mathbf{h} = \mathbf{h}(\mathbf{W}\mathbf{h})$$

$$\mathbf{h} = \begin{bmatrix} * \\ * \\ * \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix}$$

Multilayer NN: Matrix Notation

- Instead of color, use superscripts



$$h^1 = h(W^1 x)$$

$$h^2 = h(W^2 h^1)$$

$$h^1 = \begin{bmatrix} * \\ * \end{bmatrix}$$

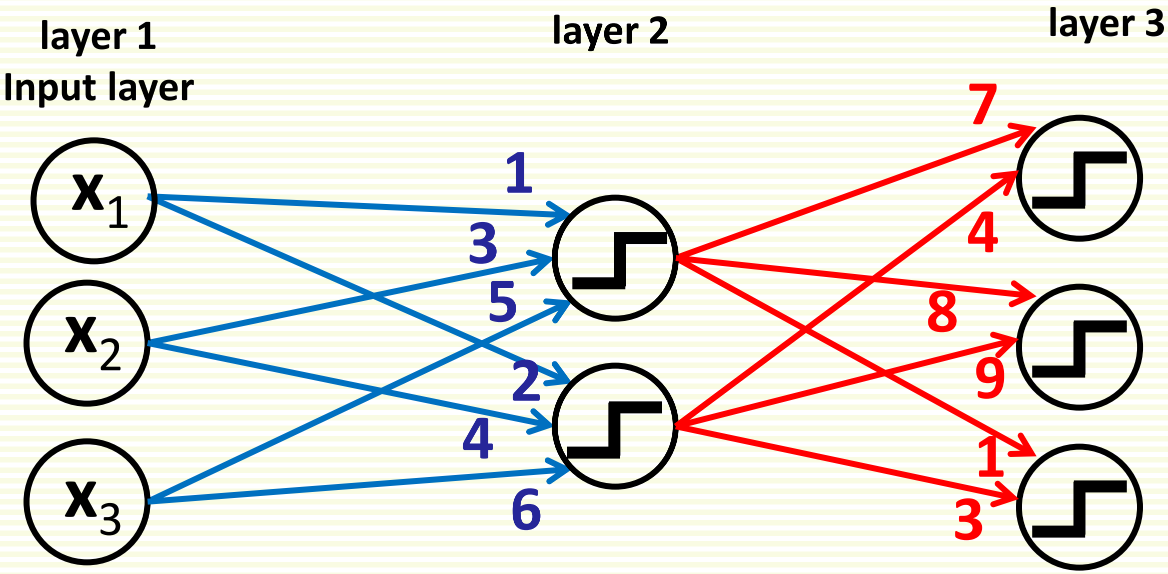
$$h^2 = \begin{bmatrix} * \\ * \\ * \end{bmatrix}$$

$$W^1 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix}$$

Multilayer NN: Matrix Notation

- Add bias weights, also as vectors

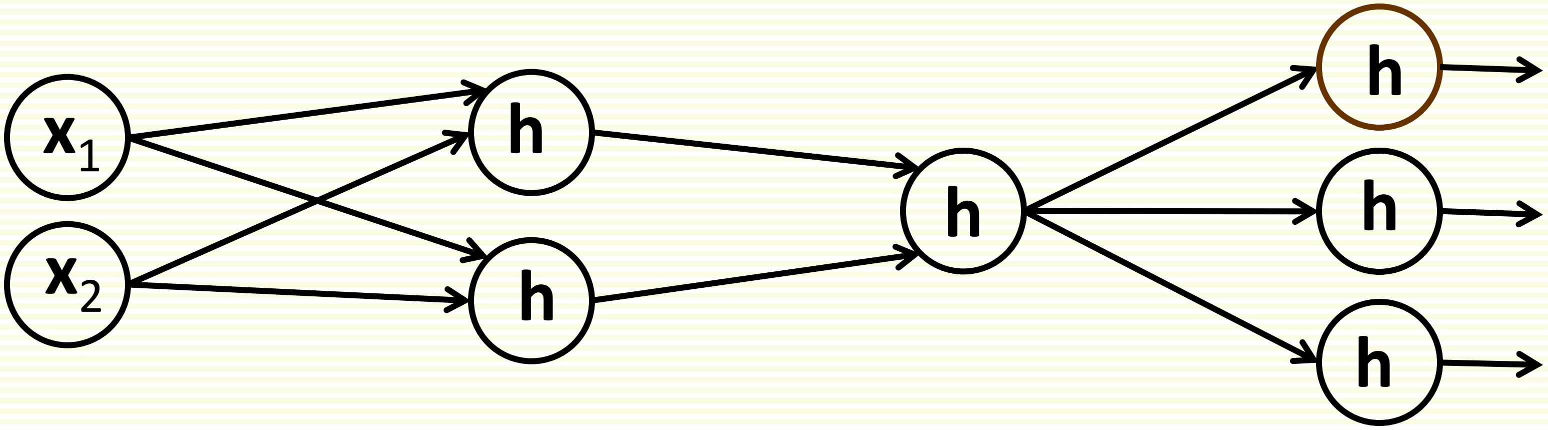


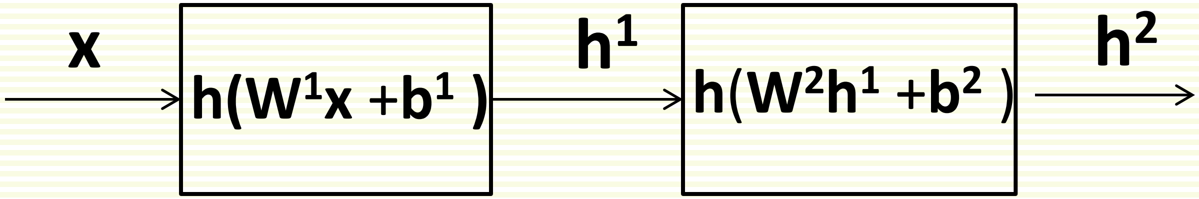
$$h^1 = h(W^1 x + b^1) \qquad h^2 = h(W^2 h^1 + b^2)$$

$$h^1 = \begin{bmatrix} * \\ * \end{bmatrix} \quad b^1 = \begin{bmatrix} * \\ * \end{bmatrix} \qquad h^2 = \begin{bmatrix} * \\ * \\ * \end{bmatrix} \quad b^2 = \begin{bmatrix} * \\ * \\ * \end{bmatrix}$$

$$W^1 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \qquad W^2 = \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix}$$

Multilayer NN: Vector Notation for Next Layer

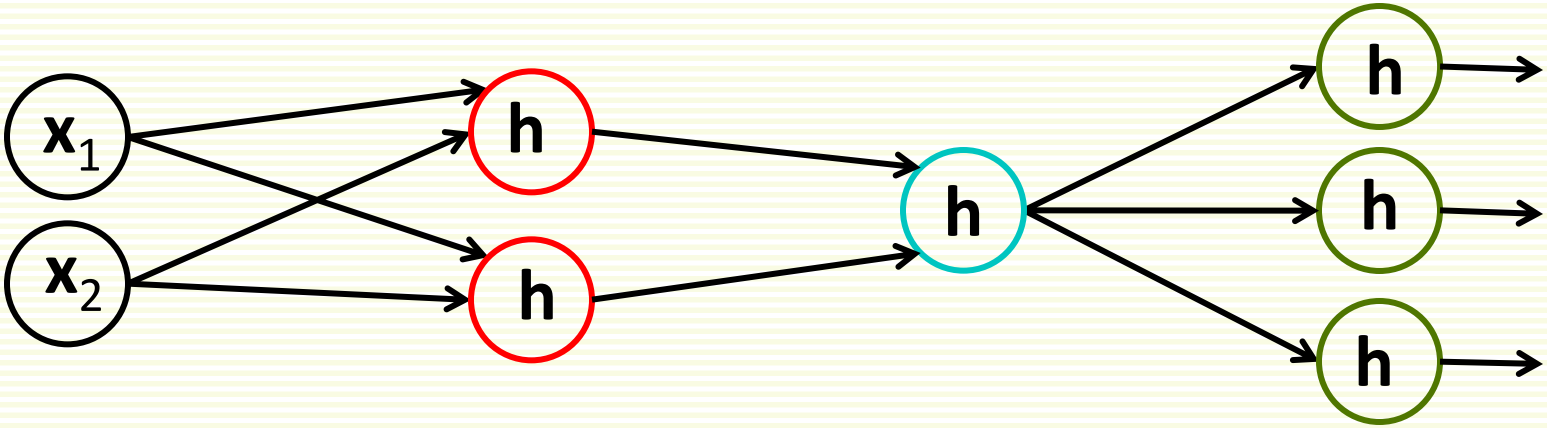


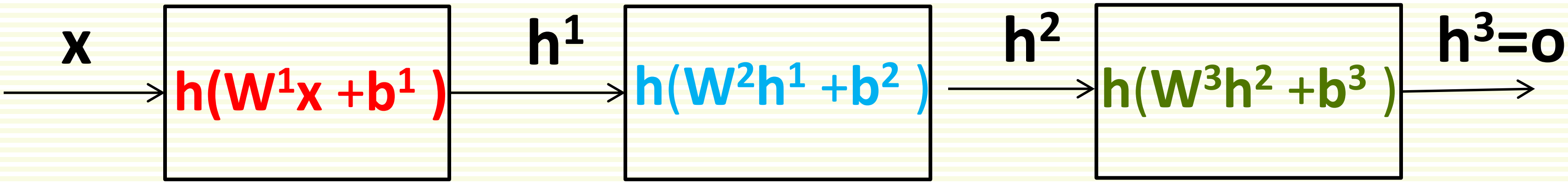


$\mathbf{x}$ → $\mathbf{h(W^1x + b^1)}$ → $\mathbf{h^1}$ → $\mathbf{h(W^2h^1 + b^2)}$ → $\mathbf{h^2}$

- $\mathbf{W^2}$ 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 weight 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

Multilayer NN: Vector Notation, all Layers





x → $h(W^1x + b^1)$ → h^1 → $h(W^2h^1 + b^2)$ → h^2 → $h(W^3h^2 + b^3)$ → $h^3 = o$

- h^3 is vector from the output layer and it is also called **f(x,W)**
- $h^3 = h(W^3h^2 + b^3)$

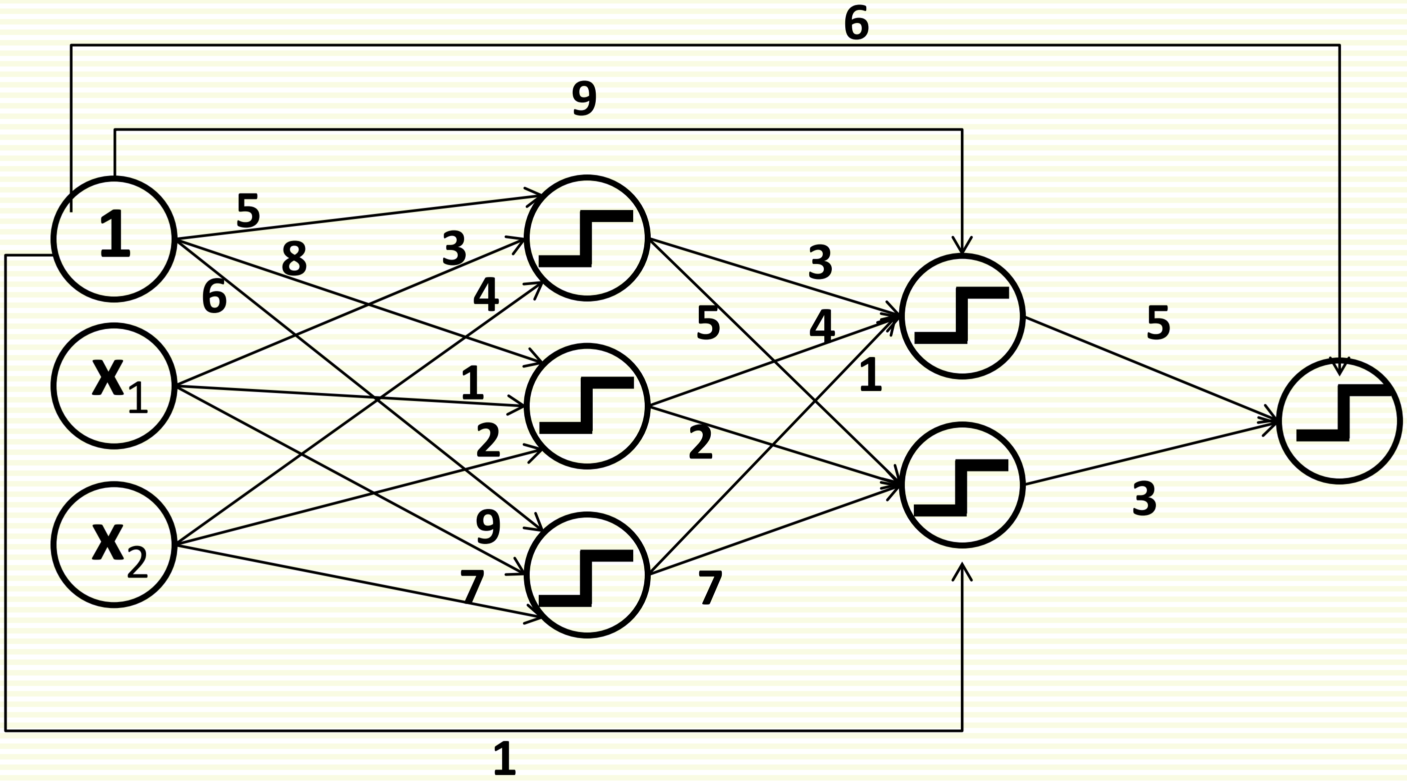
  $= h(W^3h(W^2h^1 + b^2) + b^3)$

  $= h(W^3h(W^2h(W^1x + b^1) + b^2) + b^3)$

Vector Notation, Example

- Assuming sign activation function, draw a NN given by

$$\mathbf{W}^1 = \begin{bmatrix} 3 & 4 \\ 1 & 2 \\ 9 & 7 \end{bmatrix} \quad \mathbf{b}^1 = \begin{bmatrix} 5 \\ 8 \\ 6 \end{bmatrix} \quad \mathbf{W}^2 = \begin{bmatrix} 3 & 4 & 1 \\ 5 & 2 & 7 \end{bmatrix} \quad \mathbf{b}^2 = \begin{bmatrix} 9 \\ 1 \end{bmatrix} \quad \mathbf{W}^3 = \begin{bmatrix} 5 & 3 \end{bmatrix} \quad \mathbf{b}^3 = \begin{bmatrix} 6 \end{bmatrix}$$



Multilayer NN: Output Representation

- Output of NN is a vector
- As before, if **x**^{i} be sample of class **k**, its label is

$$\mathbf{y}^i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \longleftarrow \text{row } \mathbf{k} \qquad \mathbf{f}(\mathbf{x}^i, \mathbf{W}) = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \longleftarrow \text{row } \mathbf{k}$$

wish to get this output

Training NN: Squared Difference Loss

- Wish to minimize difference between $\mathbf{y}^i$ and $\mathbf{f}(\mathbf{x}^i)$
- Let **W** be all weights (all matrices $\mathbf{W}^t$ and bias vectors $\mathbf{b}^t$)
- With squared difference **loss**
- Squared loss on one example $\mathbf{x}^i$:

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

- For this example, squared loss is $3^2+2^2=13$

$$\mathbf{f}(\mathbf{x}^i,\mathbf{W})=\begin{bmatrix}3\\1\\-2\end{bmatrix}\qquad \mathbf{y}^i=\begin{bmatrix}0\\1\\0\end{bmatrix}$$

Training NN: Squared Difference Loss

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

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

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

- Gradient descent

```
Initialize $\mathbf{W}$ to random
choose $\boldsymbol{\varepsilon}, \alpha$
while $\alpha||\nabla \mathbf{L}(\mathbf{X},\mathbf{Y};\mathbf{W})|| > \boldsymbol{\varepsilon}$
    $\mathbf{W} = \mathbf{W} - \alpha\nabla \mathbf{L}(\mathbf{X},\mathbf{Y};\mathbf{W})$
```

Training NN: Softmax Loss

- Squared error loss is not recommended for classification
- Softmax is a better loss function, seen before in linear classifier
- First put the output of the network through soft-max

$$\mathbf{f}_{\mathbf{k}}(\mathbf{x}) = \frac{\exp(\mathbf{o}_{\mathbf{k}})}{\sum_{\mathbf{j}=1}^{\mathbf{m}} \exp(\mathbf{o}_{\mathbf{j}})}$$

$$\mathbf{o} = \begin{bmatrix} 0.6 \\ -1 \\ 5 \\ 8 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0.006 \\ 0.0001 \\ 0.047 \\ 0.94 \\ 0.17 \end{bmatrix} = \mathbf{f}(\mathbf{x}) = \text{sofmax}(\mathbf{o})$$

- Interpret $\mathbf{f}_{\mathbf{k}}(\mathbf{x})$ as probability of class **k**

Training NN: Softmax Loss

- If sample **x** is of class **k**, the loss is

$$\mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{W}) = -\log \mathbf{f}_{\mathbf{k}}(\mathbf{x})$$

  - this loss function is also called –log loss, cross entropy loss
  - minimizing **–log** is equivalent to maximizing probability

- Loss on all samples

$$\mathbf{L}(\mathbf{X}, \mathbf{Y}; \mathbf{W}) = \sum \mathbf{L}(\mathbf{x}, \mathbf{y}; \mathbf{W})$$

Training NN: -Log Loss Function

- Need to find derivative of **L** wrt every network weight $\mathbf{w}_i$

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

- After derivative found, according to gradient descent, weight update is

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

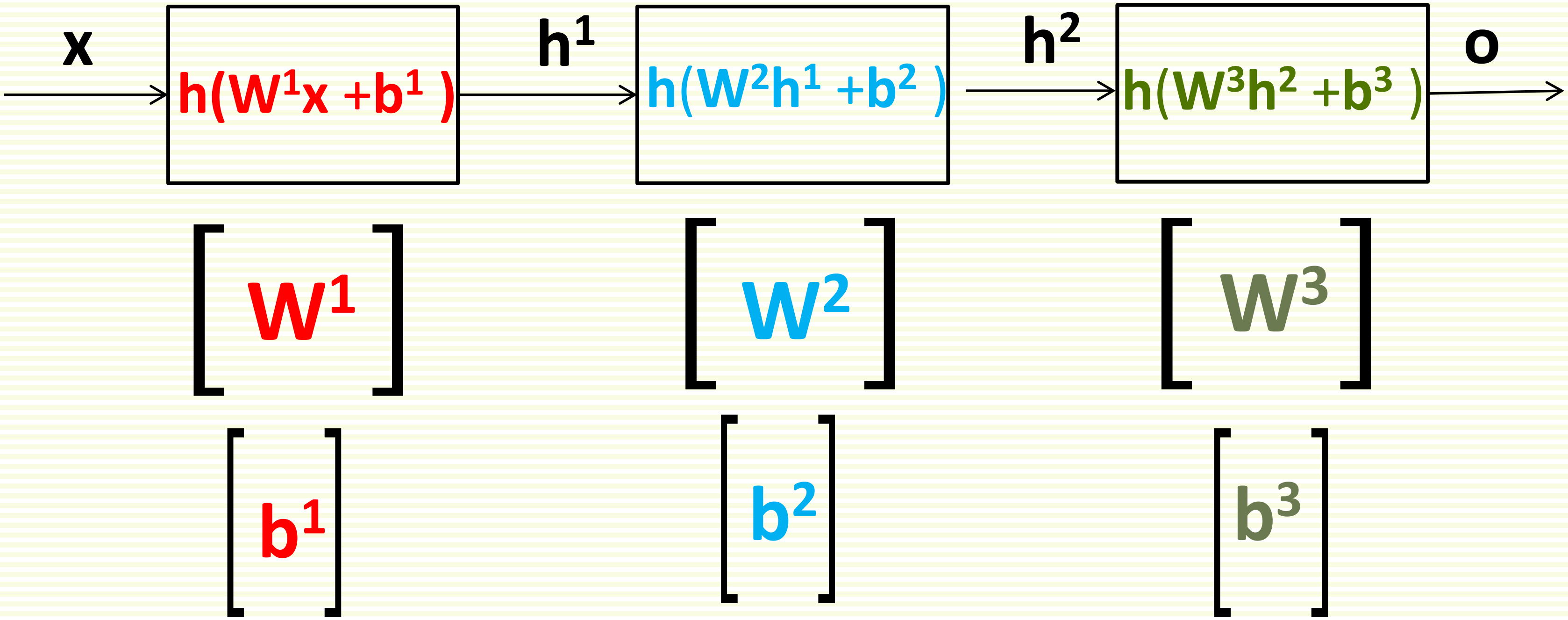
  - where α is the learning rate

- Update weight

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

Training NN: -Log Loss Function

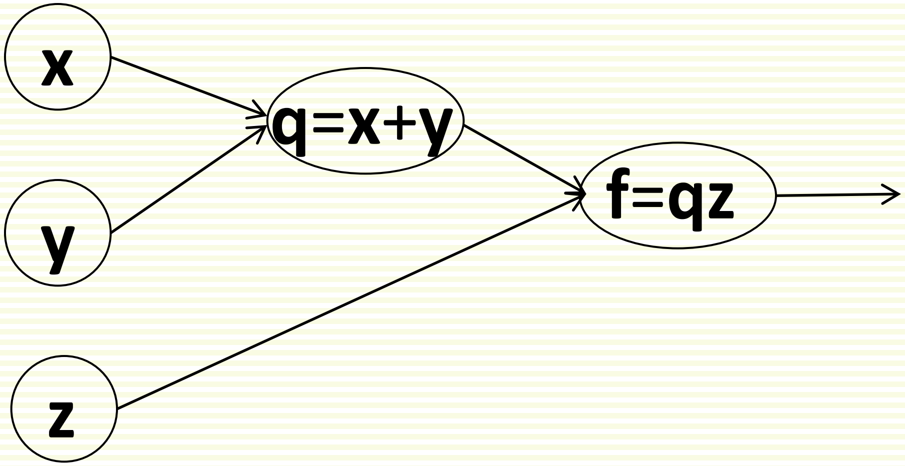
- How many weights do we have in our network?



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

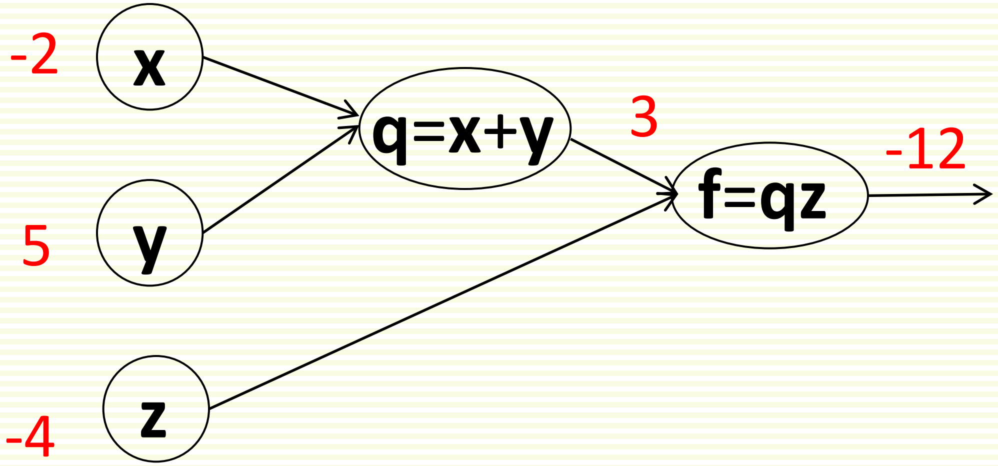
Computing Derivatives: Small Example

- Small network **f(x,y,z) = (x+y)z**
- Rewrite using
  - **q = x + y**
- **f**(**x**,**y**,**z**) = **qz**
- each node does one operation



Computing Derivatives: Small Example

- Small network **f(x,y,z)** = **(x+y)z**
- Rewrite using
  - **q** = **x** + **y**
  - **f**(**x**,**y**,**z**) = **qz**
- Example of computing **f(-2,5,-4)**

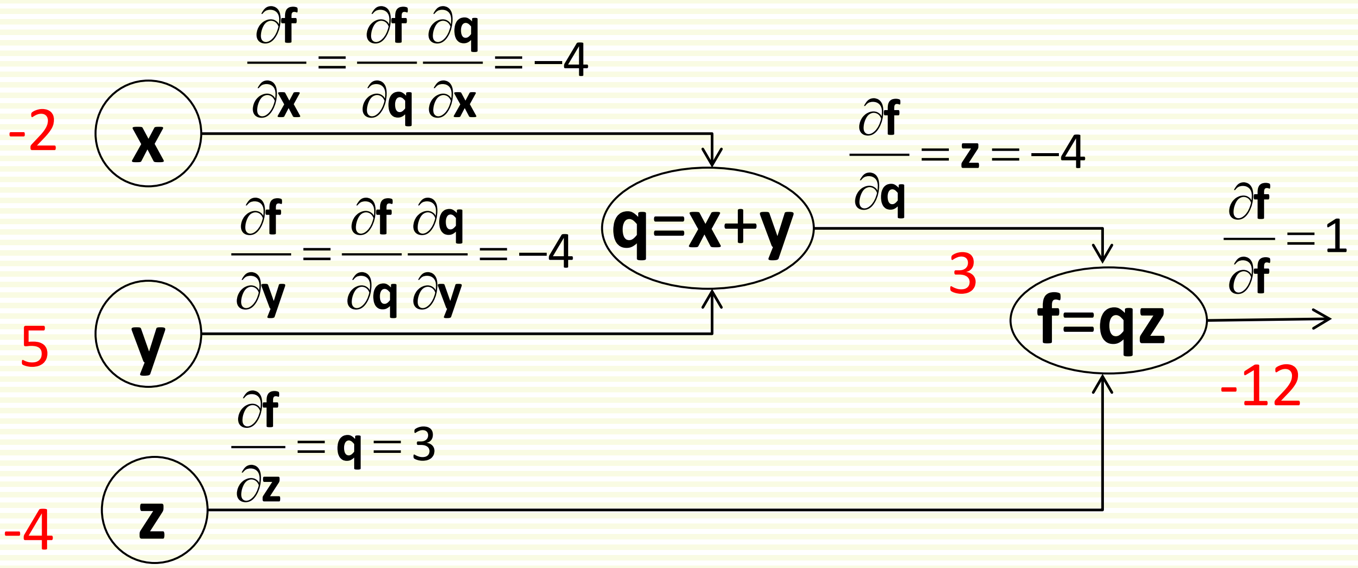


Computing Derivatives: Small Example

chain rule for **f(y(x))**

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

- Small network **f(x,y,z) = (x+y)z**
- Rewrite using **q = x + y** ⇒ **f(x,y,z) = qz**
- Want $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}, \frac{\partial \mathbf{f}}{\partial \mathbf{y}}, \frac{\partial \mathbf{f}}{\partial \mathbf{z}}$
- Compute $\frac{\partial \mathbf{f}}{\partial}$ from the end backwards
  - for each edge, with respect to the main variable at edge origin
  - using chain rule with respect to the variable at edge end, if needed

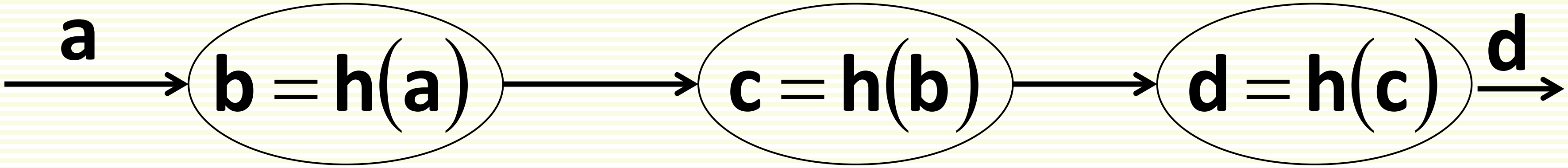


$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \mathbf{x}} = -4$
-2
x
$\frac{\partial \mathbf{f}}{\partial \mathbf{q}} = \mathbf{z} = -4$
$\frac{\partial \mathbf{f}}{\partial \mathbf{y}} = \frac{\partial \mathbf{f}}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial \mathbf{y}} = -4$
q=x+y
$\frac{\partial \mathbf{f}}{\partial \mathbf{f}} = 1$
3
5
y
f=qz
-12
$\frac{\partial \mathbf{f}}{\partial \mathbf{z}} = \mathbf{q} = 3$
-4
z

Computing Derivatives: Chain of Chain Rule

- Compute $\frac{\partial \mathbf{d}}{\partial}$ from the end backwards
  - for each edge, with respect to the main variable at edge origin
  - using chain rule with respect to the variable at edge end, if needed

direction of computation


a
$\mathbf{b} = \mathbf{h}(\mathbf{a})$
$\mathbf{c} = \mathbf{h}(\mathbf{b})$
$\mathbf{d} = \mathbf{h}(\mathbf{c})$
d

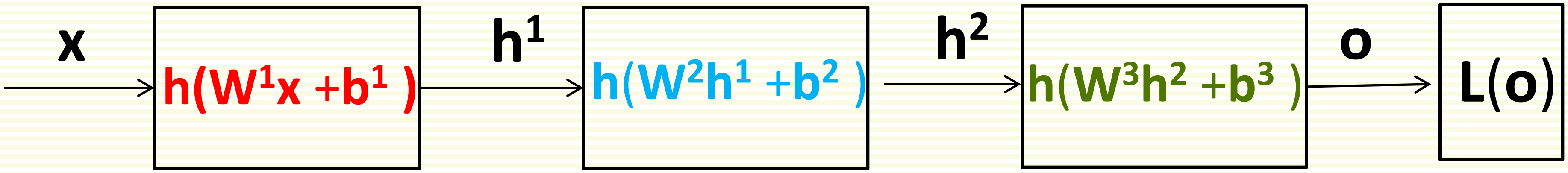
$$\frac{\partial \mathbf{d}}{\partial \mathbf{a}} = \frac{\partial \mathbf{d}}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} \qquad \frac{\partial \mathbf{d}}{\partial \mathbf{b}} = \frac{\partial \mathbf{d}}{\partial \mathbf{c}} \frac{\partial \mathbf{c}}{\partial \mathbf{b}} \qquad \frac{\partial \mathbf{d}}{\partial \mathbf{c}}$$

prev local prev local local

↓

$$\boxed{\textbf{example: if } \mathbf{h(c)} = \mathbf{c}^2, \textbf{ then } \frac{\partial \mathbf{d}}{\partial \mathbf{c}} = \frac{\partial \mathbf{h}}{\partial \mathbf{c}} = 2\mathbf{c}}$$

Computing Derivatives Backwards



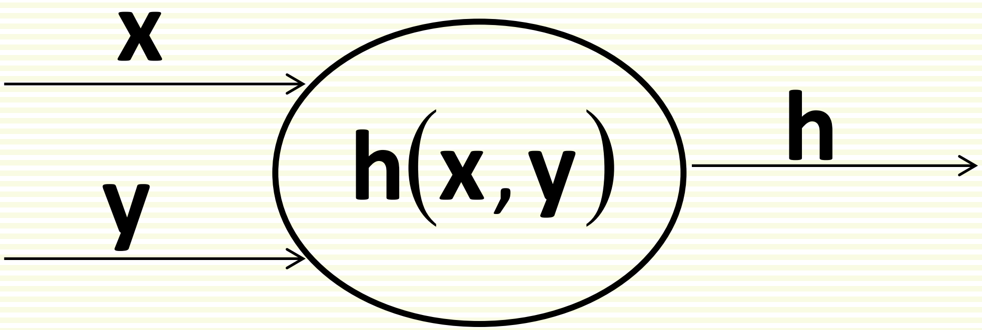
$\mathbf{x}$ → $h(W^1 x + b^1)$ → $\mathbf{h}^1$ → $h(W^2 h^1 + b^2)$ → $\mathbf{h}^2$ → $h(W^3 h^2 + b^3)$ → $\mathbf{o}$ → $\mathbf{L}(\mathbf{o})$

direction of computation

- Have loss function $\mathbf{L}(\mathbf{o})$
- Need derivatives for all $\frac{\partial \mathbf{L}}{\partial \mathbf{w}}, \frac{\partial \mathbf{L}}{\partial \mathbf{b}}$
- Will compute derivatives from end to front, backwards
- On the way will also compute intermediate derivatives $\frac{\partial \mathbf{L}}{\partial \mathbf{h}}$

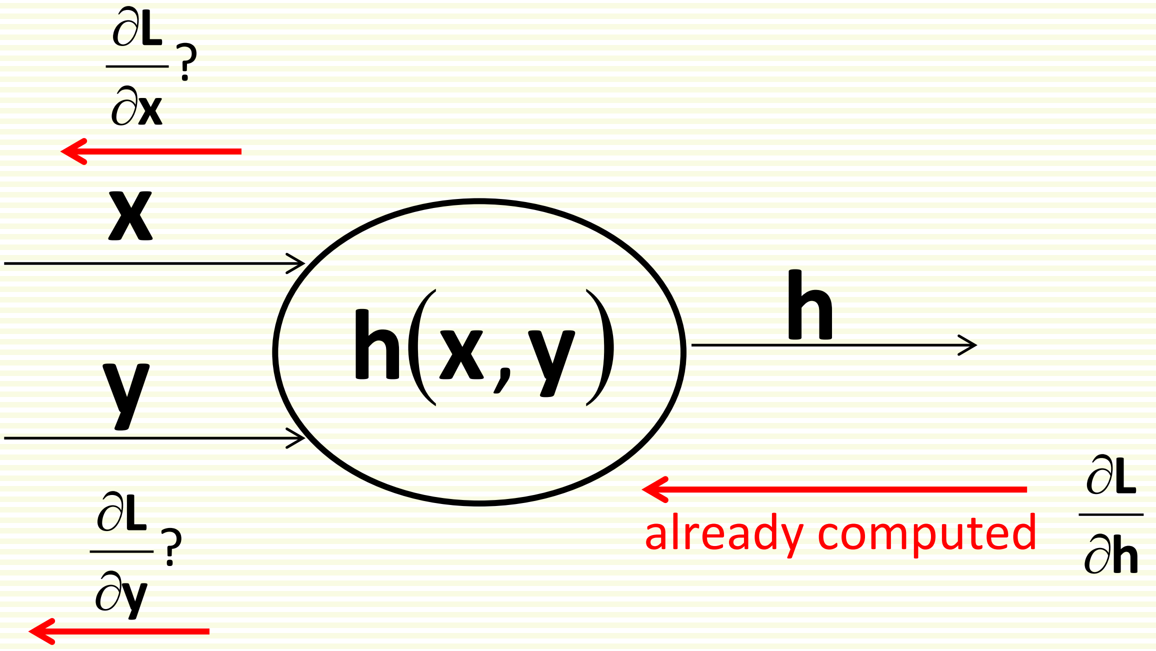
Computing Derivatives: Look at One Node

- Simplified view at a network node
  - inputs **x**,**y** come in
  - node computes some function **h**(**x**,**y**)



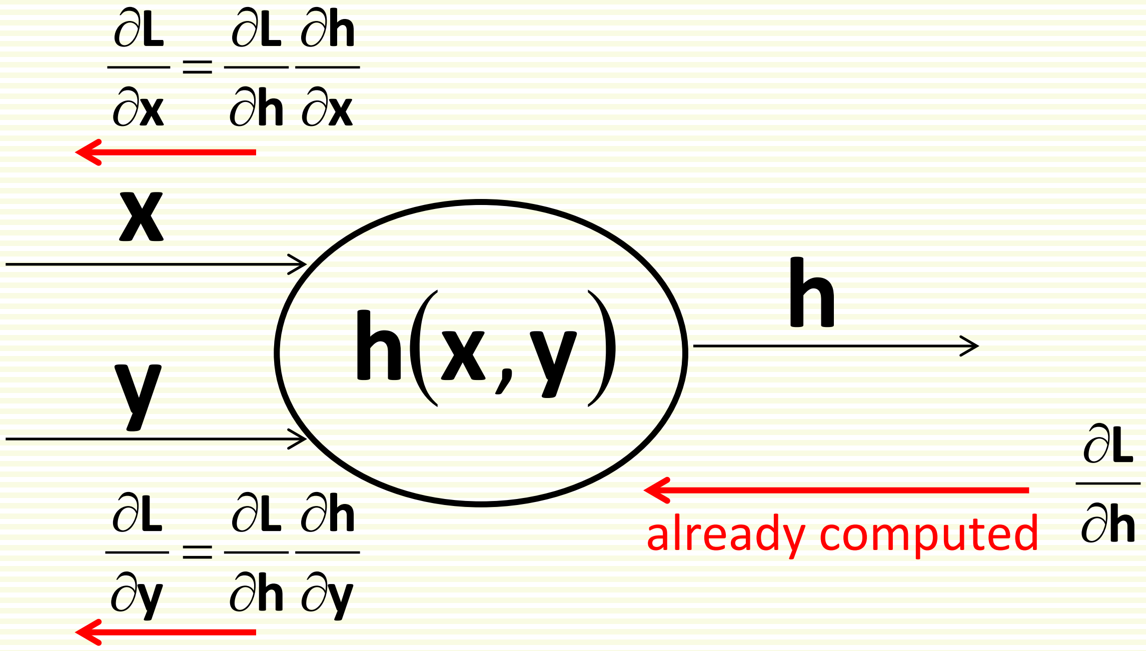
Computing Derivatives: Look at One Node

- At each network node
  - inputs **x**,**y** come in
  - nodes computes activation function **h**(**x**,**y**)
- Have loss function **L**(·)


$\frac{\partial \mathbf{L}}{\partial \mathbf{x}}$?
x
y
h(x,y)
h
$\frac{\partial \mathbf{L}}{\partial \mathbf{y}}$?
already computed
$\frac{\partial \mathbf{L}}{\partial \mathbf{h}}$

Computing Derivatives: Look at One Node

- Need $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}$
- Easy to compute local node derivatives $\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$$

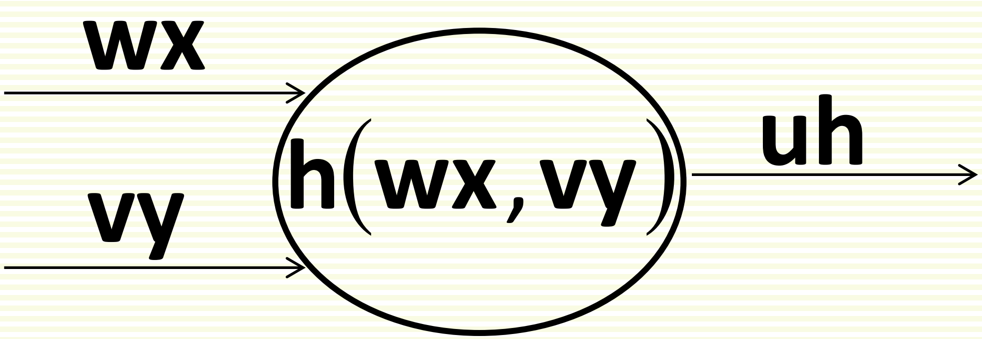
$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial y}$$

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

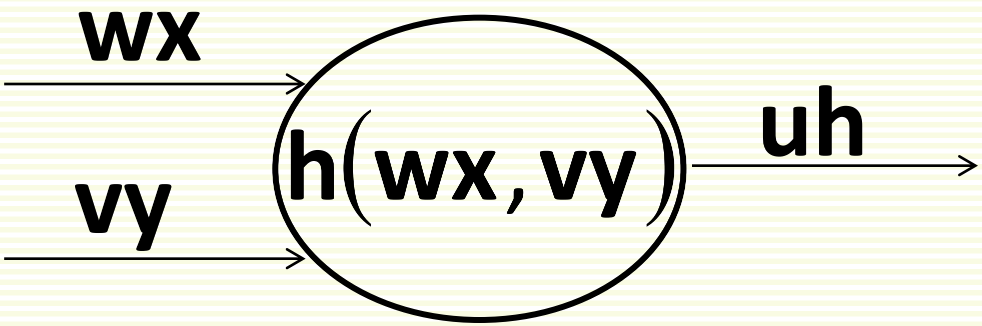
already computed

Computing Derivatives: Look at One Node

- More complete view at a network node
  - inputs **x**,**y** come in, get multiplied by weight **w** and **v**
  - node computes function **h**(**wx**,**vy**)
  - node output **h** gets multiplied by **u**

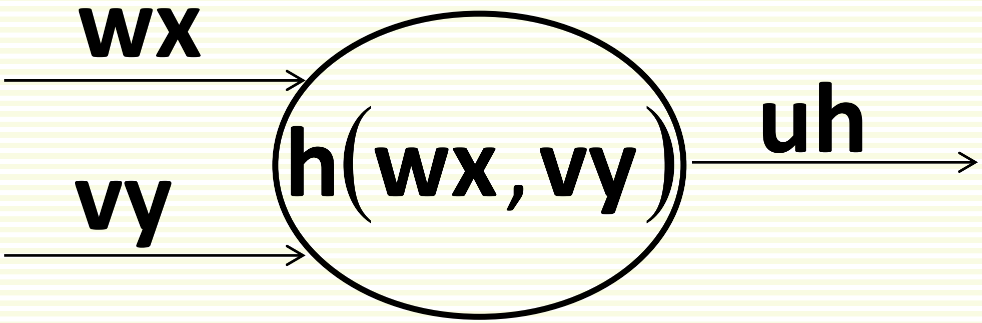


Computing Derivatives: Look at One Node



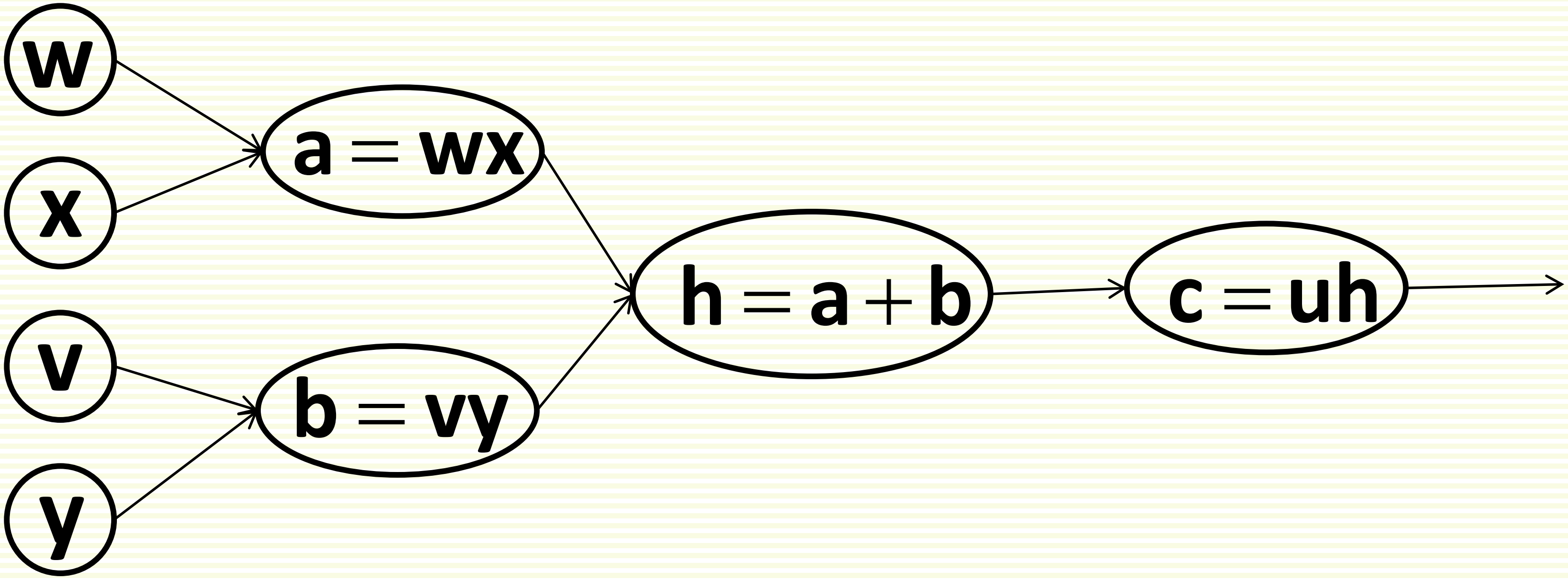
- To be concrete, let **h**(**i**,**j**) = **i** + **j**

Computing Derivatives: Look at One Node

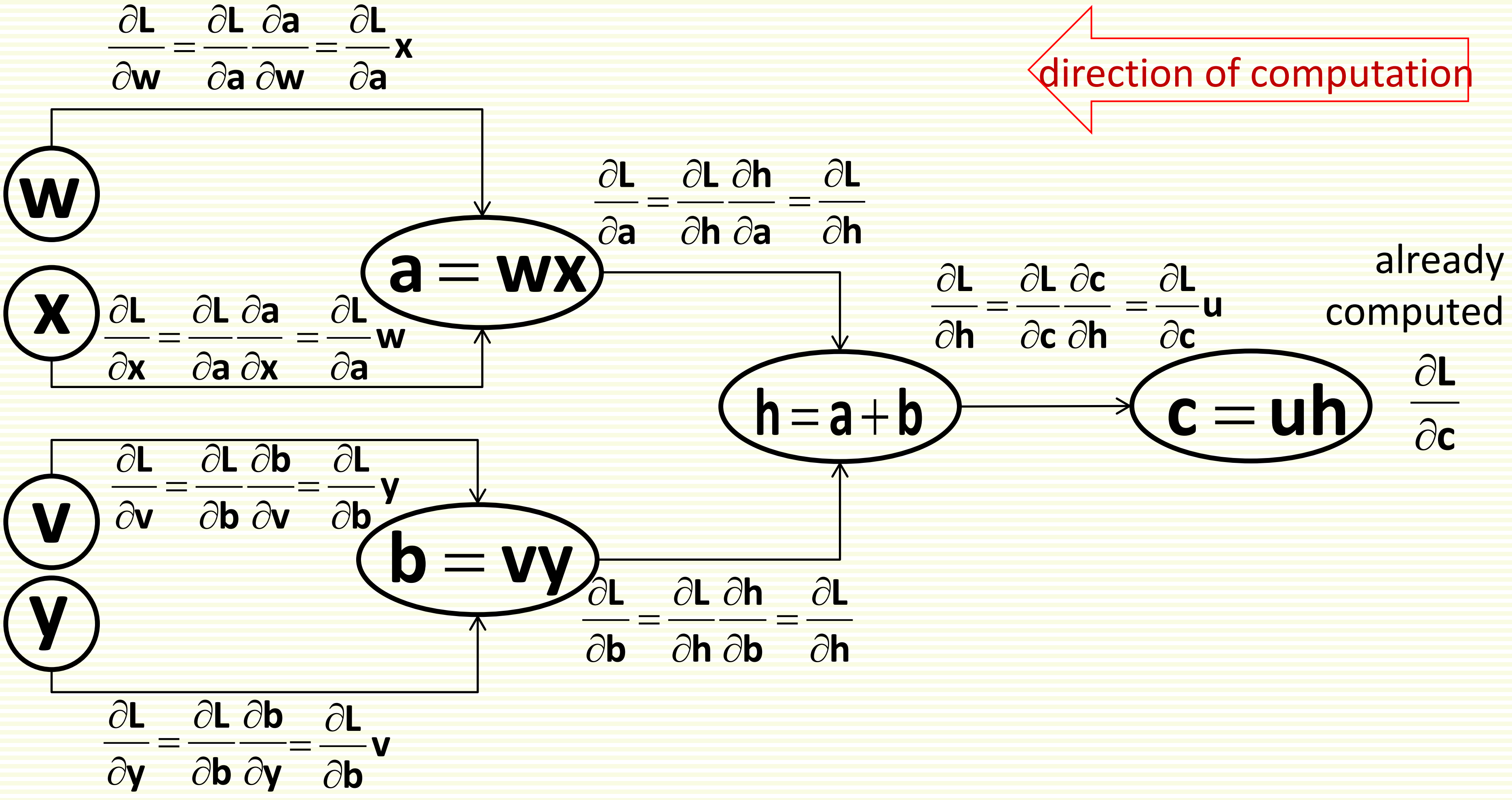


- $h(i,j) = i + j$

- Break into more computational nodes
  - all computation happens inside nodes, not on edges



Computing Derivatives: Look at One Node



$$\frac{\partial L}{\partial w}=\frac{\partial L}{\partial a}\frac{\partial a}{\partial w}=\frac{\partial L}{\partial a}x$$

direction of computation

$$\frac{\partial L}{\partial a}=\frac{\partial L}{\partial h}\frac{\partial h}{\partial a}=\frac{\partial L}{\partial h}$$

w

$a = wx$

x

$$\frac{\partial L}{\partial x}=\frac{\partial L}{\partial a}\frac{\partial a}{\partial x}=\frac{\partial L}{\partial a}w$$

$$\frac{\partial L}{\partial h}=\frac{\partial L}{\partial c}\frac{\partial c}{\partial h}=\frac{\partial L}{\partial c}u$$

already computed

$$\frac{\partial L}{\partial c}$$

$h = a + b$

$c = uh$

$$\frac{\partial L}{\partial v}=\frac{\partial L}{\partial b}\frac{\partial b}{\partial v}=\frac{\partial L}{\partial b}y$$

v

$b = vy$

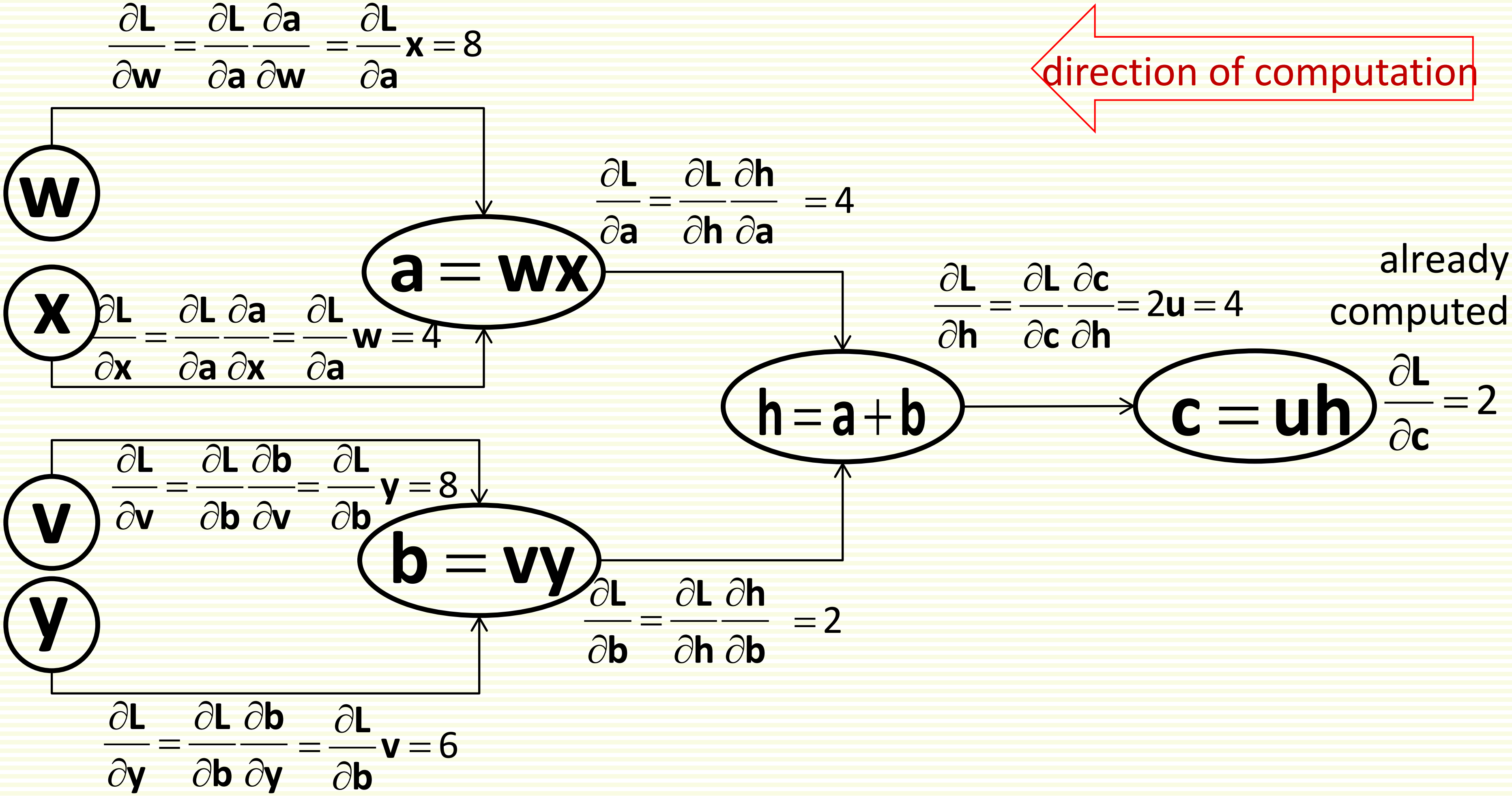
y

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

$$\frac{\partial L}{\partial y}=\frac{\partial L}{\partial b}\frac{\partial b}{\partial y}=\frac{\partial L}{\partial b}v$$

- Some of these partial derivatives are intermediate
  - their values will not be used for gradient descent

Computing Derivatives: Look at One Node



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w} = \frac{\partial L}{\partial a}\mathbf{x} = 8$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial x} = \frac{\partial L}{\partial a}\mathbf{w} = 4$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial a} = 4$$

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial h} = 2\mathbf{u} = 4$$

$$\frac{\partial L}{\partial c} = 2$$

$$\frac{\partial L}{\partial v} = \frac{\partial L}{\partial b}\frac{\partial b}{\partial v} = \frac{\partial L}{\partial b}\mathbf{y} = 8$$

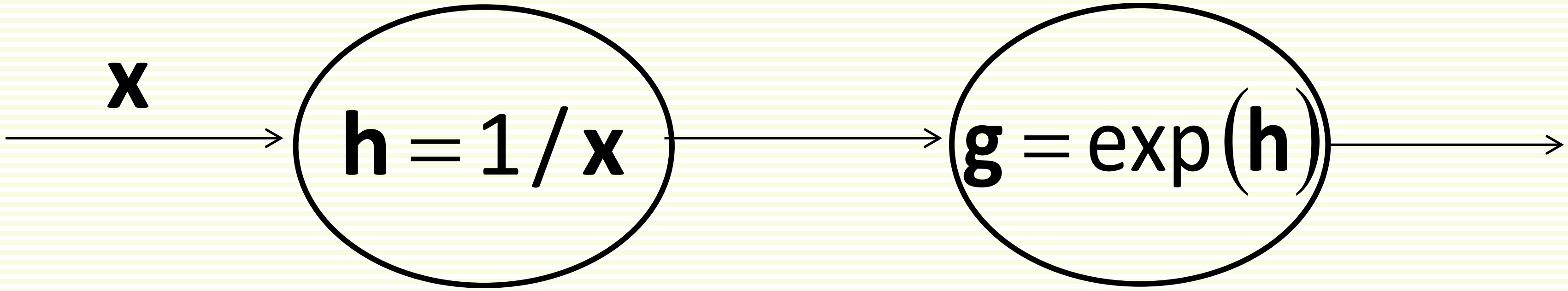
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial b} = 2$$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial b}\frac{\partial b}{\partial y} = \frac{\partial L}{\partial b}\mathbf{v} = 6$$

- Example when **w** = 1, **x** = 2, **v** = 3, **y** = 4, u = 2, $\frac{\partial L}{\partial c} = 2$

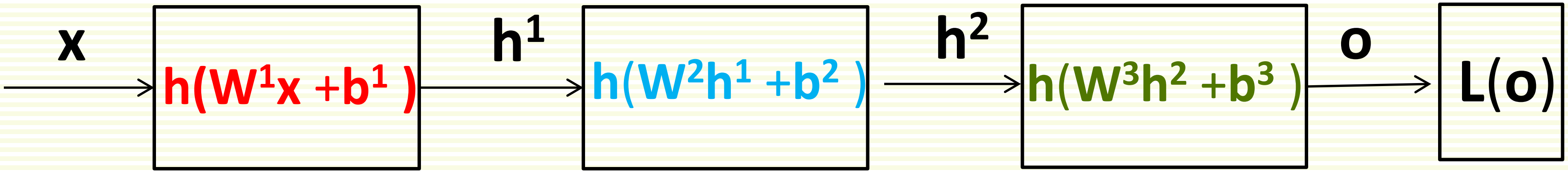
Computing Derivatives: Staging Computation

- Each node is responsible for one function
- To compute exp(1/**x**)



Computing Derivatives: Vector Notation

- Inputs and outputs are often vectors and/or matrices



$\mathbf{x}$

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

$\mathbf{h}^1$

$\mathbf{h}(\mathbf{W}^2\mathbf{h}^1 + \mathbf{b}^2)$

$\mathbf{h}^2$

$\mathbf{h}(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3)$

$\mathbf{o}$

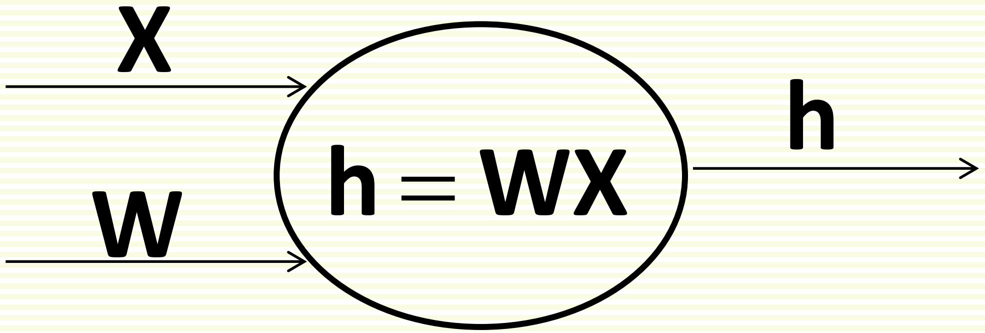
$L(\mathbf{o})$

- **h**(a) is a function from $\mathbf{R}^n$ to $\mathbf{R}^m$
- Chain rule generalizes to vector and matrix functions
- Will not derive it, but will give you the end result

Computing Derivatives: Vector Notation

- Assume loss **L** is a scalar
  - if not, can do derivation for each component independently
- Assume **W**, **X**, and **h** are matrices
  - subsumes the case when they are vectors as well

$$\frac{\partial \mathbf{L}}{\partial \mathbf{X}} = \mathbf{W}^{T} \frac{\partial \mathbf{L}}{\partial \mathbf{h}}$$

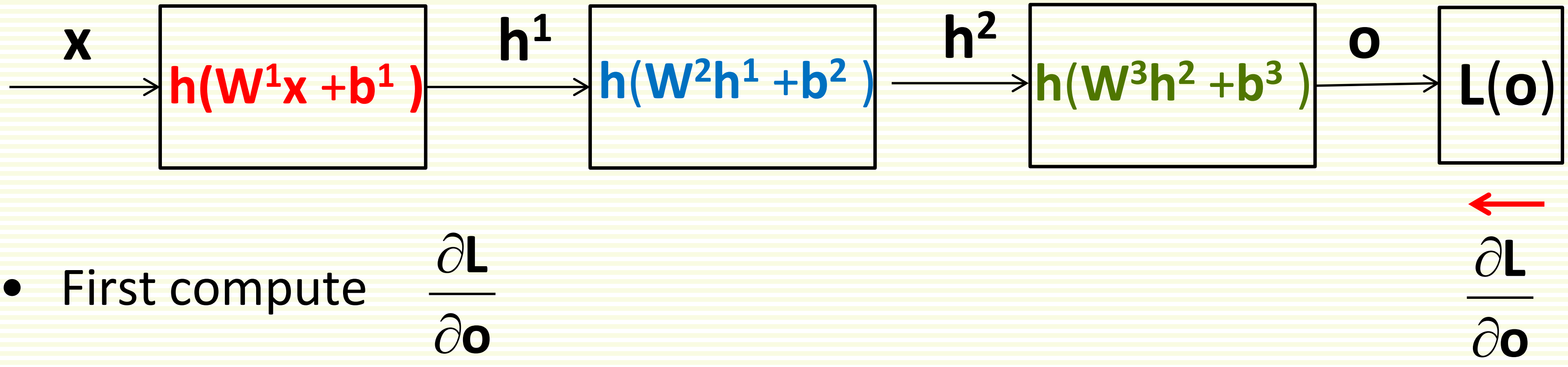


already computed $\frac{\partial \mathbf{L}}{\partial \mathbf{h}}$

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}} \mathbf{X}^{T}$$

- $\frac{\partial \mathbf{L}}{\partial \mathbf{W}}$ is a matrix of partial derivatives of the same shape as **W**

Computing Derivatives: Vector Notation



- First compute $\frac{\partial \mathbf{L}}{\partial \mathbf{o}}$

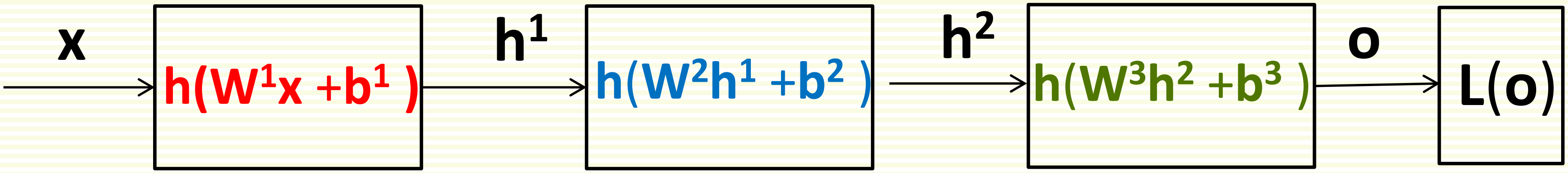
- Under quadratic loss

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

- Under softmax loss

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

Computing Derivatives: Vector Notation



$$\xleftarrow{\quad\quad\quad} \quad \frac{\partial L}{\partial \mathbf{o}}$$

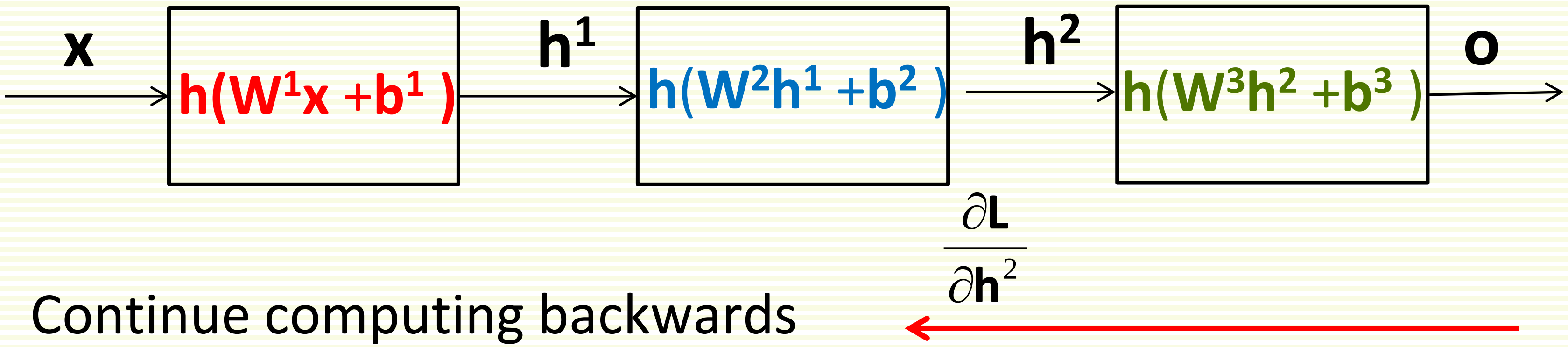
- Let vector $\mathbf{a}^3 = \mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3$ $\quad\quad$ $\mathbf{a}^3 = \begin{bmatrix} a^3{}_1 \\ a^3{}_2 \end{bmatrix}$

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

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

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

Computing Derivatives: Vector Notation



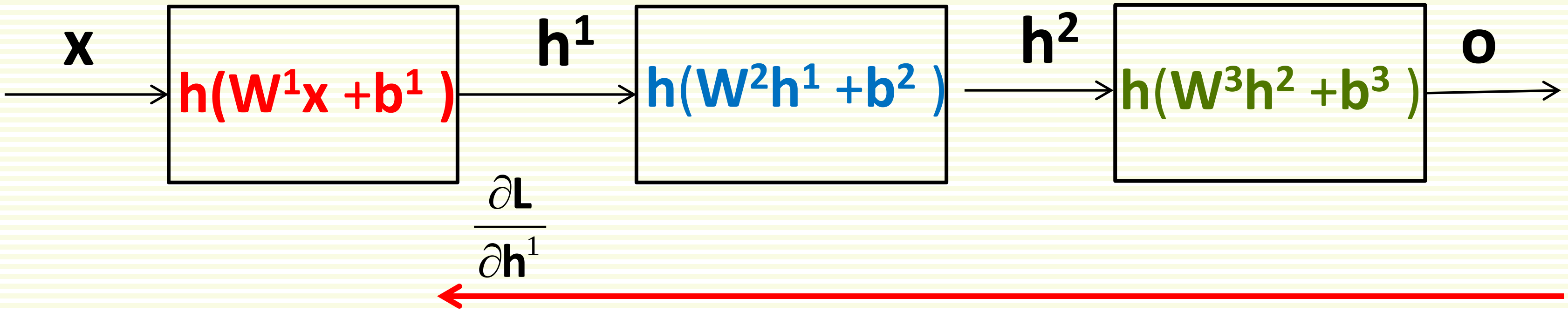
- Continue computing backwards
- Let vector **a² = W²h¹ +b²**

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

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

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

Computing Derivatives: Vector Notation


$\mathbf{x}$
$h(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1)$
$\mathbf{h}^1$
$h(\mathbf{W}^2\mathbf{h}^1 + \mathbf{b}^2)$
$\mathbf{h}^2$
$h(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3)$
$\mathbf{o}$
$\frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}$

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

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

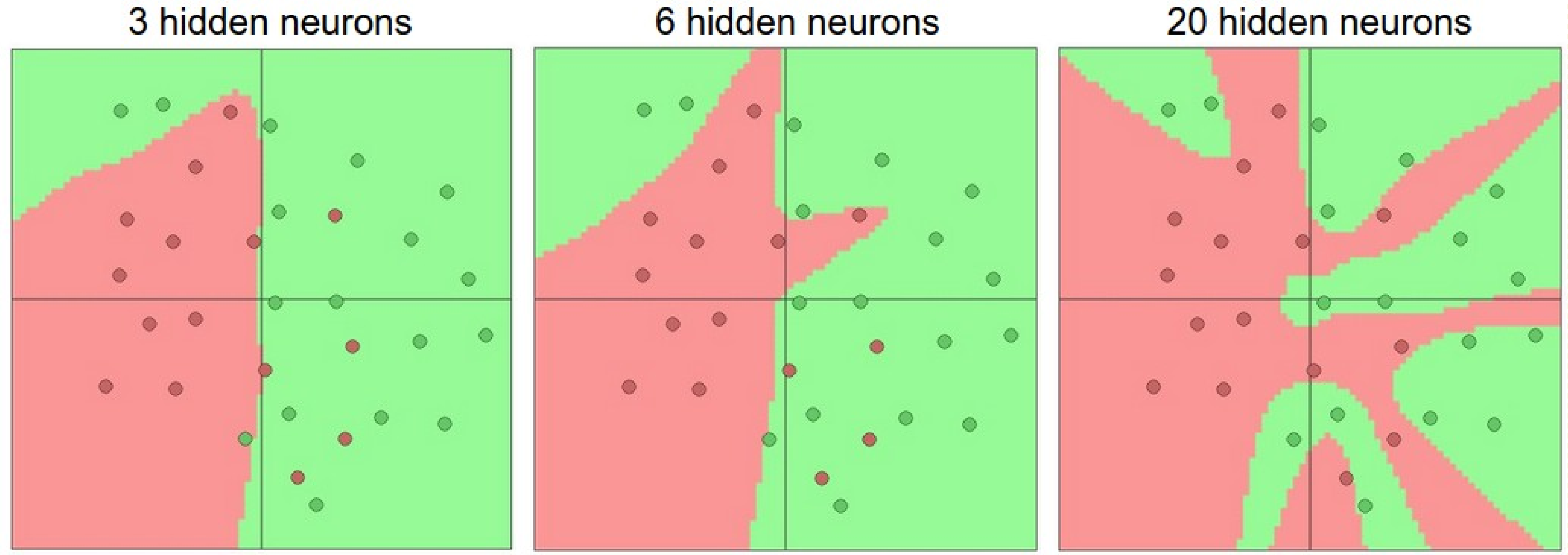
$$\frac{\partial \mathbf{L}}{\partial \mathbf{b}^1} = \mathbf{diag}\left(\mathbf{h}'\left(\mathbf{a}^1\right)\right)\frac{\partial \mathbf{L}}{\partial \mathbf{h}^1}$$

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**

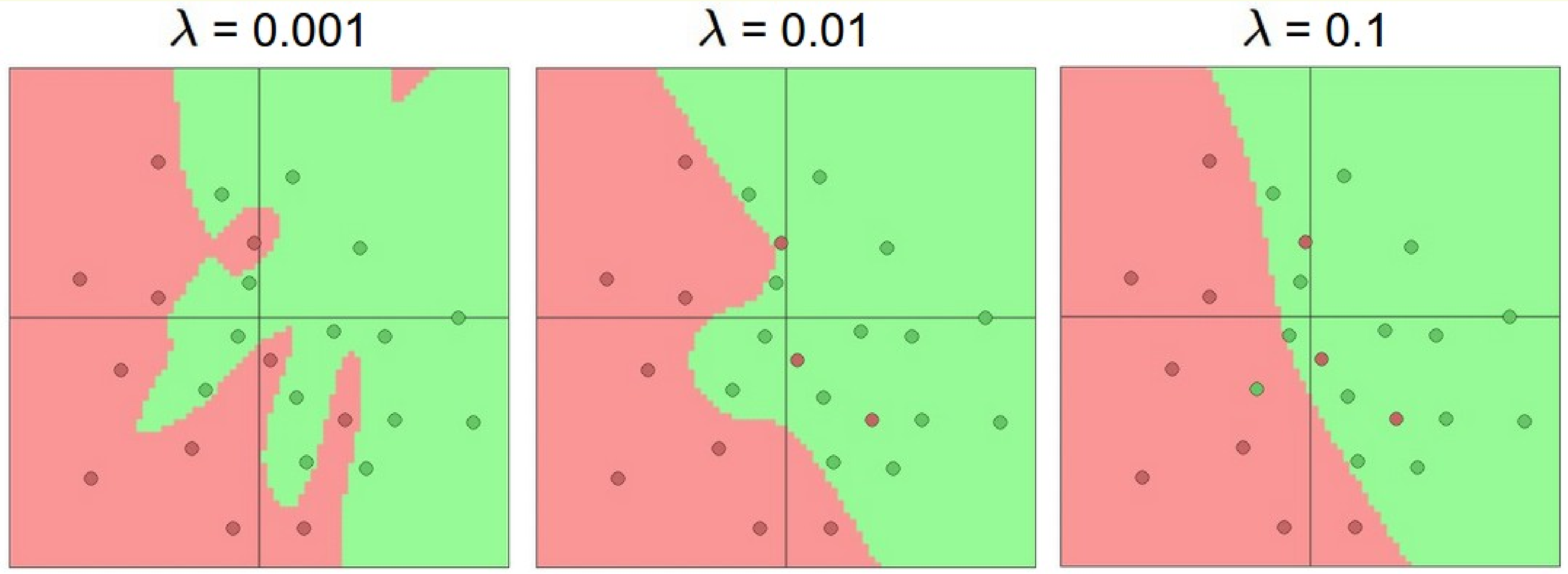
Regularization

- Larger networks are more prone to overfitting



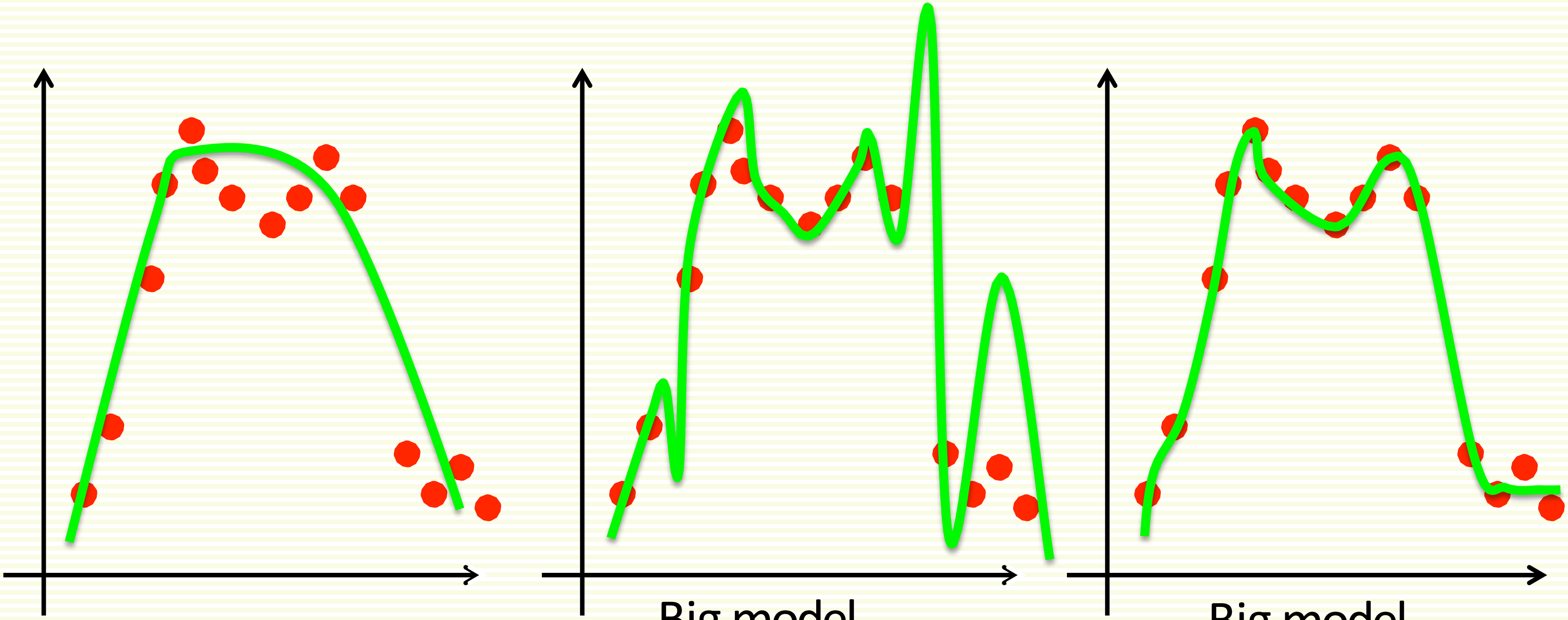
Regularization

- Can control overfitting by using network with less units
- Better if control overfitting by adding weight regularization $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss function



- During gradient descent, subtract λ**w** from each weight **w**
  - intuitively, implements weight decay

Small model vs. Big Model+Regularize



Small model

Big model

Big model

+Regularize

Ensembles of Neural Networks

- Train multiple independent models, average their predictions
- Improvements are more dramatic with higher model variety
- Few approaches to forming an ensemble
  - **Same model, different initializations**
    - train multiple models with the best set of hyperparameters (found through cross validation) but with different random initialization.
    - drawback is that the variety is only due to initialization
  - **Top models discovered during cross-validation**
    - Use cross-validation to determine the best hyperparameters, then pick the top few
    - Improves ensemble variety but has the danger of including suboptimal models
    - practical, does not require additional retraining of models after cross-validation
  - **Different checkpoints of a single model**
    - Take different “checkpoints” of a single network over time
    - Lacks variety, but very cheap
  - **Running average of parameters during training**
    - Maintain a second copy of the network’s weights in memory that maintains an exponentially decaying sum of previous weights during training
    - This way you’re averaging the state of the network over last several iterations

Practical Training Tips: Initialization

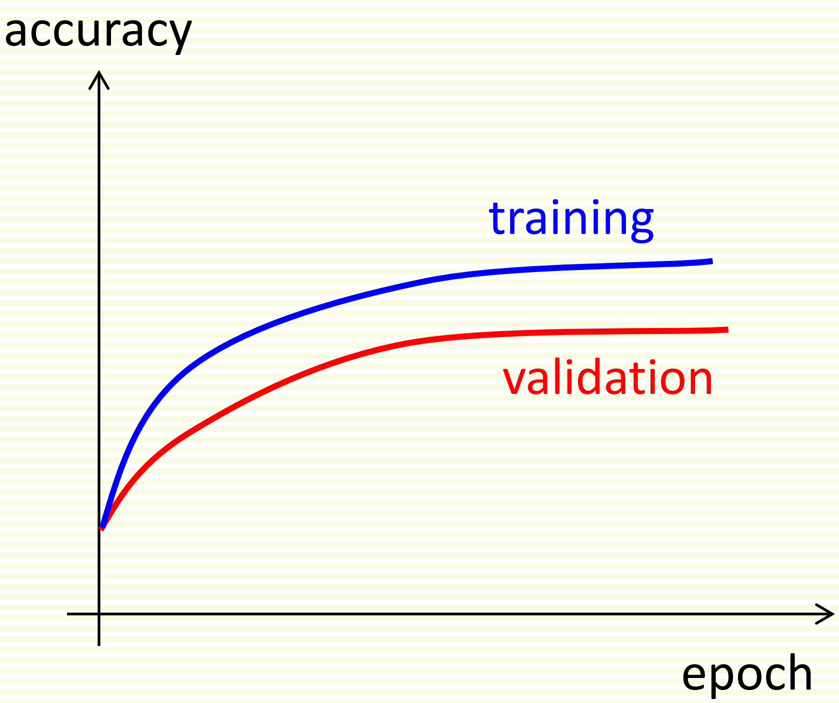
- Initialization parameters for **W**
  - do not set all the parameters **W** equal
    - all units compute the same output, gradient descent updates are the same
  - can initialize **W** to small random numbers
  - if using RELU, better initialize with $\mathbf{randn(n)}\frac{2}{\sqrt{\mathbf{n}}}$, where **n** is number of inputs to the unit
- Biases **b** usually initialized to 0
  - with ReLU often intialize to small positive number, like 0.1

Practical Training Tips: Learning Rate

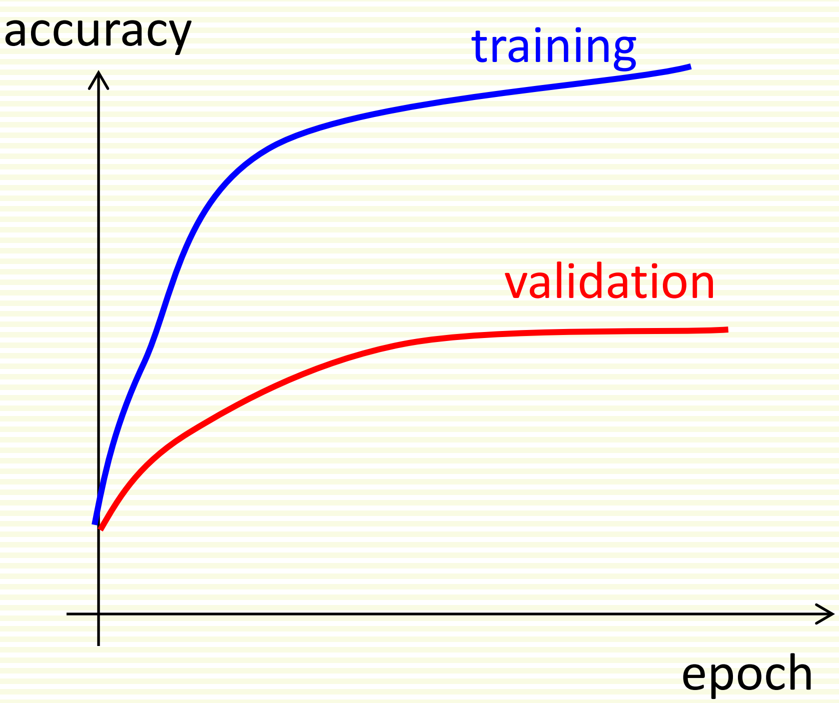
- Loss **L**(**w**) should decrease during gradient descent
  - If **L**(**w**) oscillates, α is too large, decrease it
  - If **L**(**w**) goes down but very slowly, α is too small, increase it
- Typically cross-validate learning rates from 10^{-2} to 10^{-5}
- Helps to adjust α at the training time, especially for many layered (deep) networks
  - **Step decay**
    - reduce learning rate by some factor every few epochs
    - i.e. by a factor 0.5 every 5 epochs, or by 0.1 every 20 epochs
  - **Exponential decay**
    - $\alpha=\alpha_0 e^{-kt}\alpha$, where α_0,k are hyperparameters and t is epoch number
  - **1/t decay**
    - $\alpha=\alpha_0/(1+kt)$ where a_0,k are hyperparameters and t is epoch number
  - Err on the side of slower decay, if time budget allows

Practical Training Tips: Validation/Training Accuracy

- Track number of epoch vs. validation/training accuracy



- Not much overfitting, increase network capacity?



- Strong overfitting, increase regularization?

Practical Training Tips: Momentum

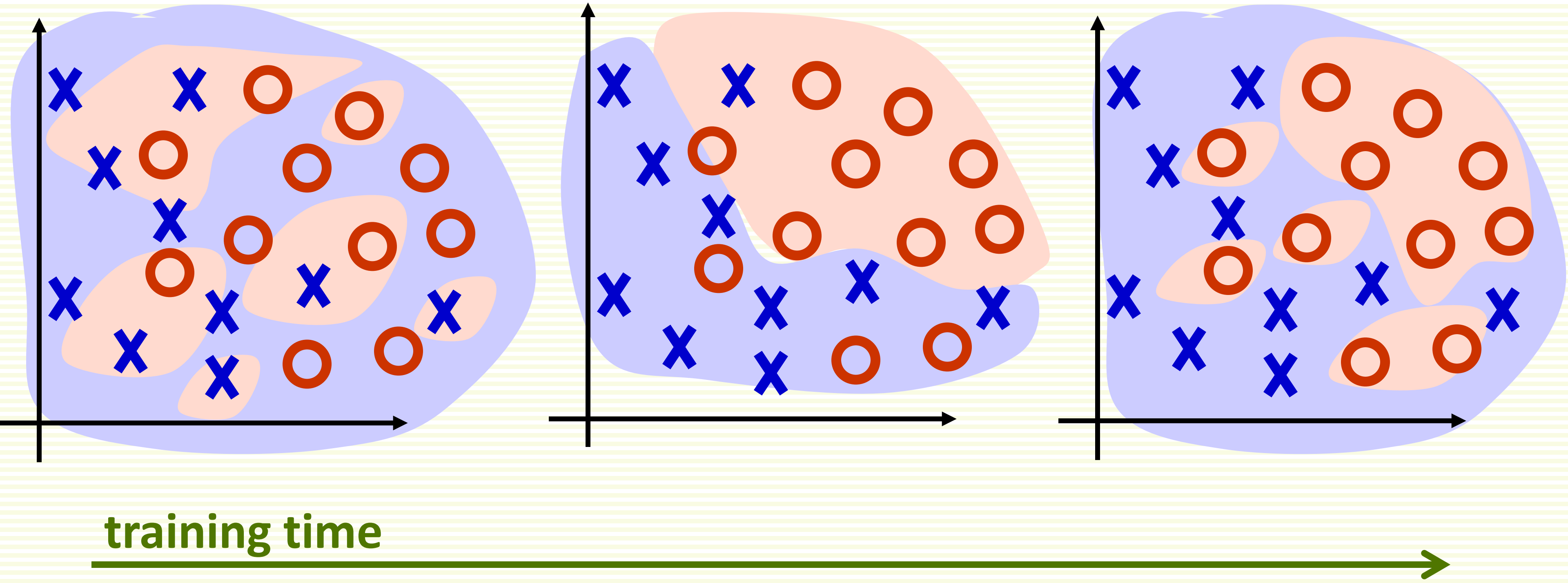
- Add temporal average direction in which weights have been moving recently
- Parameter vector will build up velocity in direction that has consistent gradient
- Helps avoid local minima and speed up descent in flat (plateau) regions
- Previous direction: $\Delta \mathbf{w}^t = \mathbf{w}^t - \mathbf{w}^{t-1}$
- Weight update rule with momentum
    - common to set **β** ∈ (0.6,0.9), also can cross-validate

$$\mathbf{w}^{\mathbf{t}+1} = \mathbf{w}^{\mathbf{t}} + \underbrace{(1-\beta)\nabla \mathbf{L}(\mathbf{w}^{\mathbf{t}})}_{\text{steepest descent direction}} \quad + \quad \underbrace{\beta\,\Delta\mathbf{w}^{\mathbf{t}-1}}_{\text{previous direction}}$$

Practical Training Tips: 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
- Images are already roughly normalized
  - intensity/color are in the range [0,255]
  - usually subtract mean image from training data, zero-centers data
    - mean computed on training data only
    - subtracted from test data as well

Training NN: How Many Epochs?



training time

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

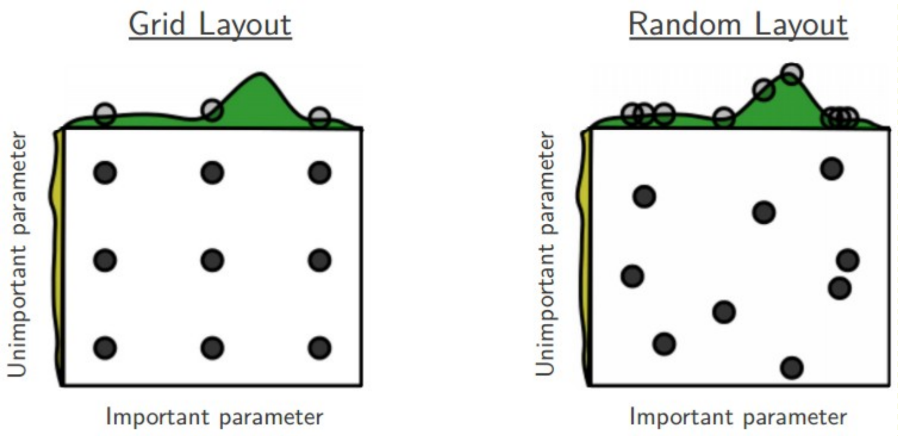
Small training error: decision regions improve with time

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

- Learn when to stop training through validation

Other Practical Training Tips

- Before training on full dataset, make sure can overfit on a small portion of the data
  - turn regularization off
- Search hyperparameters on coarse scale for a few epoch, and then on finer scale for more epochs
  - random search might be better than grid search


Grid Layout
Unimportant parameter
Important parameter
Random Layout
Unimportant parameter
Important parameter