CS9840

Learning and Computer Vision

Prof. Olga Veksler

Lecture 10

Neural Networks

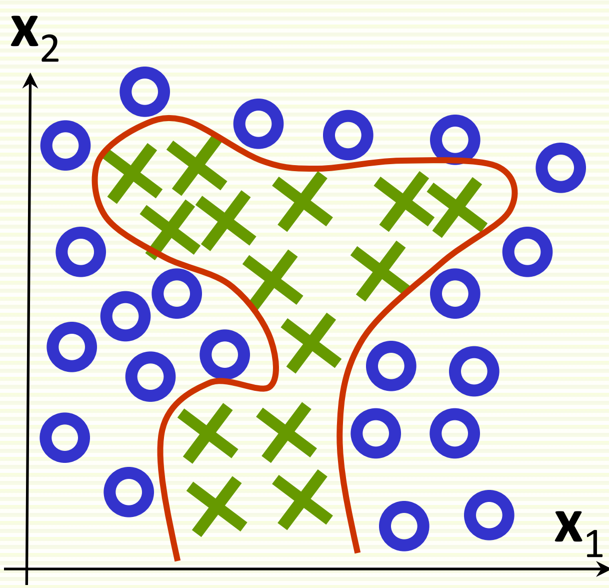
Many slides are from A. Ng, Y. LeCun, G. Hinton, A. Ranzato, Fei-Fei Li, R. Fergus

Outline

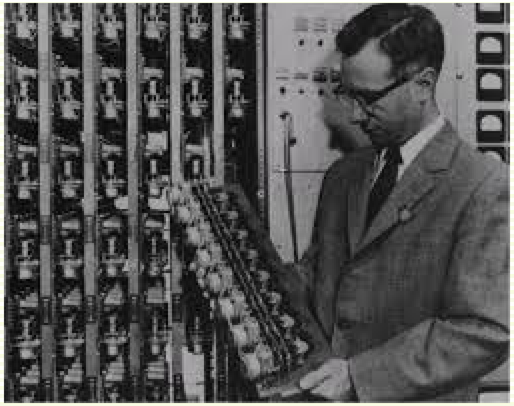
- Intro/History
- Perceptron: 1 layer Neural Network (NN)
- Multilayer NN
  - also called
    - Multilayer Perceptron (MLP)
    - Artificial Neural Network (ANN)
    - Feedforward Neural Network
- Training Neural Networks
  - Backpropagation algorithm
  - Practical tips for training

Artificial Neural Networks

- Neural Networks correspond to some classifier function $\mathbf{f}_{\mathbf{NN}}(\mathbf{x})$
- Can carve out arbitrarily complex decision boundaries without requiring as many terms as polynomial functions
- Originally inspired by research in how human brain works
  - but cannot claim that this is how the brain actually works
- Now very successful in practice, but took a while to get there



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

- 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)

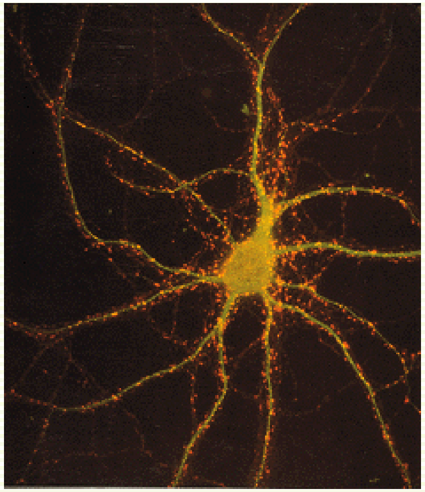
- Revival of ANN in early 1980
- 1986, (re)discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows training a MLP
- Many examples of mulitlayer Neural Networks appear
- 1998, Convolutional network (convnet) by Y. Lecun for digit recognition, very successful
- 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

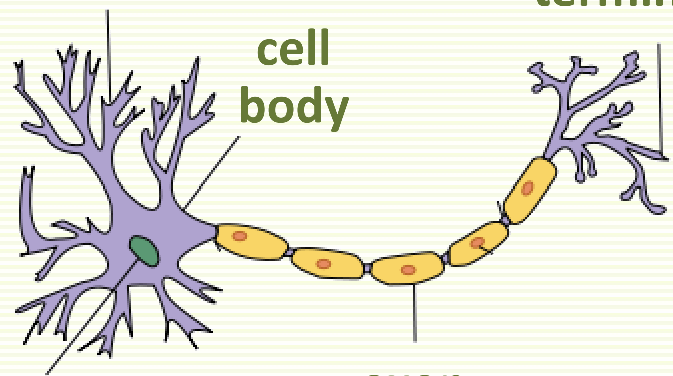
- Deep networks are inspired by brain architecture
- Until now, no success at training them, except convnet
- 2006-now: deep networks are trained successfully
  - massive training data becomes available
  - better hardware: fast training on GPU
  - better training algorithms for network training when 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.

Biology: 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

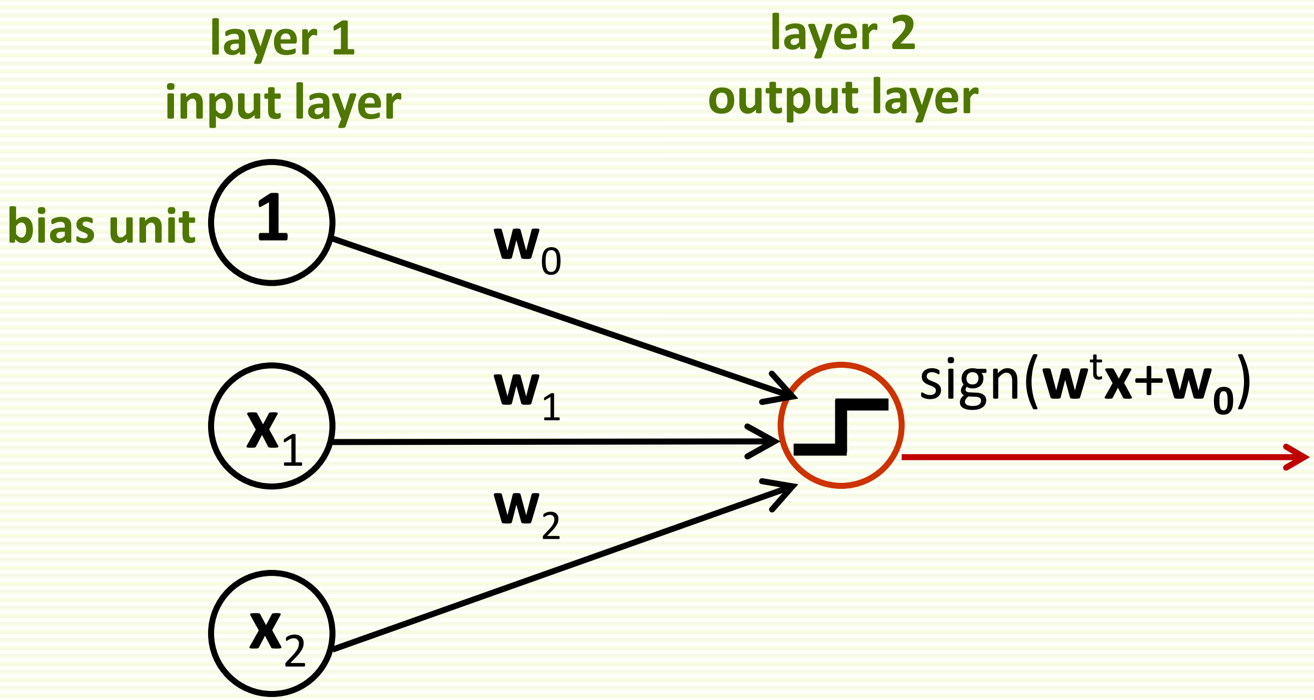


Biology: Main Components of Neuron



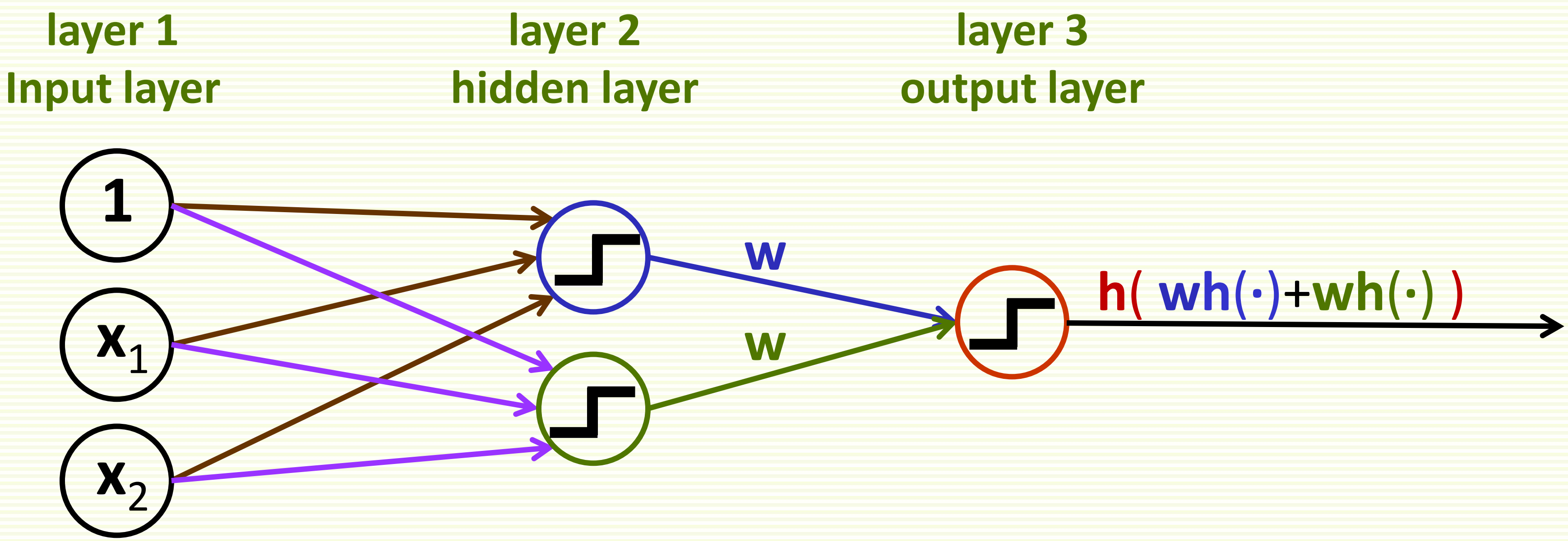
- **cell body**
  - computational unit
- **dendrites**
  - “input wires”, receive inputs from other neurons
  - a neuron may have thousands of dendrites, usually short
- **axon**
  - “output wire”, sends signal to other neurons
  - single long structure (up to 1 meter)
  - splits in possibly thousands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

Perceptron: 1 Layer Neural Network



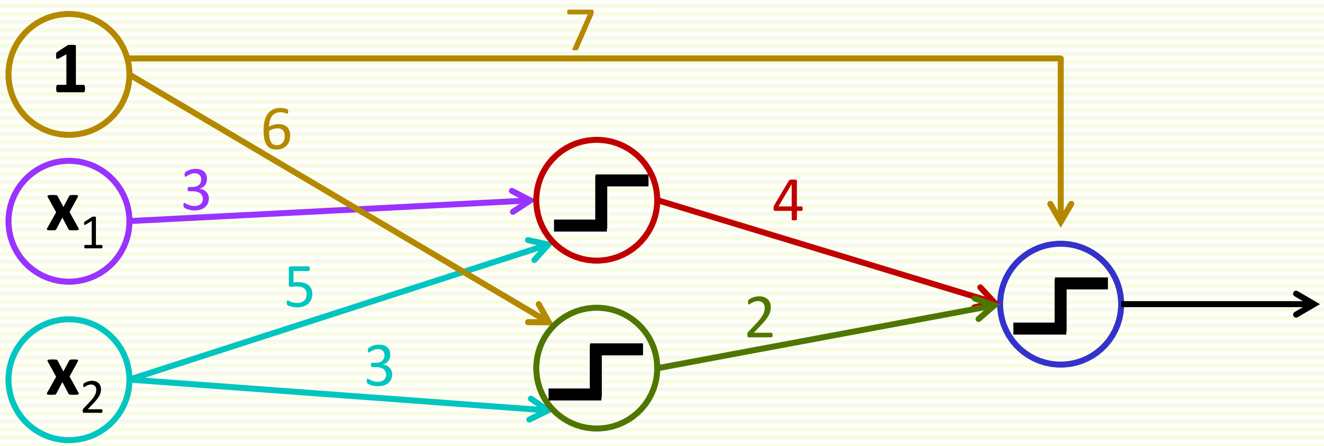
- Linear classifier $\mathbf{f}(\mathbf{x}) = \text{sign}(\mathbf{w}^t\mathbf{x}+\mathbf{w}_0)$ is a single neuron "net"
- Input layer units emits features, except bias emits "1"
- Output layer unit applies $\mathbf{h}(t) = \textbf{sign}(t)$
- $\mathbf{h}(t)$ is also called an *activation function*

Multilayer Neural Network



- 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

Multilayer Neural Network: Small Example



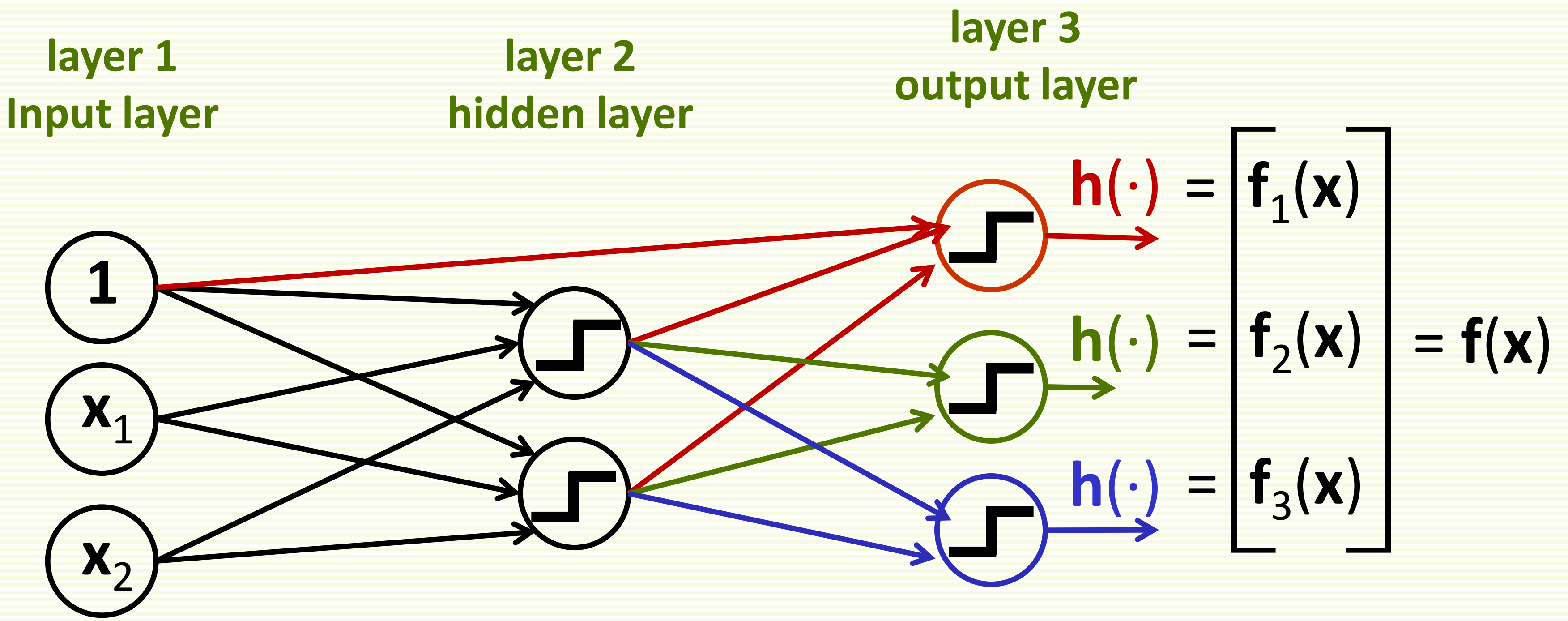
- Implements classifier

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

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

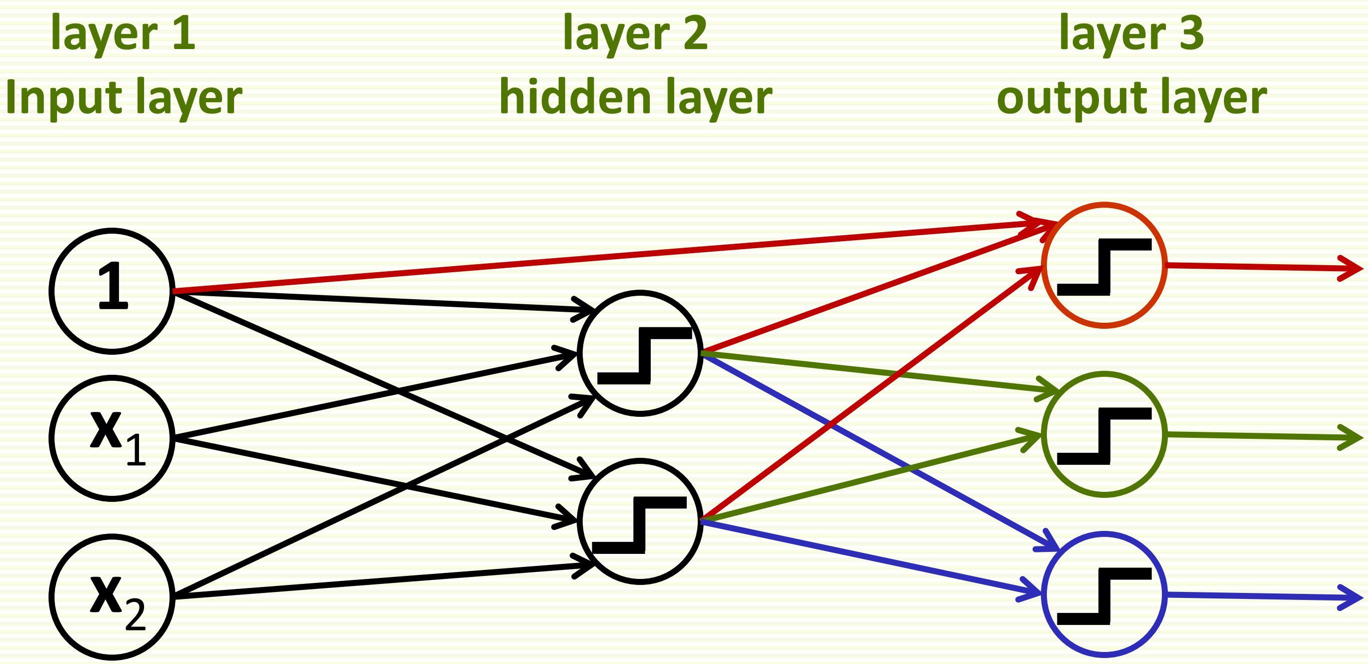
- Computing **f**(**x**) is called *feed forward operation*
  - graphically, function is computed from left to right
- Edge weights are learned through training

Multilayer NN: General Structure



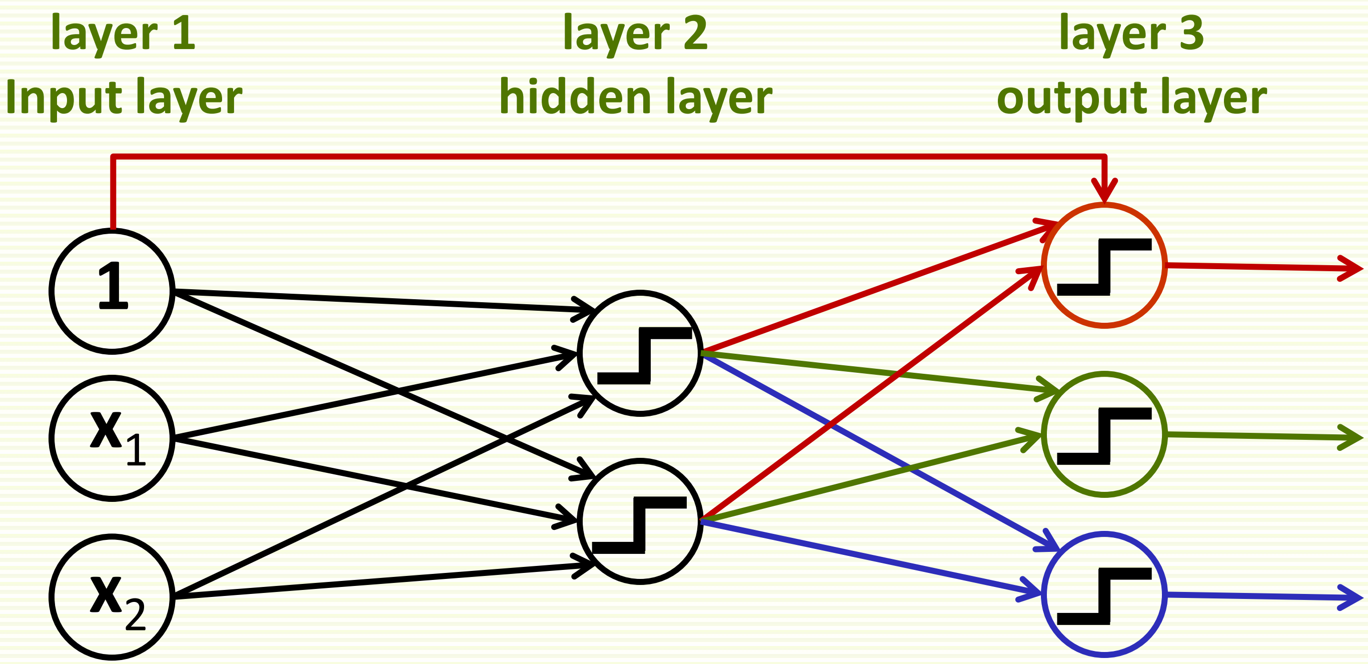
- $\mathbf{f}(\mathbf{x})$ is multi-dimensional
- Classification
  - If $\mathbf{f}_1(\mathbf{x})$ is largest, decide class 1
  - If $\mathbf{f}_2(\mathbf{x})$ is largest, decide class 2
  - If $\mathbf{f}_3(\mathbf{x})$ is largest, decide class 3

Multilayer NN : Multiple Classes



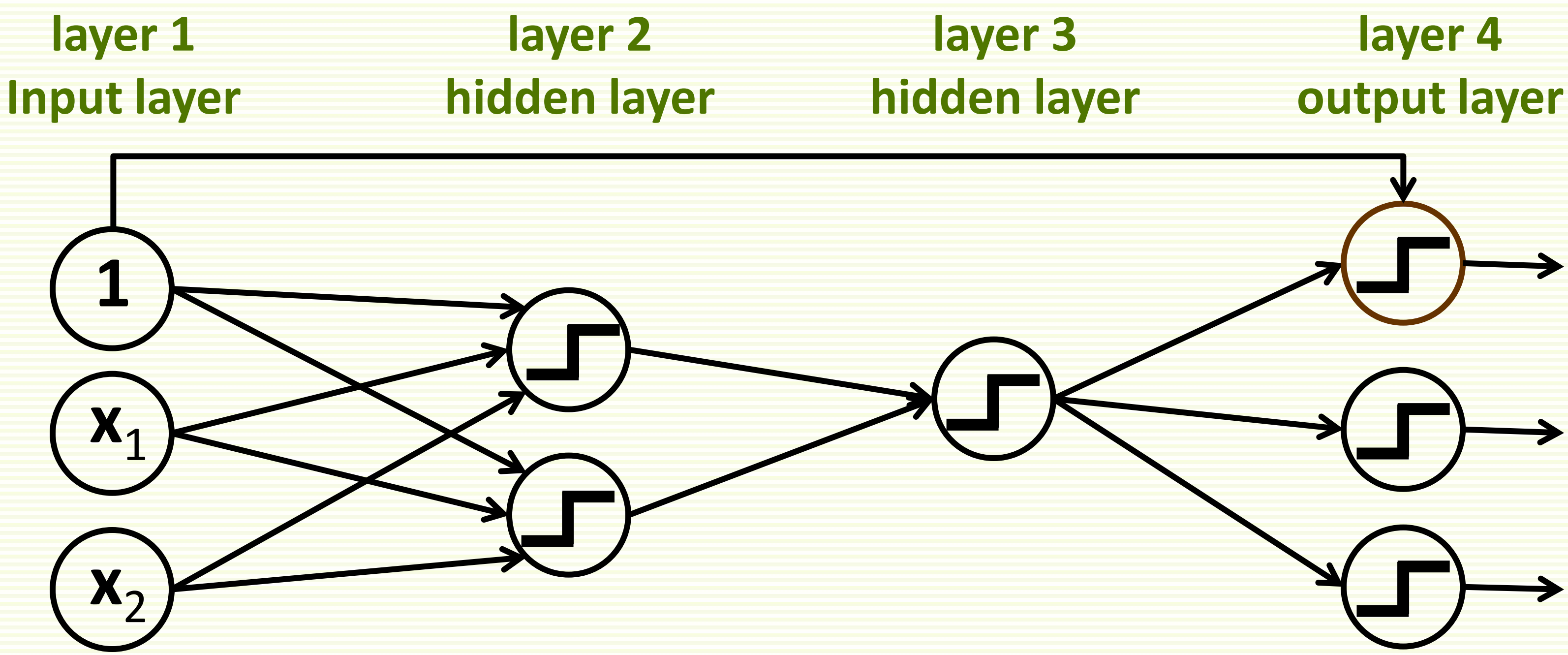
- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, can draw in layer 1, or each layer has one

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

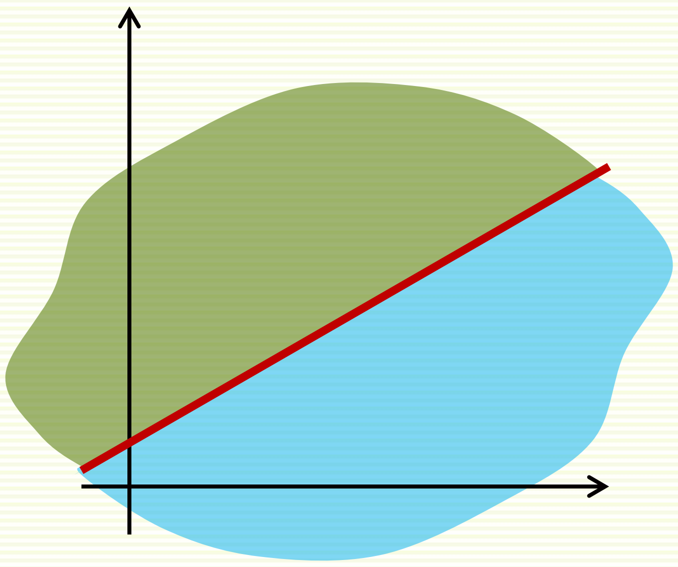
Multilayer NN : Overview

- 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
    - notation gets ugly
- To train NN, just as before
  - formulate an objective or *loss* function **L**(**w**)
  - optimize it with gradient descent
    - lots of heuristics to get gradient descent work well enough

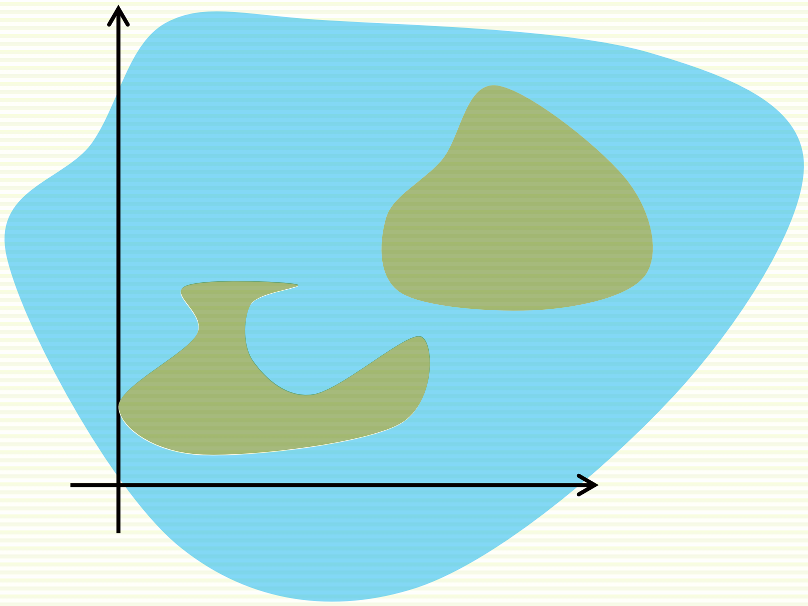
Multilayer NN : Expressive Power

- 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



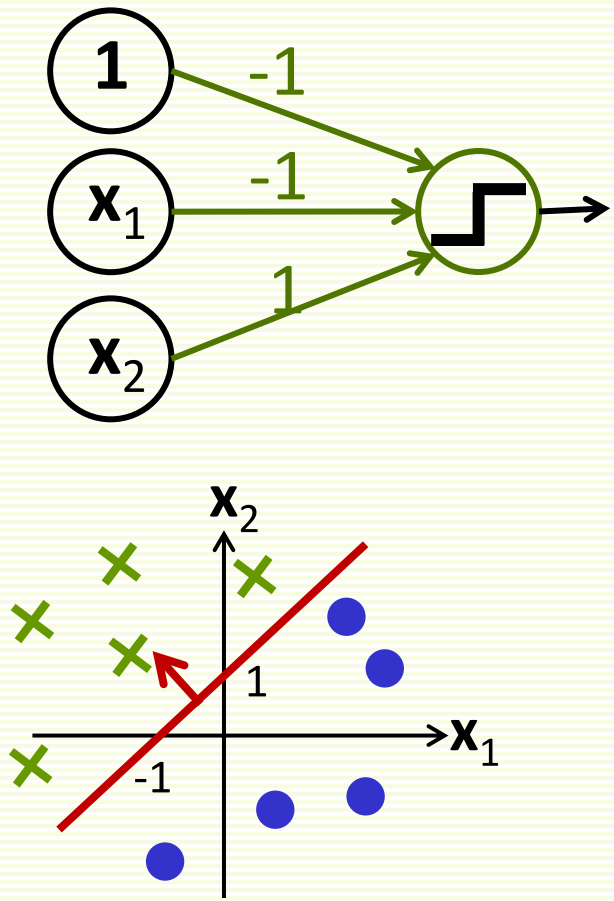
- Perceptron (single layer neural net)



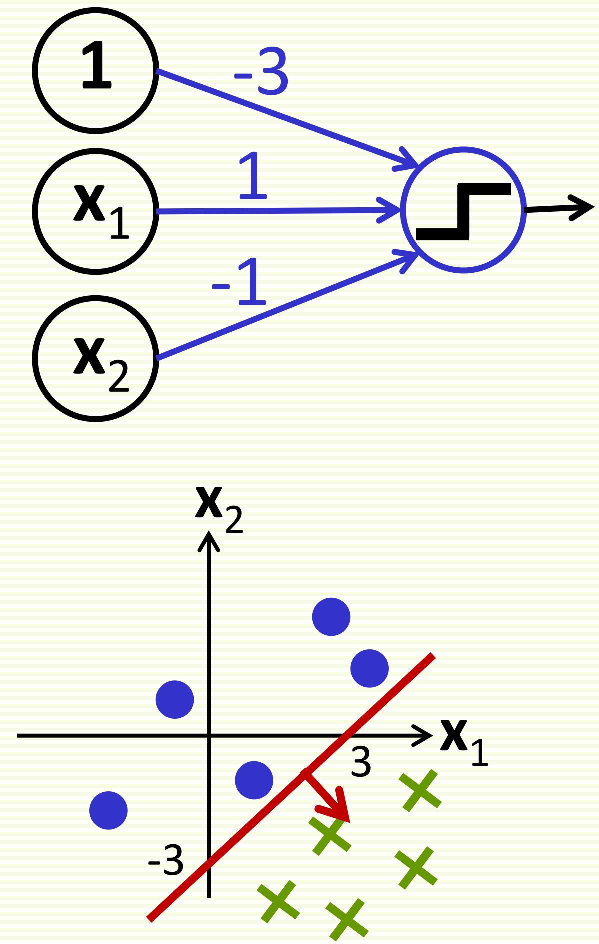
- Multilayer NN
- Arbitrarily complex decision regions
- Even not contiguous

Multilayer NN : Nonlinear Boundary Example

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

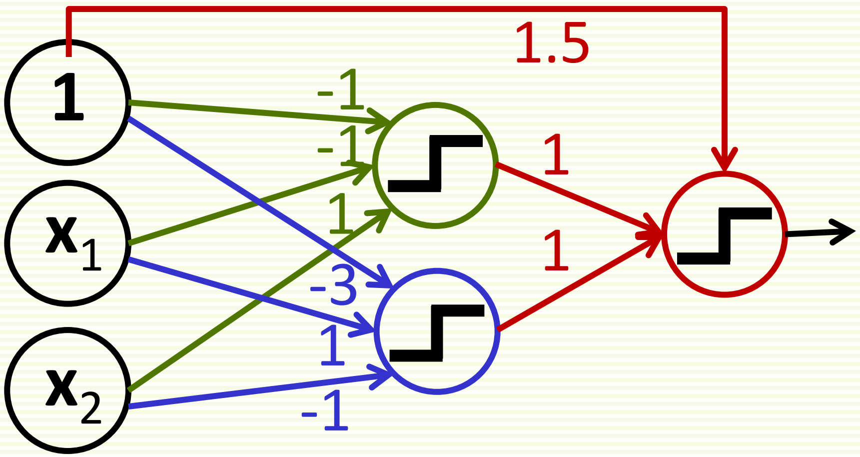


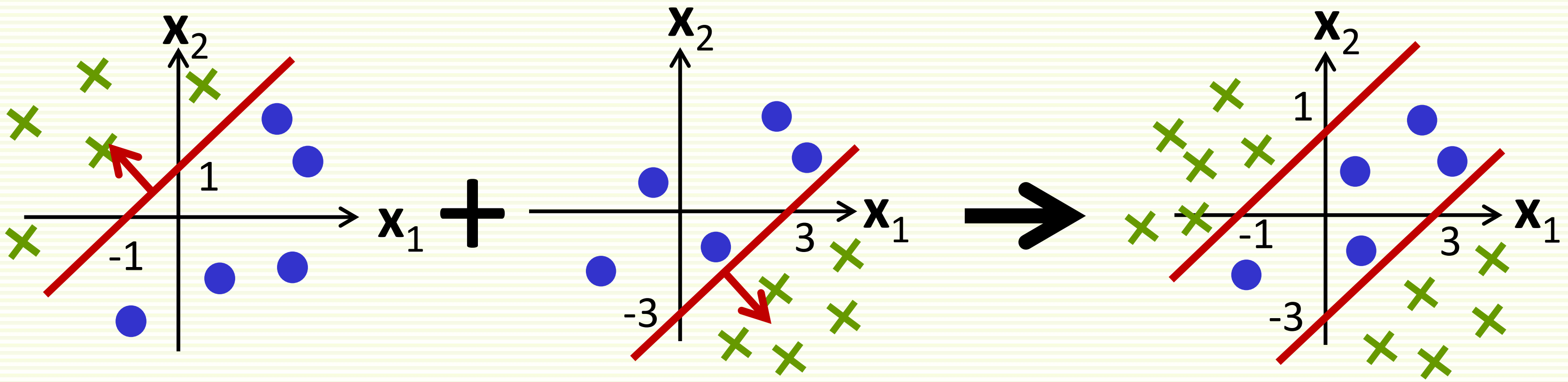
$\mathbf{x}_1 - \mathbf{x}_2 - 3 > 0 \Rightarrow \text{class } 1$



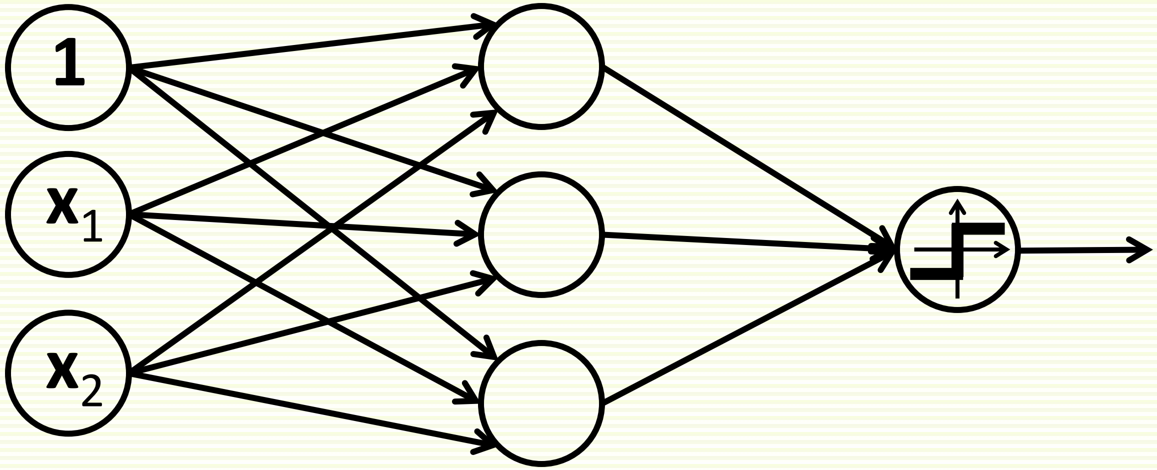
Multilayer NN : Nonlinear Boundary Example

- Combine two Perceptrons into a 3 layer NN



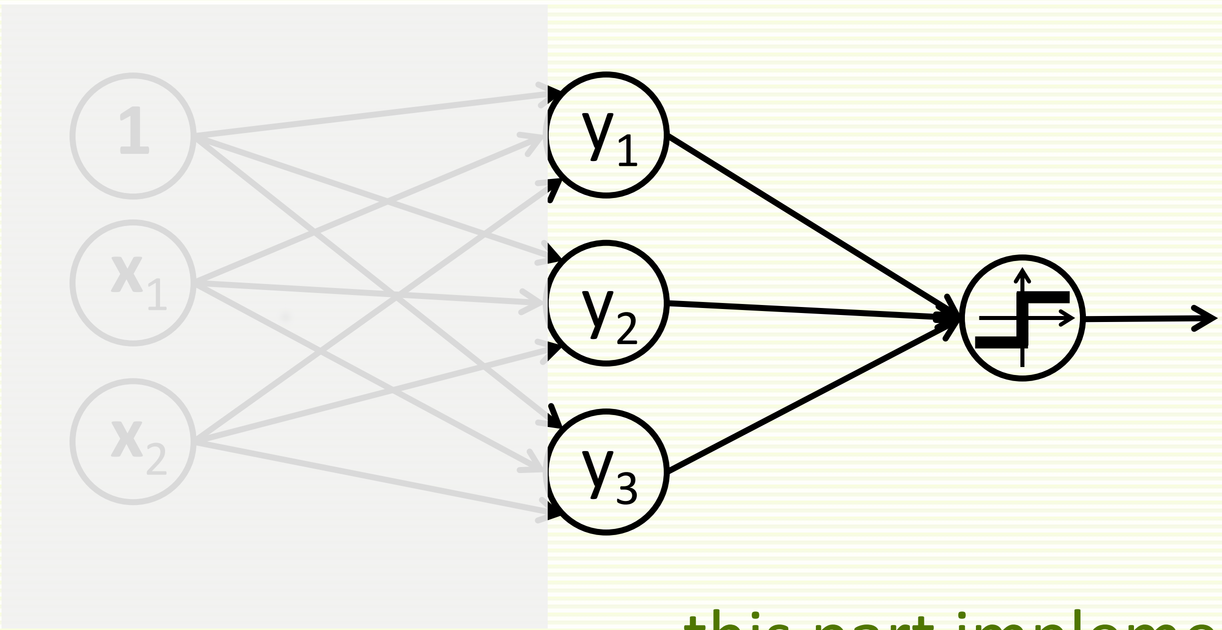


Multilayer NN as Non-Linear Feature Mapping



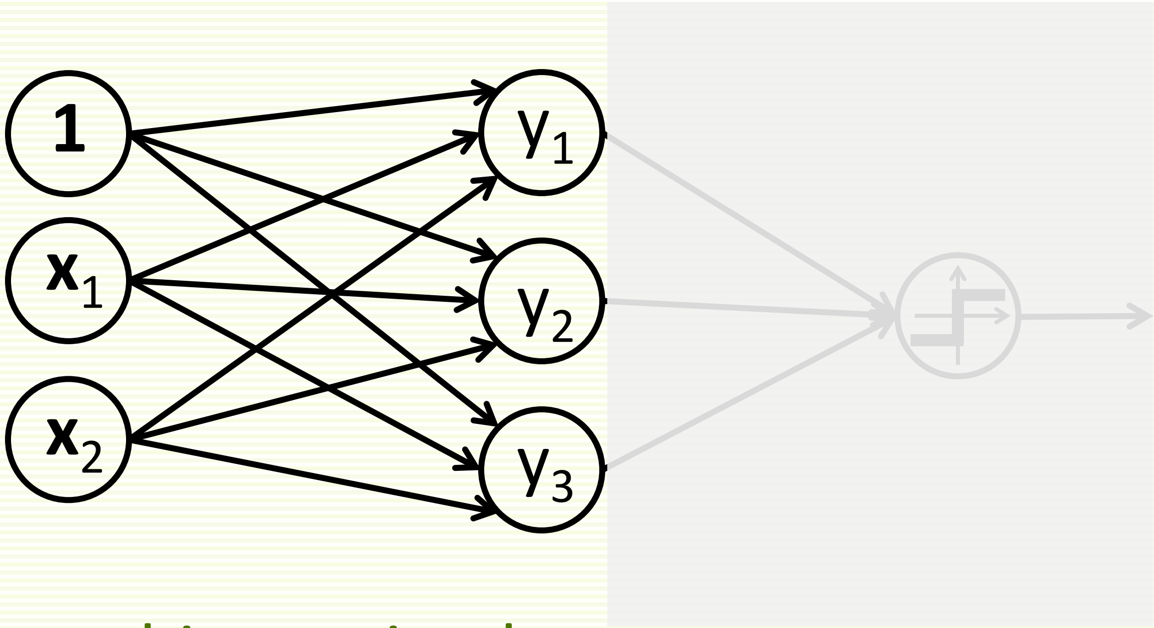
- Interpretation
  - 1 hidden layer maps input features to new features
  - next layer then applies linear classifier to the new features

Multilayer NN as Non-Linear Feature Mapping



this part implements Perceptron (liner classifier)

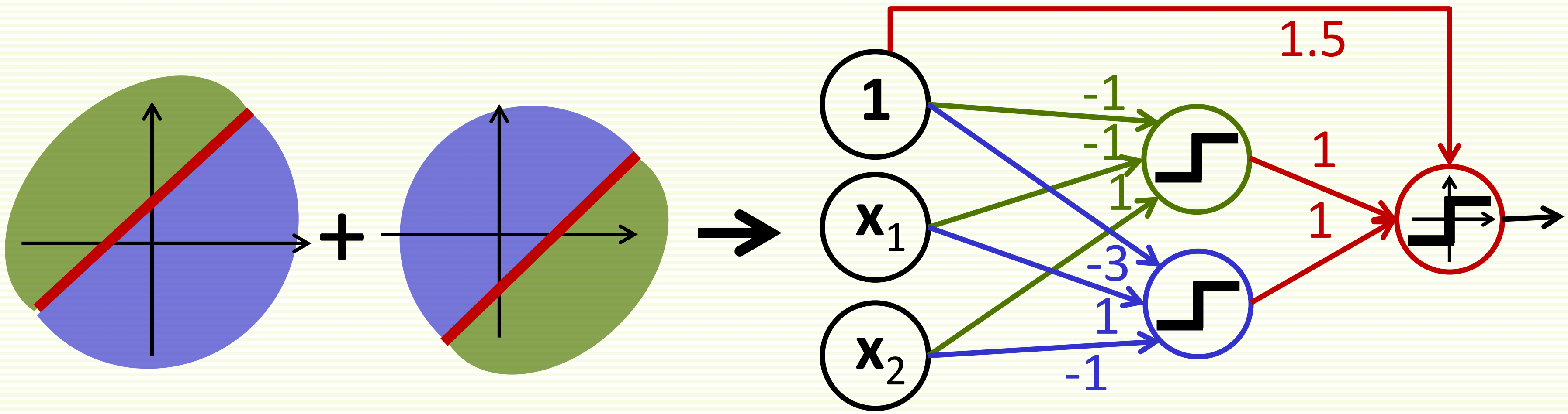
Multilayer NN as Non-Linear Feature Mapping

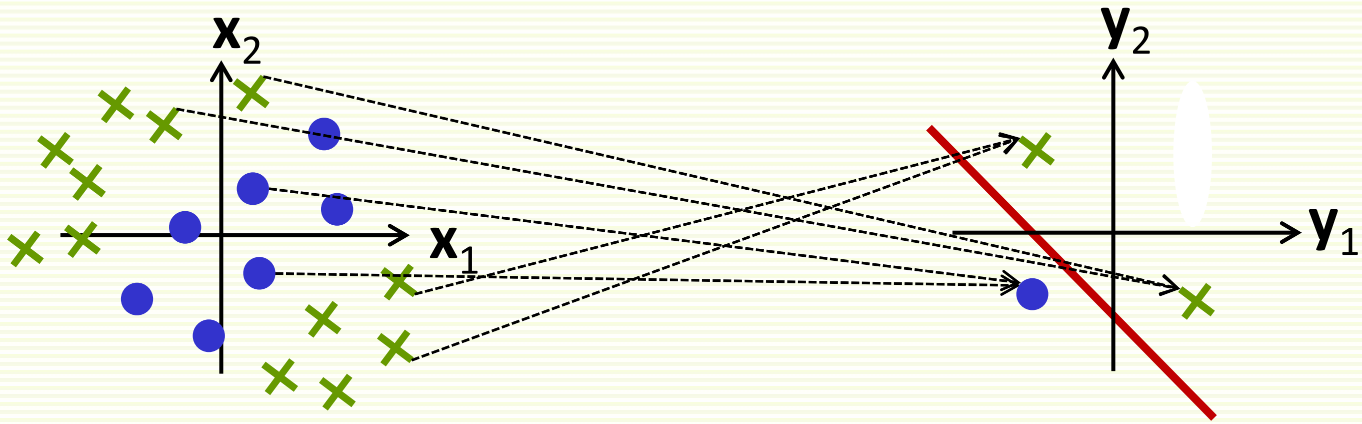


this part implements
mapping to new features **y**

Multilayer NN as Non-Linear Feature Mapping

- Consider 3 layer NN example we saw previously:



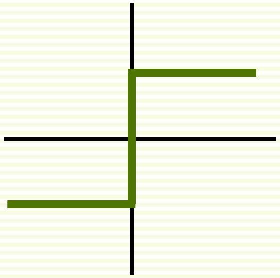


non linearly separable in the original feature space

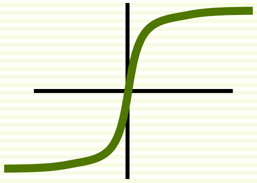
linearly separable in the new feature space

Multi Layer NN: Activation Function

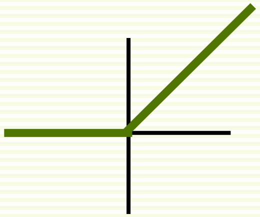
- **h**() = **sign**() does not work for gradient descent



- 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



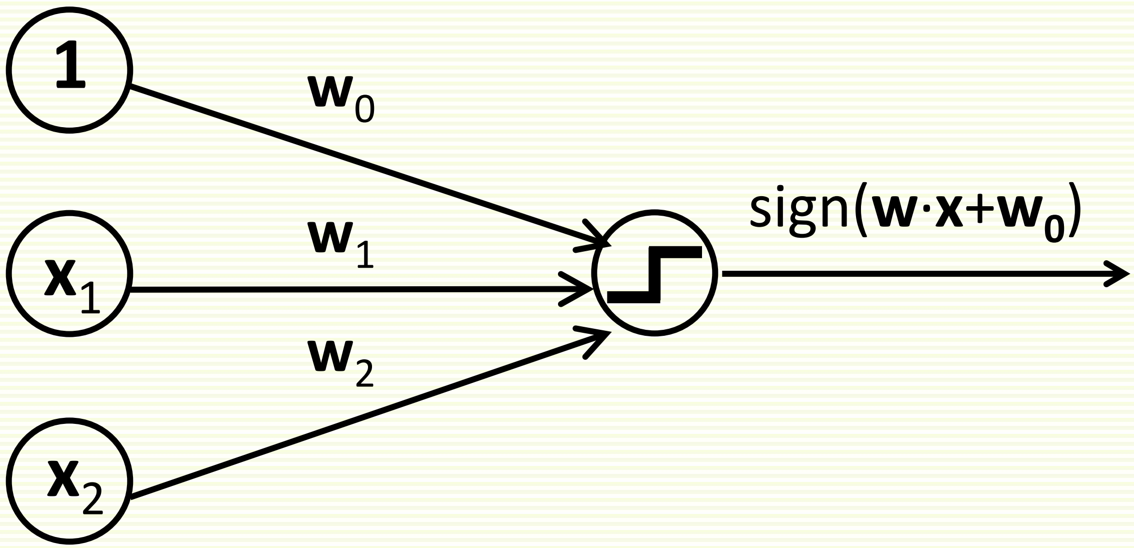
Multilayer NN: Modes of Operation

- 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: Vector Notation

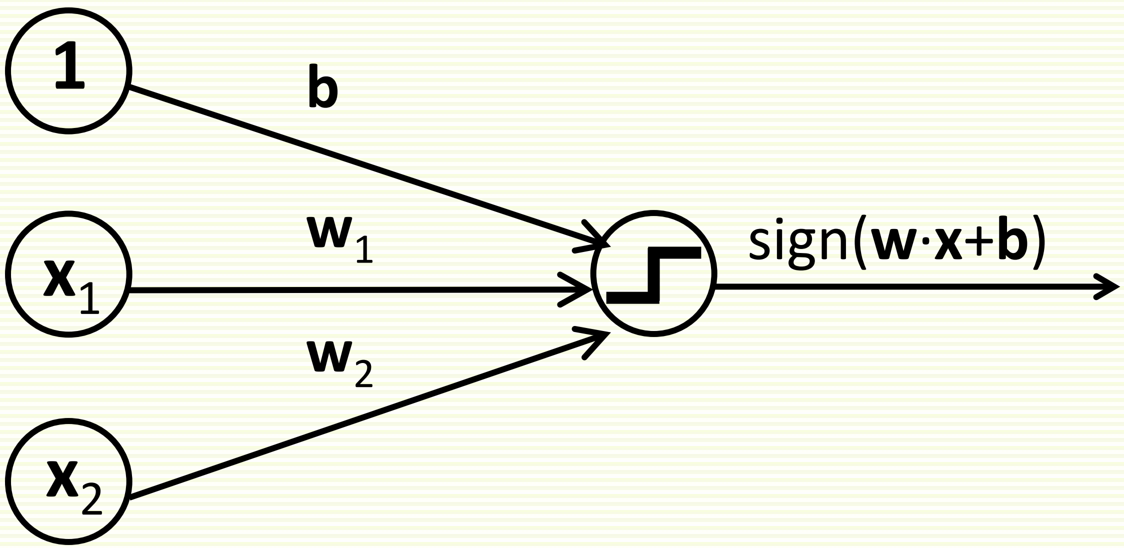
- Convenient compact notation
- For Perceptron



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

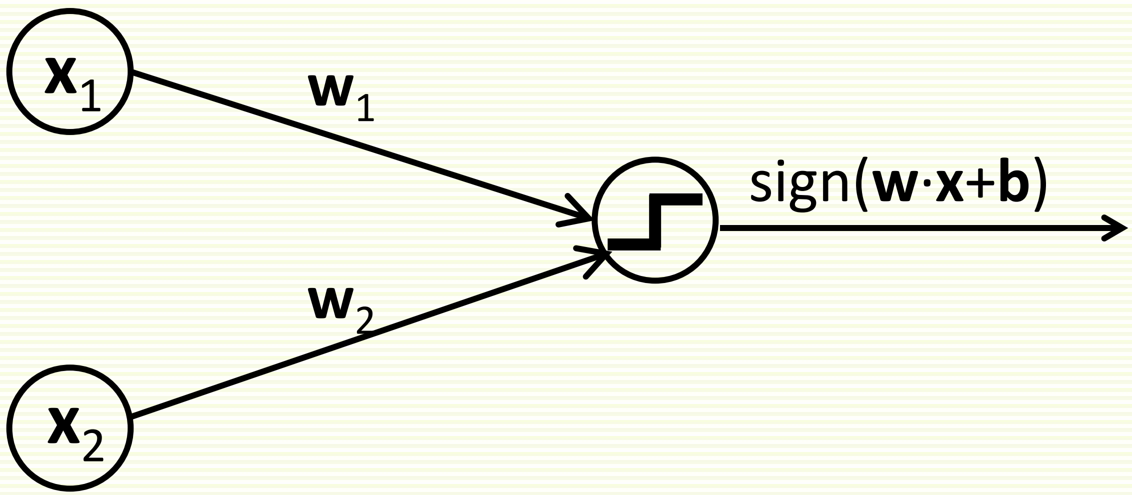
Multilayer NN: Vector Notation

- Change notation a bit

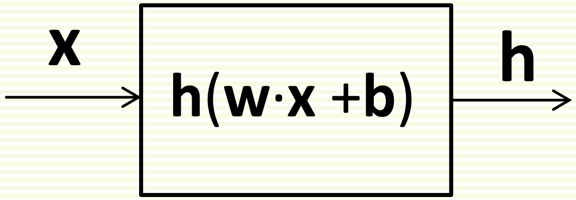


Multilayer NN: Vector Notation

- Do not draw bias unit

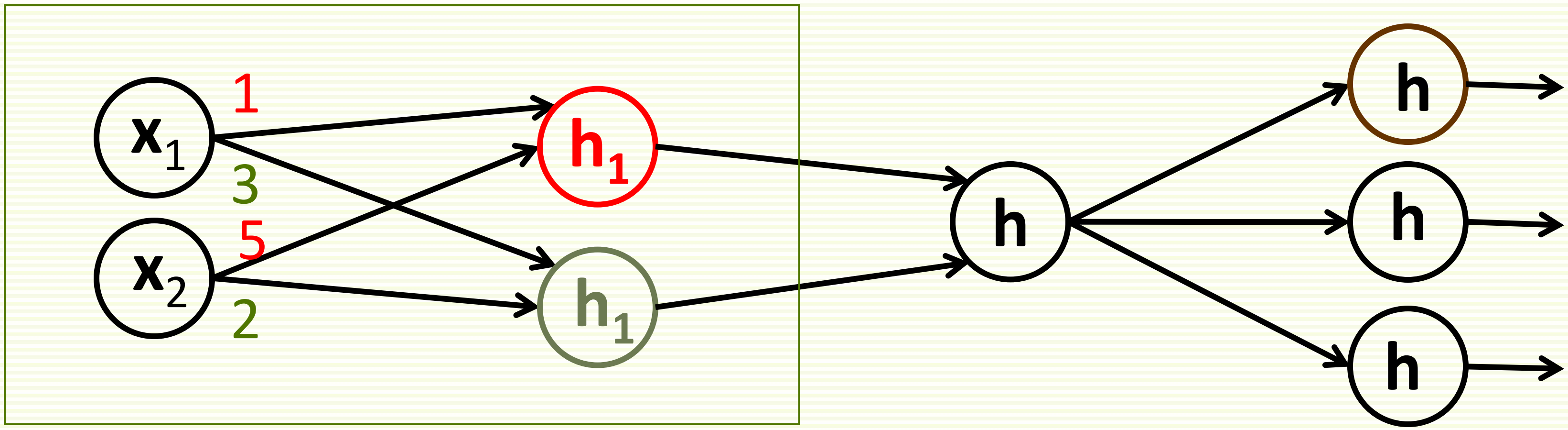


- Compact picture
  - $\mathbf{h}$(t) = sign(t)

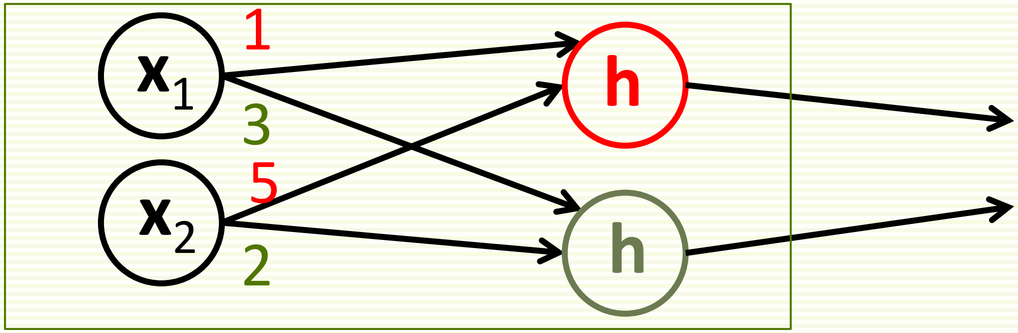


Multilayer NN: Vector Notation

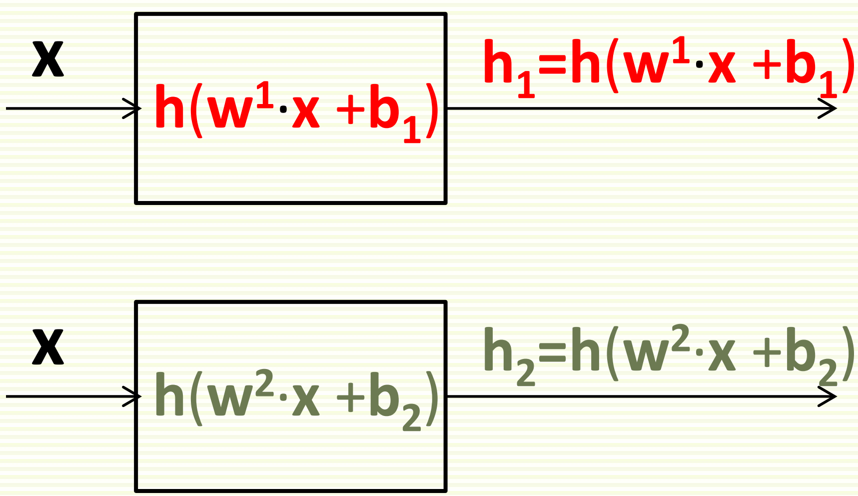
- Consider the first layer (2 perceptrons)



Multilayer NN: Vector Notation for First Layer



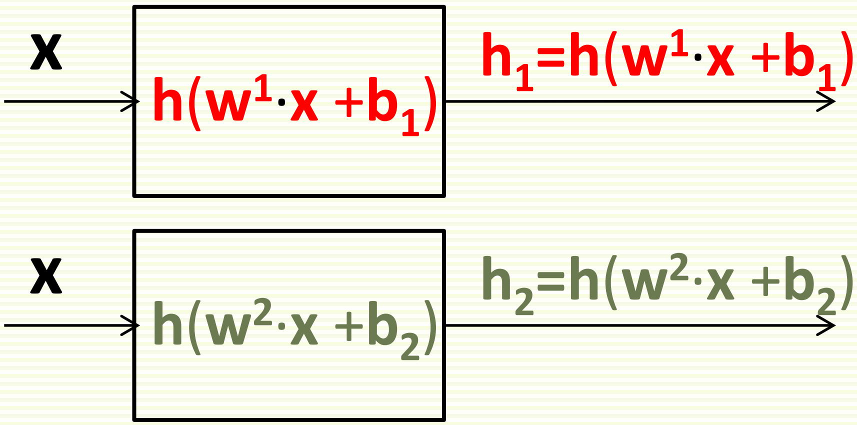
- Red perceptron has weights w^1 and bias b_1
- Green perceptron has weights w^2 and bias b_2



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

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

Multilayer NN: Vector Notation for First Layer

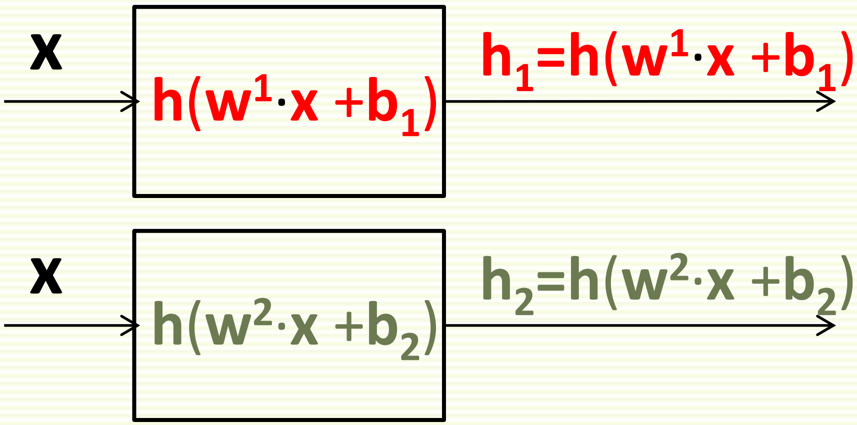


$$\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w^1 \cdot x \\ w^2 \cdot x \end{bmatrix}$$

$$\mathbf{W^1} \cdot \mathbf{x}$$

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

Multilayer NN: Vector Notation for First Layer

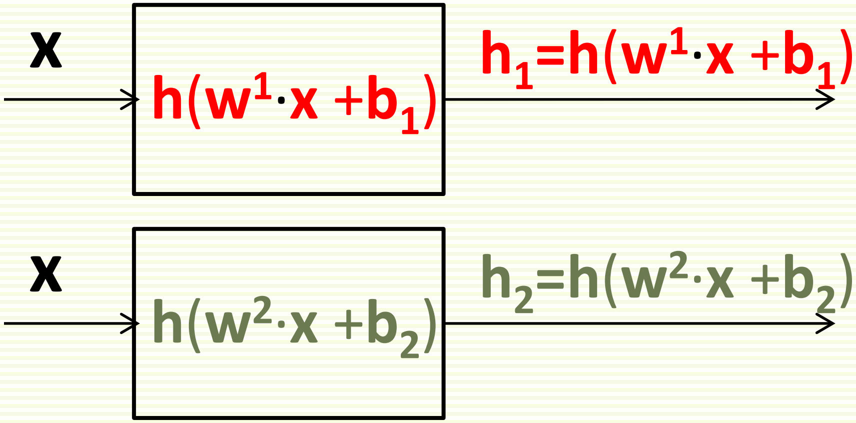


$$\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w^1 \cdot x + b_1 \\ w^2 \cdot x + b_2 \end{bmatrix}$$

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

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

Multilayer NN: Vector Notation for First Layer



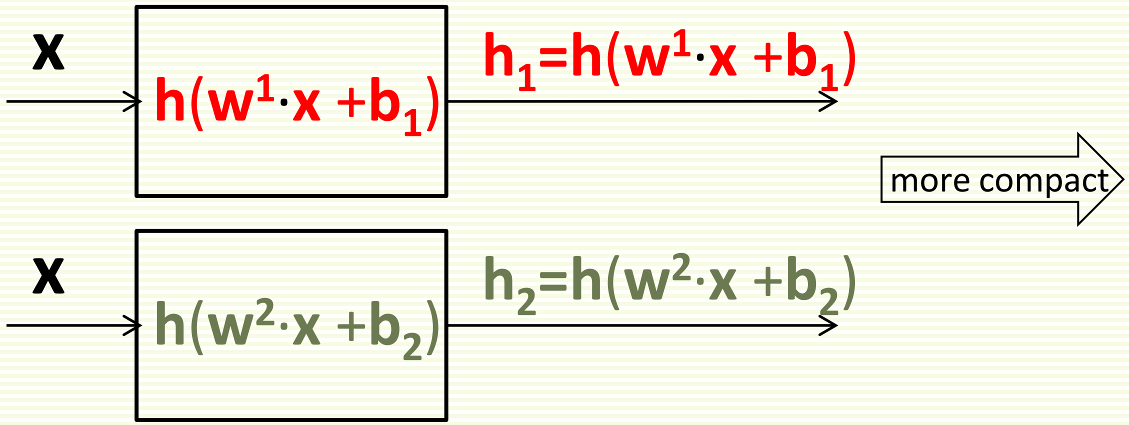
$$\mathbf{h}\left(\begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

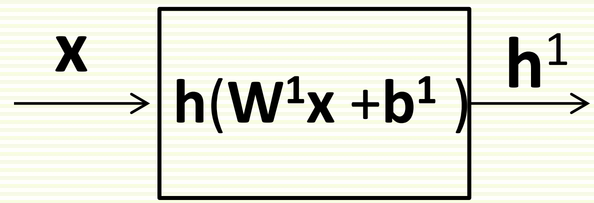
$$\mathbf{h}(\mathbf{W}^1 \cdot \mathbf{x} + \mathbf{b}^1)$$

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

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

Multilayer NN: Vector Notation for First Layer



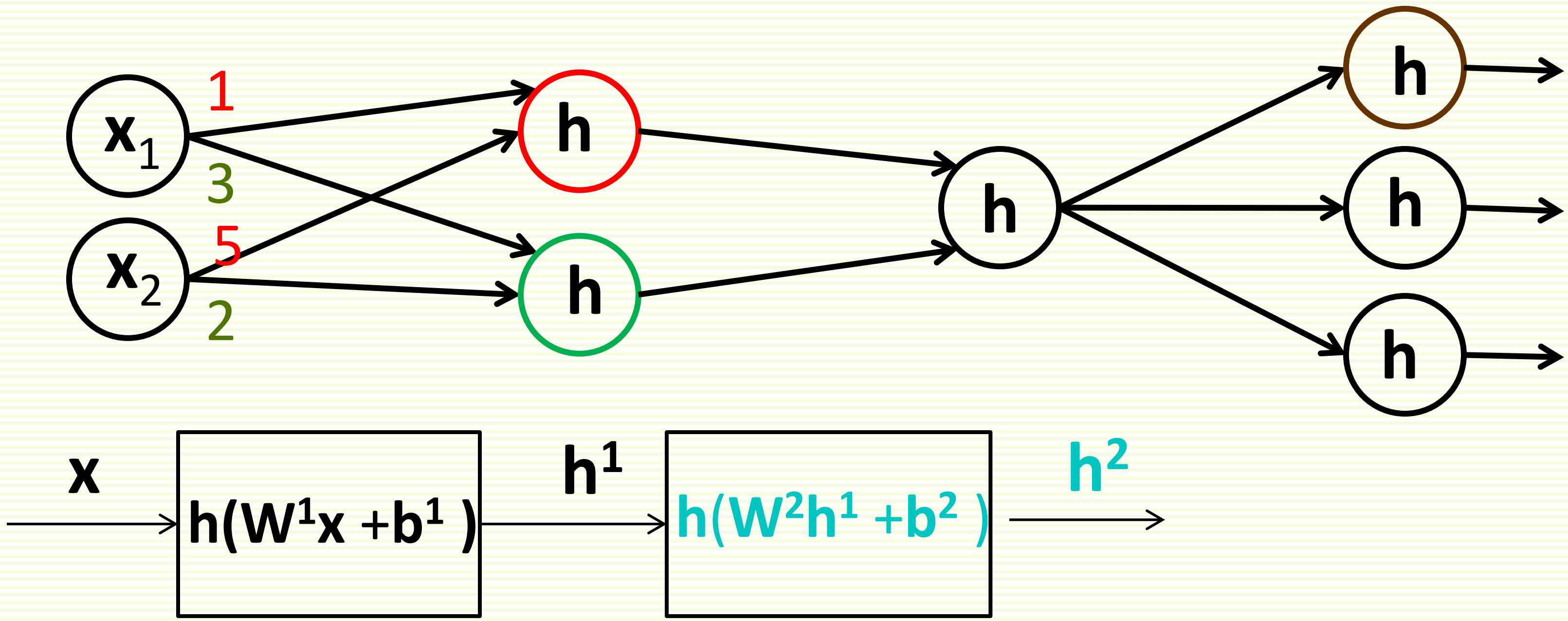


$$\mathbf{W}^1 = \begin{bmatrix} 1 & 5 \\ 3 & 2 \end{bmatrix} \qquad \mathbf{b}^1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

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

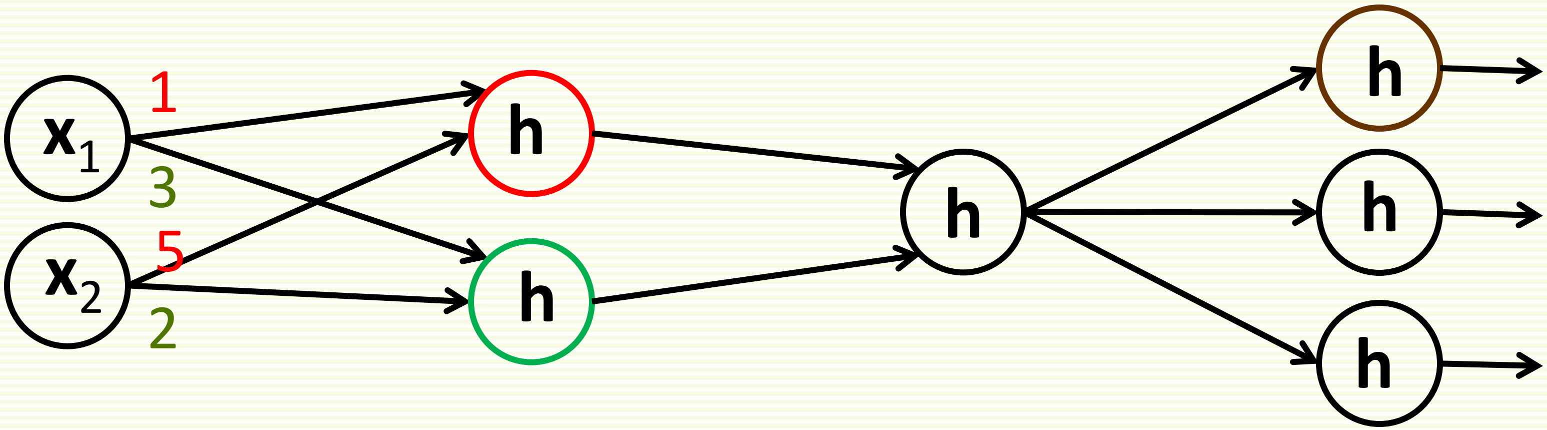
- **h**(**v**) for vector **v** means applying **h** to each component of **v**

Multilayer NN: Vector Notation for Next Layer

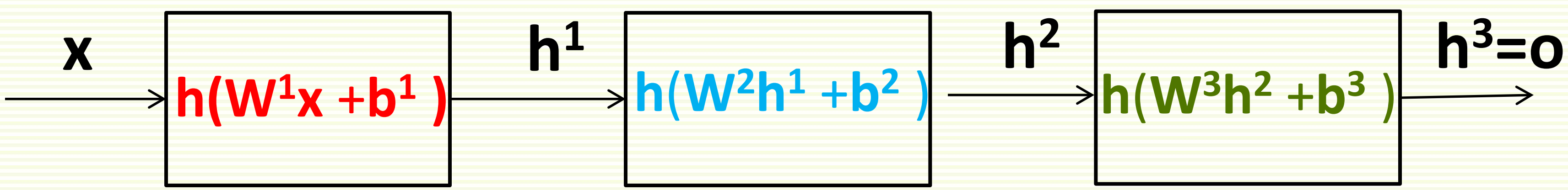


- $\mathbf{W}^2$ is a matrix of weights between hidden layer 1 and 2
  - $\mathbf{W}^2(\mathbf{r},\mathbf{c})$ is weight from unit **c** to unit **r**
- $\mathbf{b}^2$ is a vector of bias weights for second hidden layer
  - b^2_r is bias weight of unit **r** in second layer
- $\mathbf{h}^2$ is a vector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

Multilayer NN: Vector Notation, all Layers



- Complete depiction



- $\mathbf{h}^3 = \mathbf{o}$ is vector from the output layer
- $\mathbf{o} = \mathbf{h}(\mathbf{W}^3\mathbf{h}^2 + \mathbf{b}^3)$

$$= \mathbf{h}(\mathbf{W}^3\mathbf{h}(\mathbf{W}^2\mathbf{h}^1 + \mathbf{b}^2) + \mathbf{b}^3)$$

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

Multilayer NN: Output Representation

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

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

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

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

Training NN: Squared Difference Loss

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

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

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

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

- **f** depends on **W**, but too cumbersome to write **f**(**x**,**W**) everywhere

Training NN: Squared Difference Loss

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

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

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

- Gradient descent

initialize **w** to random
choose **ε**, α
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: Cross Entropy Loss

- Squared error loss is usually not recommended for classification
- Better Loss function for classification: Cross Entropy
- First put the output **o** 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: Cross Entropy Loss

- One sample cross entropy loss, dropping superscripts from $\mathbf{x}^i$, $\mathbf{y}^i$:

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

- If sample **x** is of class **k**, then the above is equivalent to

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

  - this loss function is also called –log 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

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

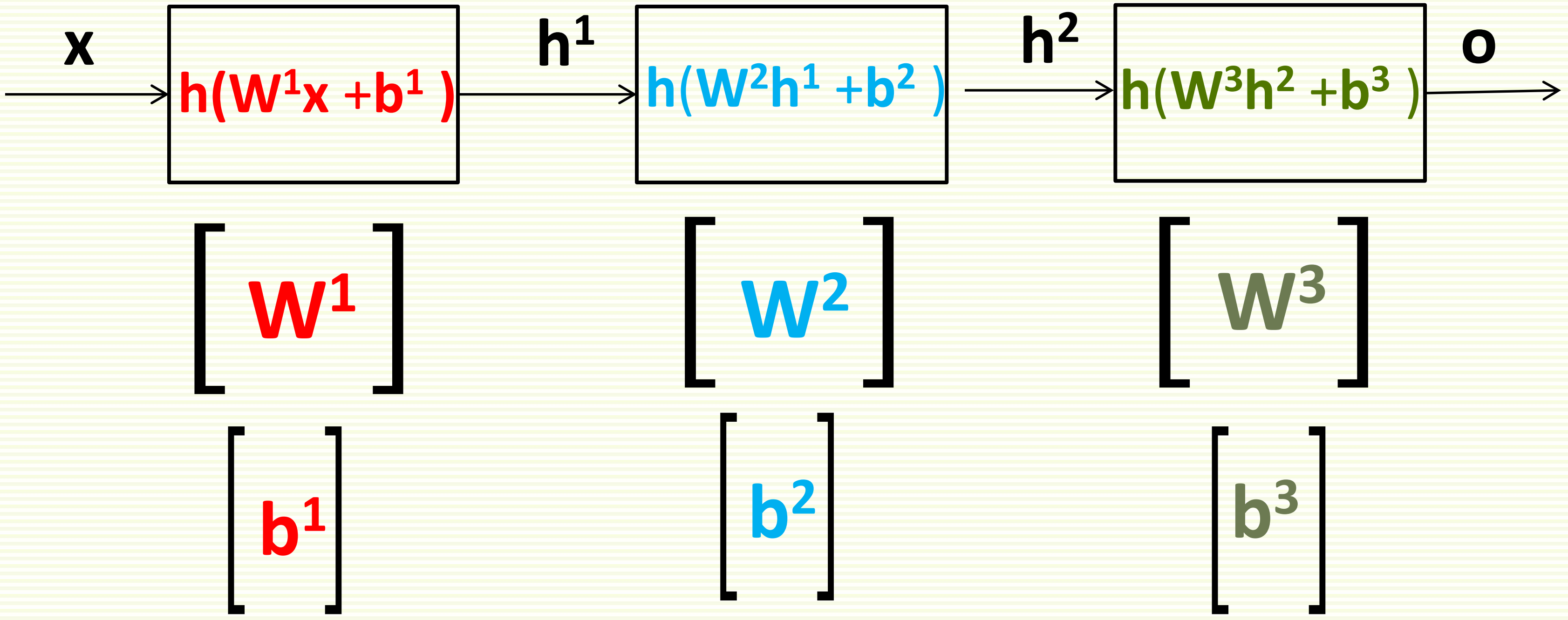
  - where α is the learning rate

- Update weight:

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

Training NN: -Log Loss Function

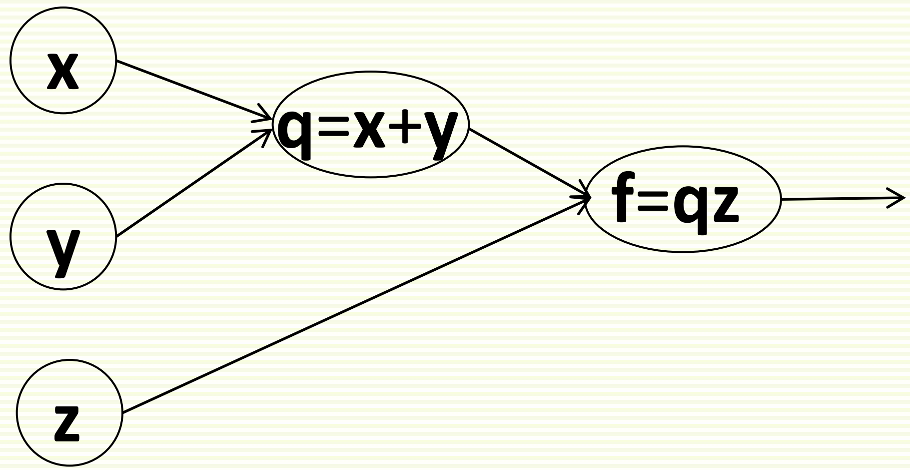
- How many weights do we have in our network?



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

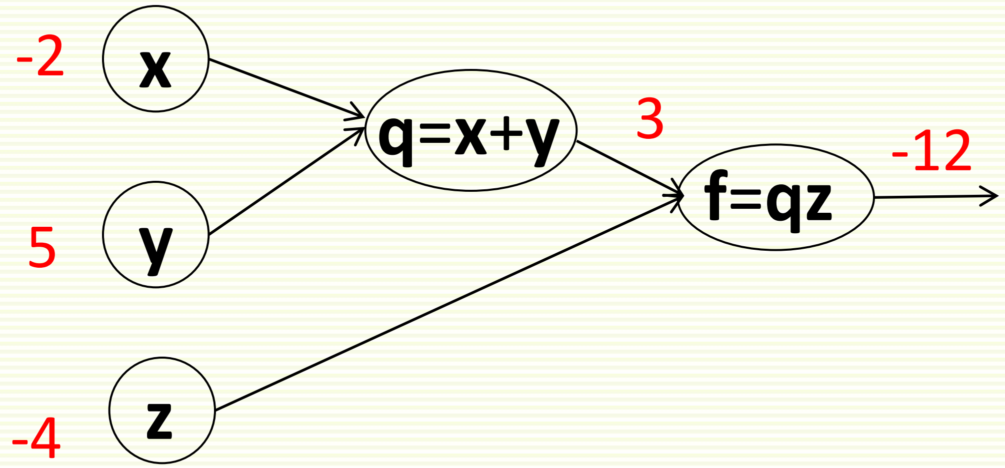
Computing Derivatives: Small Example

- 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)**

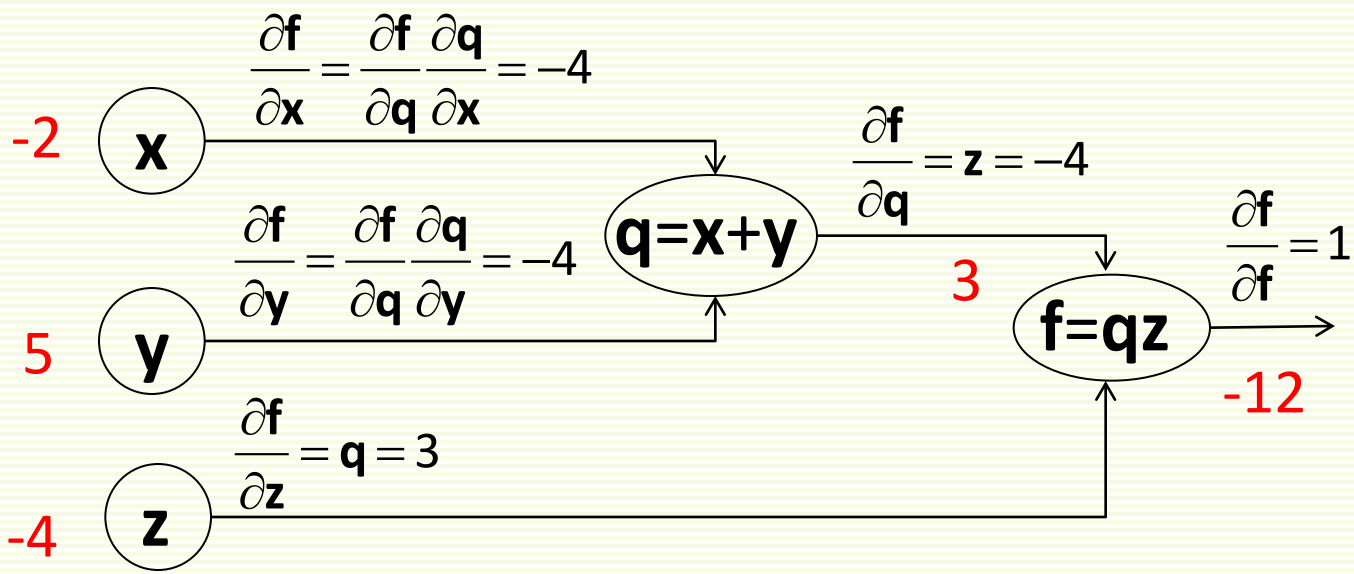


Computing Derivatives: Small Example

- Small network $\mathbf{f(x,y,z) = (x+y)z}$
- Rewrite using $\mathbf{q = x + y} \;\Rightarrow\; \mathbf{f(x,y,z) = qz}$
- 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

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

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



Computing Derivatives: Chain of Chain Rule

direction of computation

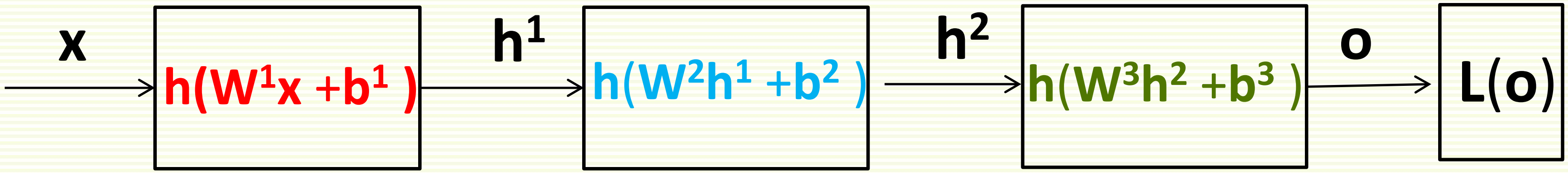
- Compute $\frac{\partial \mathbf{d}}{\partial}$ from the end backwards
  - for each edge, with respect to the main variable at edge origin
  - using chain rule with respect to the variable at edge end, if needed

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

$$\frac{\partial \mathbf{d}}{\partial \mathbf{a}} = \underbrace{\frac{\partial \mathbf{d}}{\partial \mathbf{b}}}_{\text{prev}} \underbrace{\frac{\partial \mathbf{b}}{\partial \mathbf{a}}}_{\text{local}} \qquad \frac{\partial \mathbf{d}}{\partial \mathbf{b}} = \underbrace{\frac{\partial \mathbf{d}}{\partial \mathbf{c}}}_{\text{prev}} \underbrace{\frac{\partial \mathbf{c}}{\partial \mathbf{b}}}_{\text{local}} \qquad \underbrace{\frac{\partial \mathbf{d}}{\partial \mathbf{c}}}_{\text{local}}$$

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

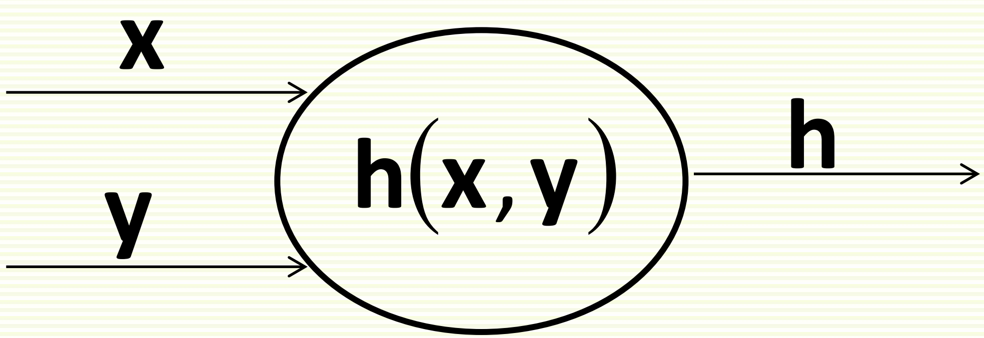
Computing Derivatives Backwards



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

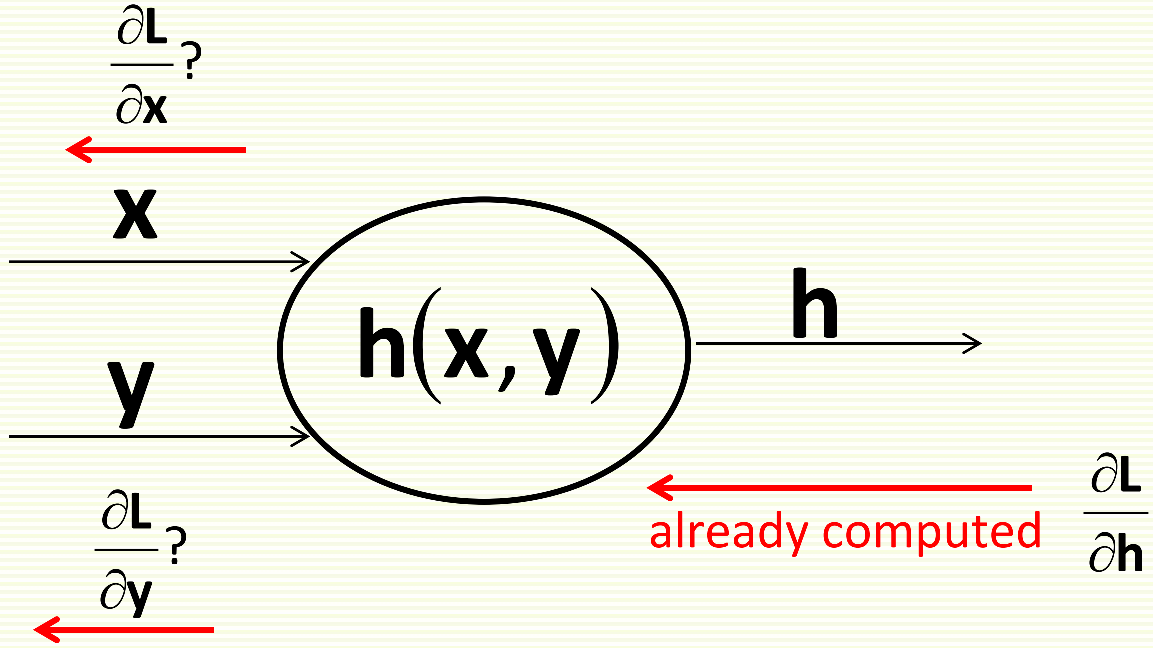
Computing Derivatives: Look at One Node

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



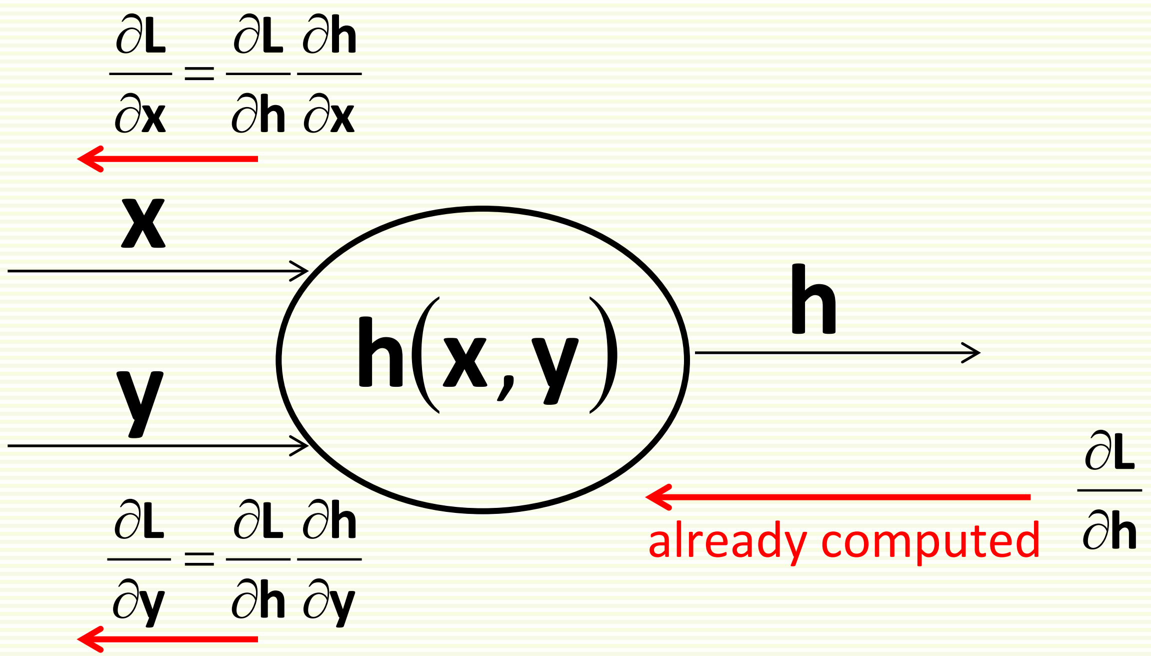
Computing Derivatives: Look at One Node

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



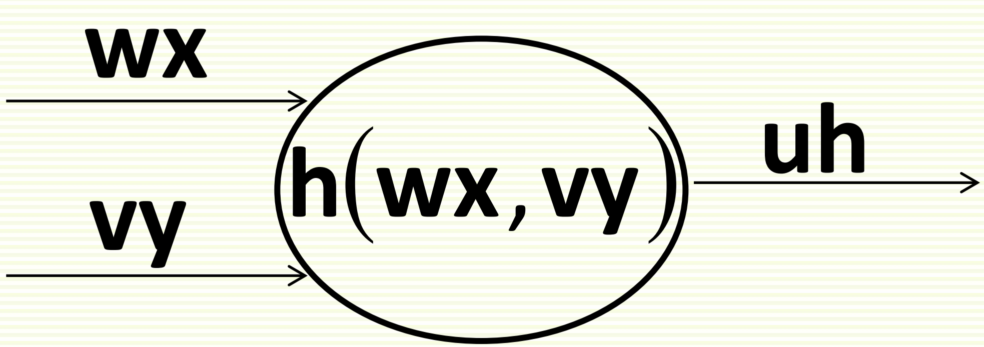
Computing Derivatives: Look at One Node

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

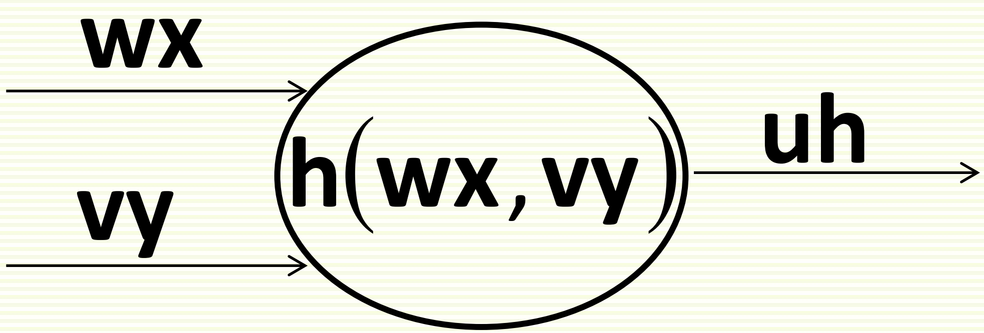


Computing Derivatives: Look at One Node

- 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**

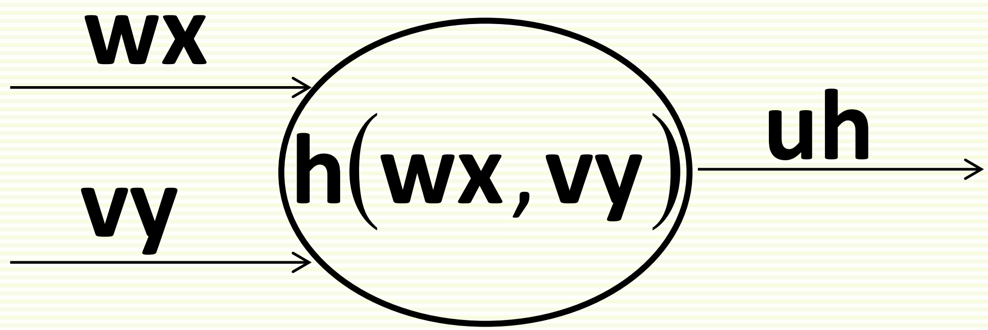


Computing Derivatives: Look at One Node



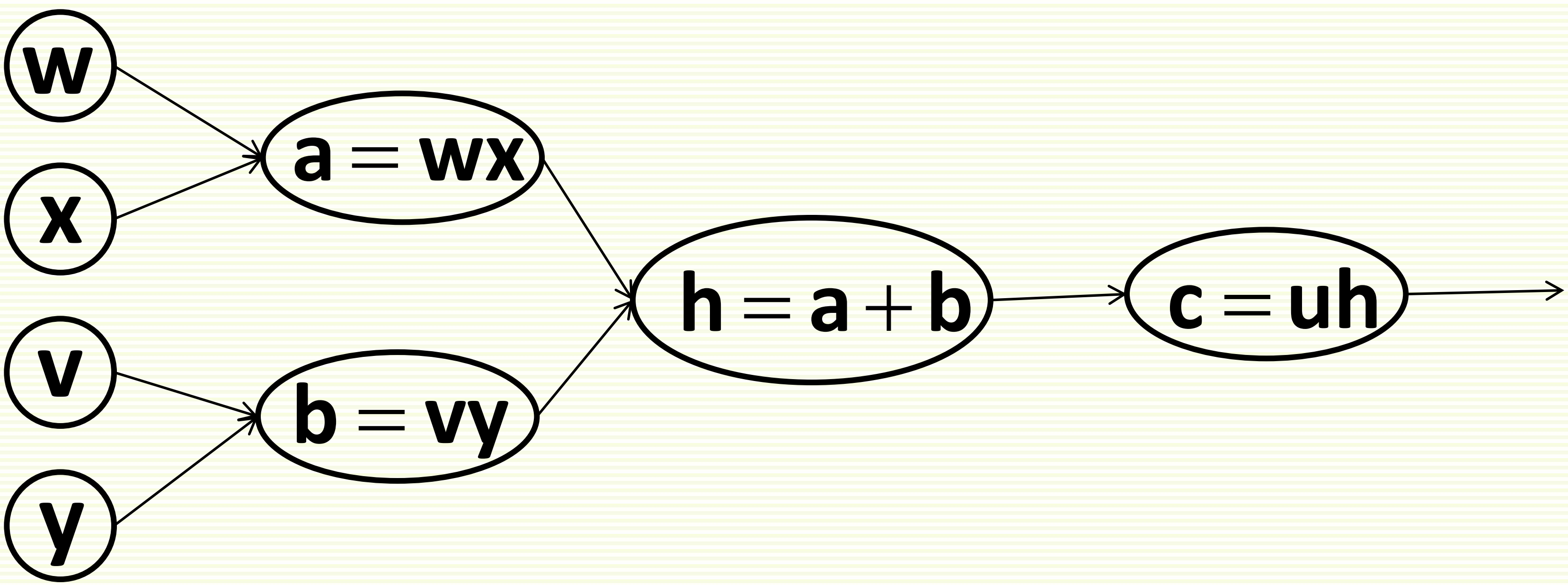
- To be concrete, let **h**(**i**,**j**) = **i** + **j**

Computing Derivatives: Look at One Node

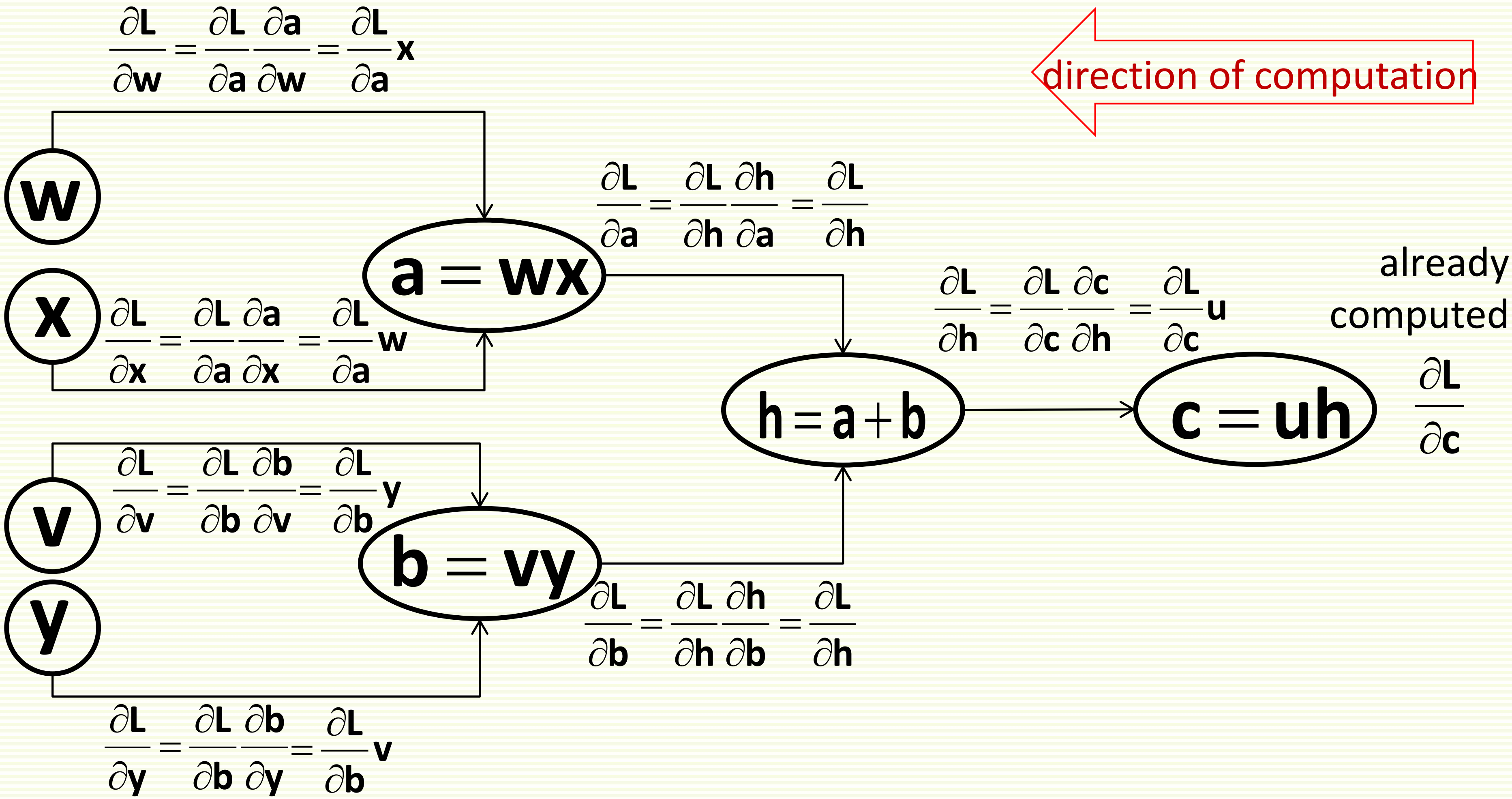


- **h**(**i**,**j**) = **i** + **j**

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

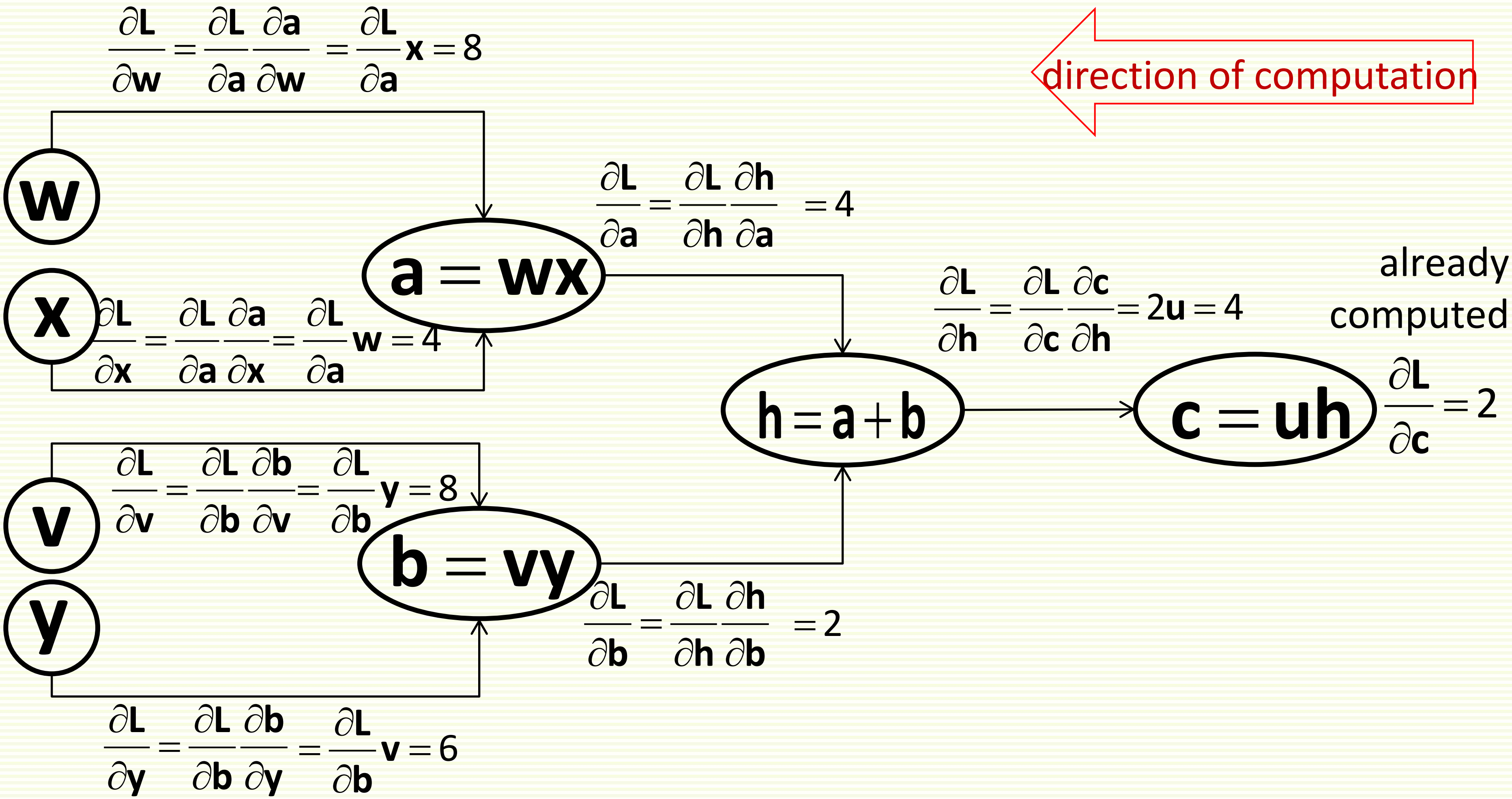


Computing Derivatives: Look at One Node



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

Computing Derivatives: Look at One Node



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

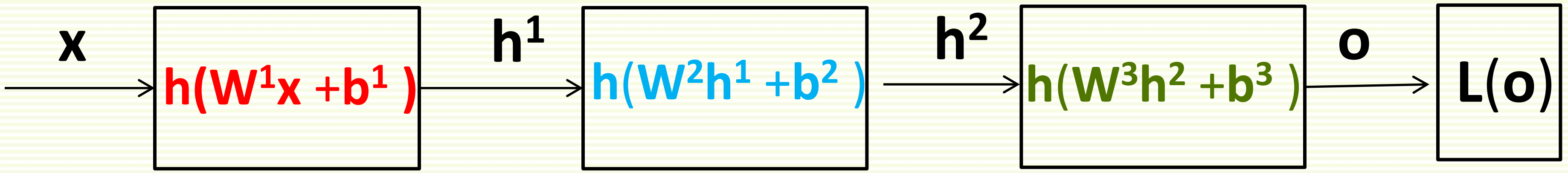
Computing Derivatives: Staging Computation

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

$$\begin{tikzcd} {} \arrow[r, "\mathbf{x}"] & \mathbf{h} = 1/\mathbf{x} \arrow[r] & \mathbf{g} = \exp(\mathbf{h}) \arrow[r] & {} \end{tikzcd}$$

Computing Derivatives: Vector Notation

- Inputs outputs are often vectors



- **h**(a) is a function from $\mathbf{R}^n$ to $\mathbf{R}^m$
- Chain rule generalizes to vector functions

Computing Derivatives: Vector Notation

- Let $\mathbf{f}(\mathbf{x})$: $\mathbf{R}^n \rightarrow \mathbf{R}^m$,
  - $\mathbf{x}$ is $\mathbf{n}$-dimensional vector and output $\mathbf{f}(\mathbf{x})$ is $\mathbf{m}$-dimensional vector
- Jacobian matrix
  - has $\mathbf{m}$ rows and $\mathbf{n}$ columns
  - has $\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}$ in row $\mathbf{i}$, column $\mathbf{j}$
- Example $\mathbf{f}(\mathbf{x})$: $\mathbf{R}^3 \rightarrow \mathbf{R}^2$, Jacobian matrix

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_3} \\ \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_3} \end{bmatrix}$$

Computing Derivatives: Vector Notation

- $\mathbf{f}(\mathbf{x})$: $\mathbf{R}^n \rightarrow \mathbf{R}^m$ and $\mathbf{g}(\mathbf{x})$: $\mathbf{R}^k \rightarrow \mathbf{R}^n$
- $\mathbf{f}(\mathbf{g}(\mathbf{x}))$: $\mathbf{R}^k \rightarrow \mathbf{R}^m$
- Chain rule for vector functions

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

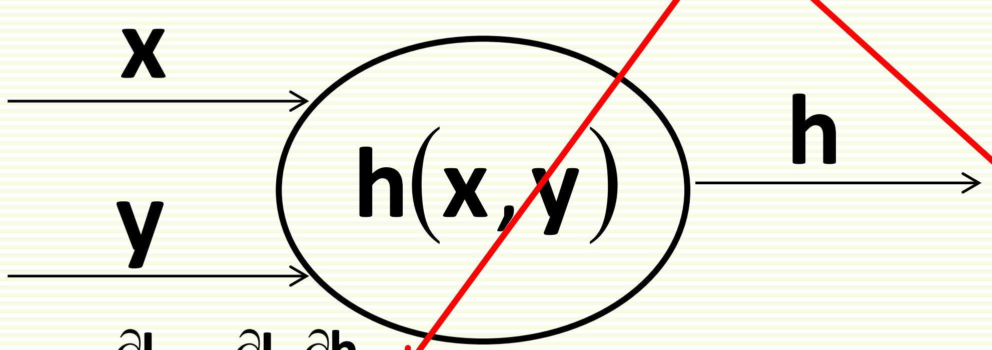
Jacobian matrices

Vector Notation: Look at One Node

- **h**, **x**, **y** are vectors
- already computed Jacobian $\frac{\partial \mathbf{L}}{\partial \mathbf{h}}$
- Need Jacobians $\frac{\partial \mathbf{L}}{\partial \mathbf{x}}, \frac{\partial \mathbf{L}}{\partial \mathbf{y}}$
- Easy to compute local node Jacobians $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}, \frac{\partial \mathbf{h}}{\partial \mathbf{y}}$

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

Jacobian matrices

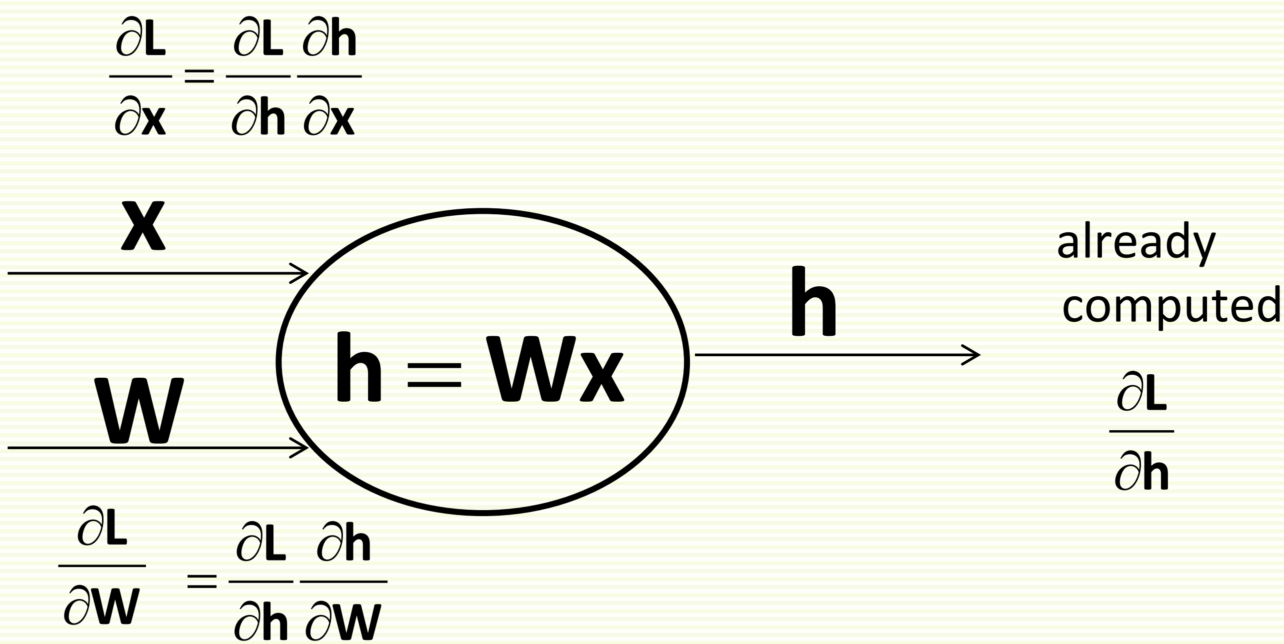


already computed $\frac{\partial \mathbf{L}}{\partial \mathbf{h}}$

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

Vector Notation: Look at One Node

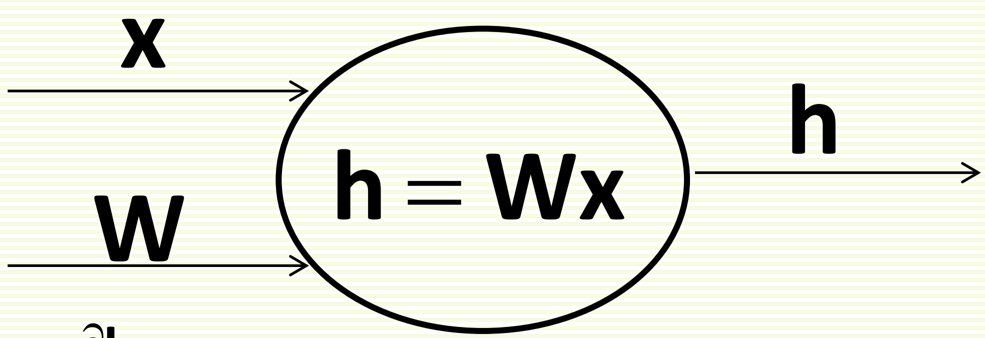
- Can apply to matrices (and tensors) as well
- But first vectorize matrix (or tensor)
- Say **W** is 10 x 5, stretch into 50x1 vector
- Still denote Jacobian by $\frac{\partial \mathbf{h}}{\partial \mathbf{W}}$



Vector Notation: Look at One Node

- Easy to compute local node Jacobians $\dfrac{\partial \mathbf{h}}{\partial \mathbf{x}}, \dfrac{\partial \mathbf{h}}{\partial \mathbf{W}}$
- But they can get very large (although sparse)
- Say **h** is 1000 x 1, **W** is 1000 x 500, then $\dfrac{\partial \mathbf{h}}{\partial \mathbf{W}}$ is 1000 x 500,000

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



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

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{W}}$$

Compact Vector Notation

- Assume loss **L** is a scalar
  - if not, can do derivation for each component independently
- Consider matrix $\mathbf{W} = \begin{bmatrix} \mathbf{w}_{11} & \cdots & \mathbf{w}_{1k} \\ \vdots & \cdots & \vdots \\ \mathbf{w}_{d1} & \cdots & \mathbf{w}_{dk} \end{bmatrix}$
- Organize derivatives in matrix the same shape as **W**, denoted with

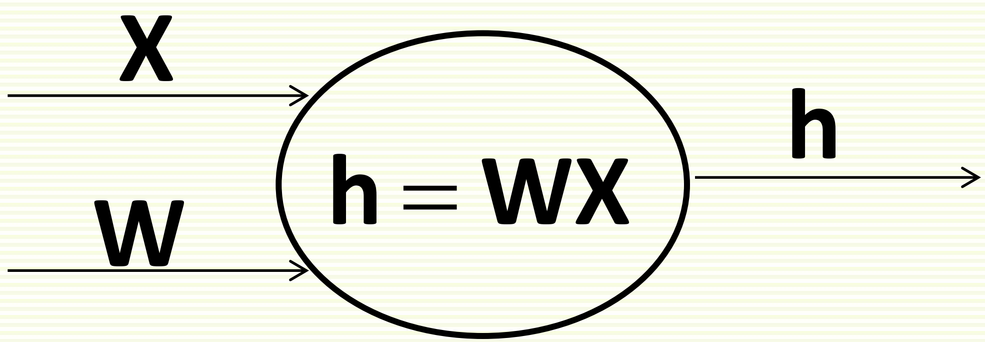
$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{\mathbf{o}}} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{1k}} \\ \vdots & \cdots & \vdots \\ \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{d1}} & \cdots & \frac{\partial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

- Contrast with Jacobian $\frac{\partial \mathbf{L}}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{11}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{1k}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{d1}} & \cdots & \frac{\partial \mathbf{L}}{\partial \mathbf{w}_{dk}} \end{bmatrix}$

Compact 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

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



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

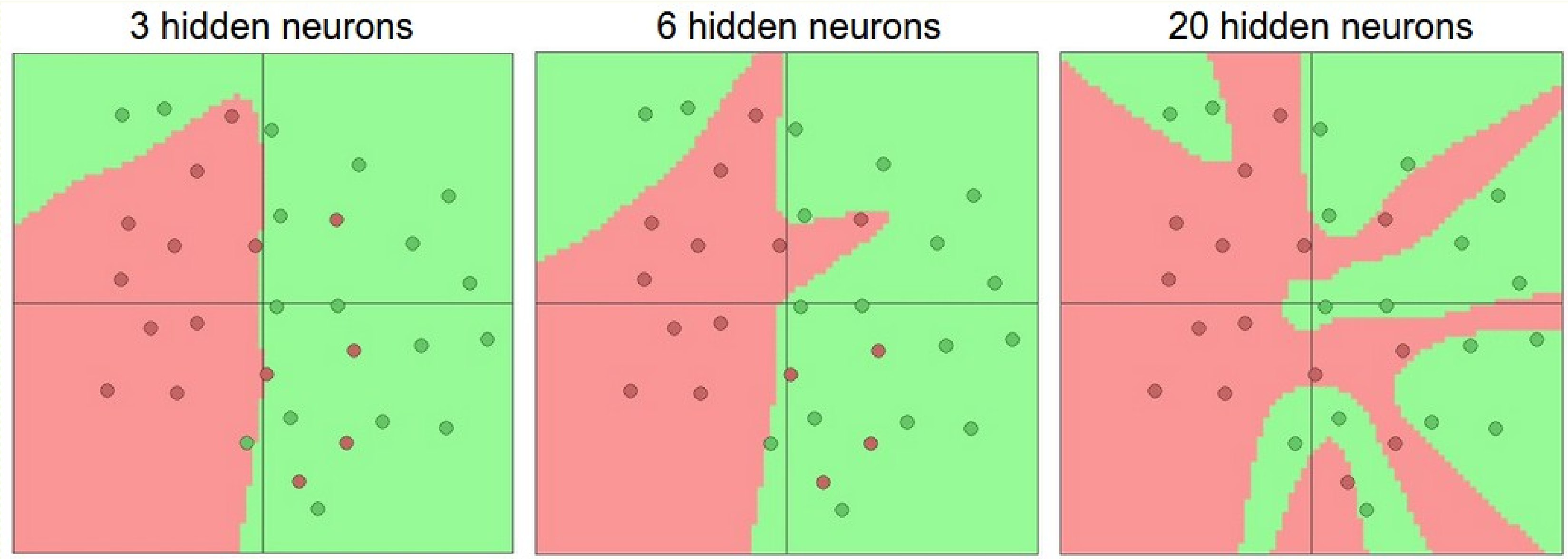
$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{o}} = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^{o}} \mathbf{X}^{T}$$

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**

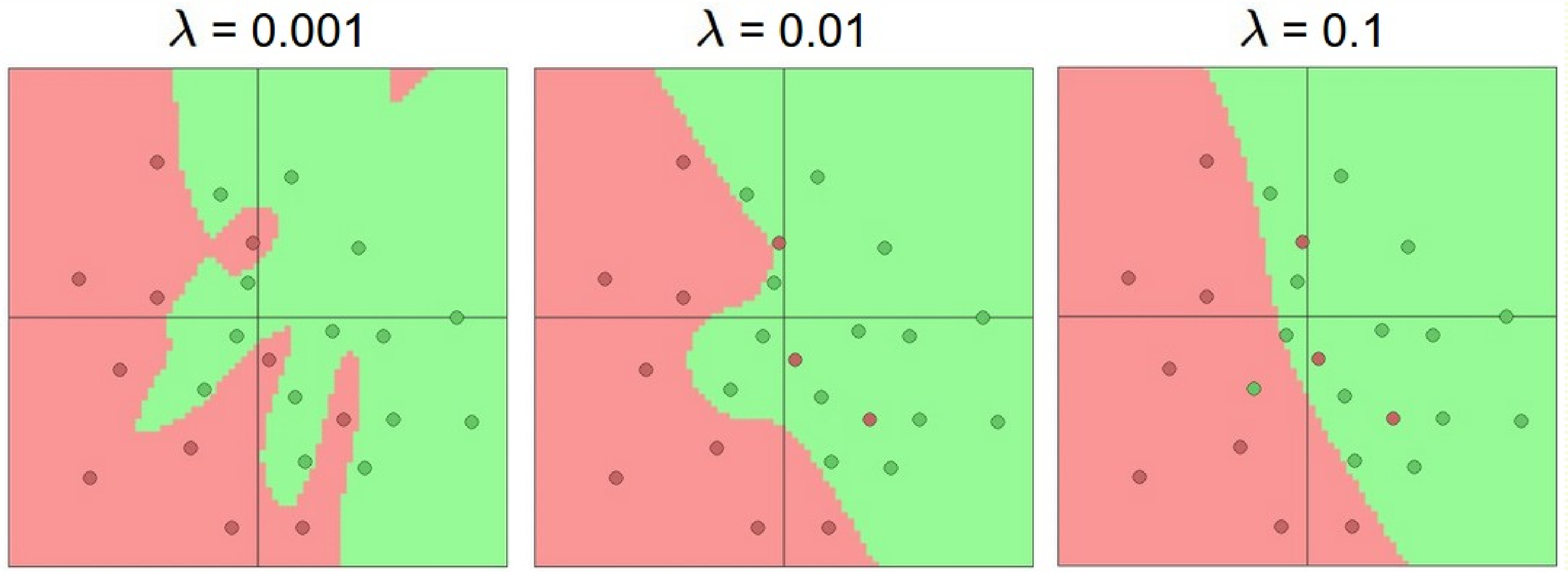
Regularization

- Larger networks are more prone to overfitting



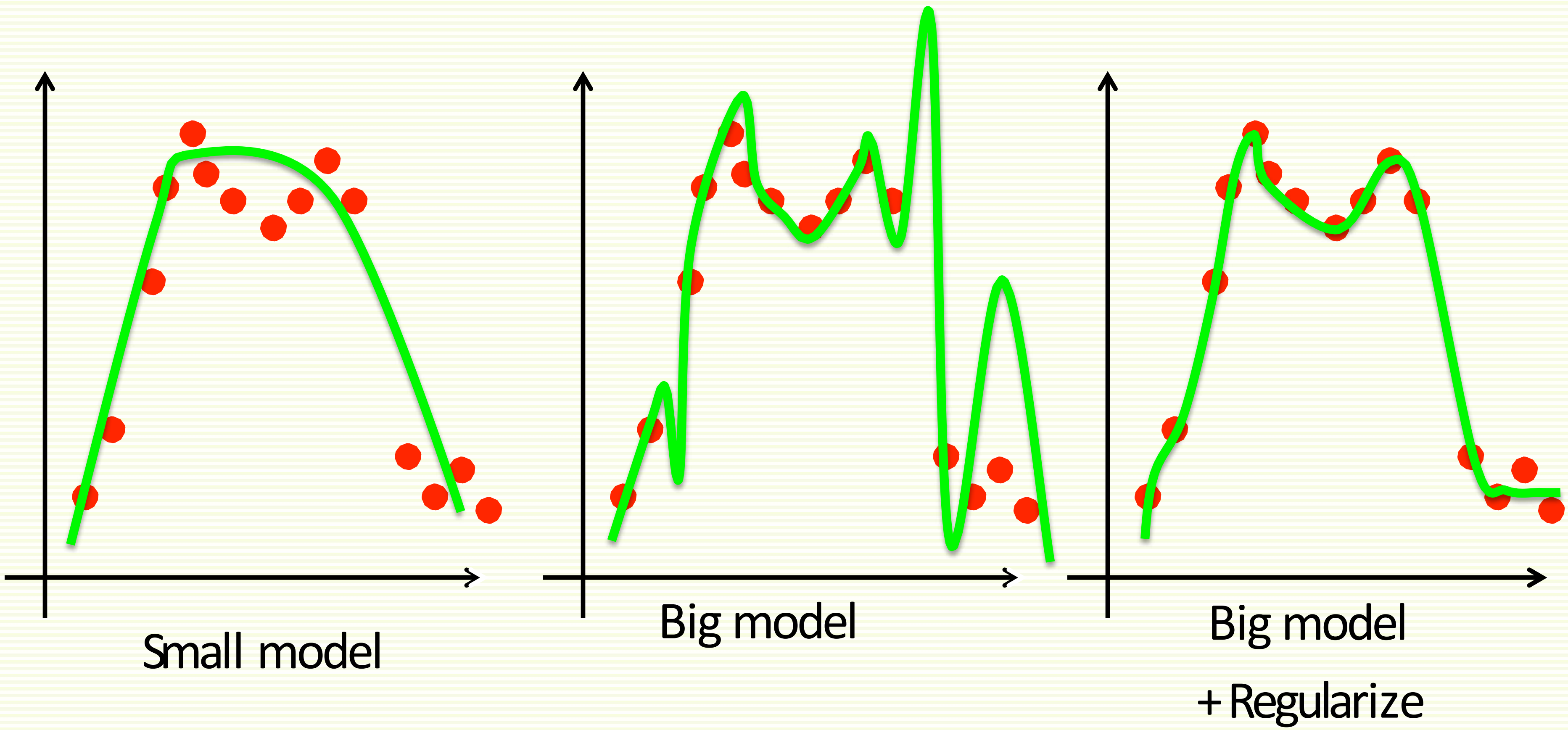
Regularization

- 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 $\lambda\mathbf{w}$ from each weight $\mathbf{w}$
  - intuitively, implements weight decay

Small model vs. Big Model+Regularize

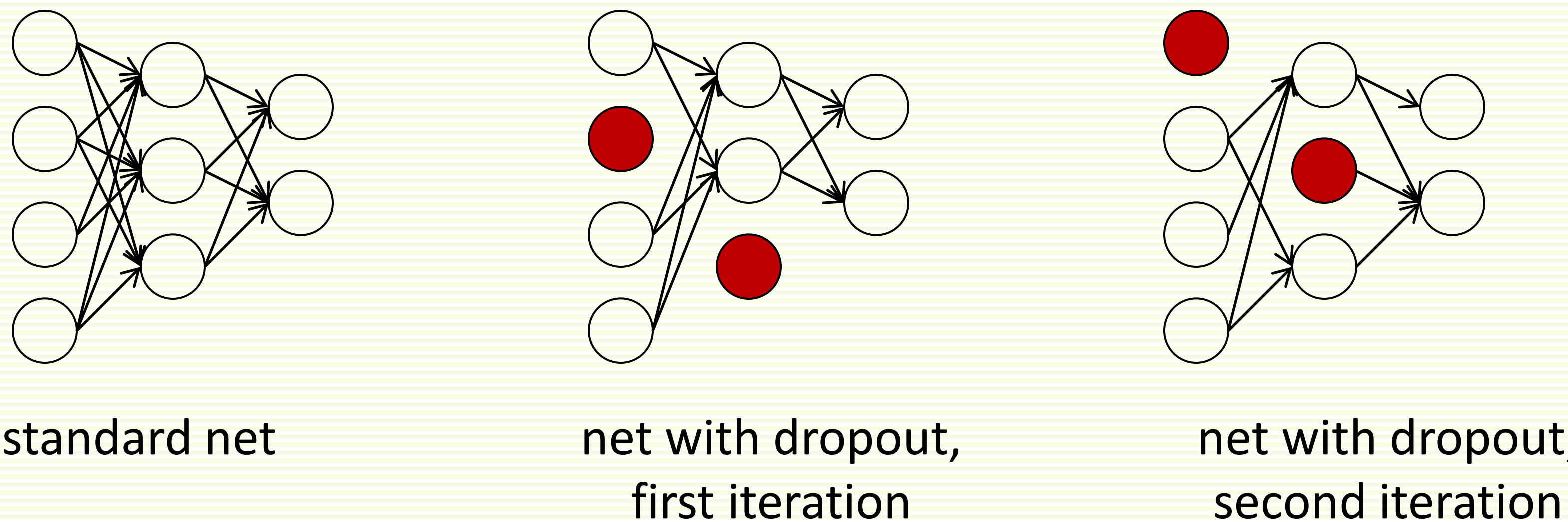


Ensembles of Neural Networks

- 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

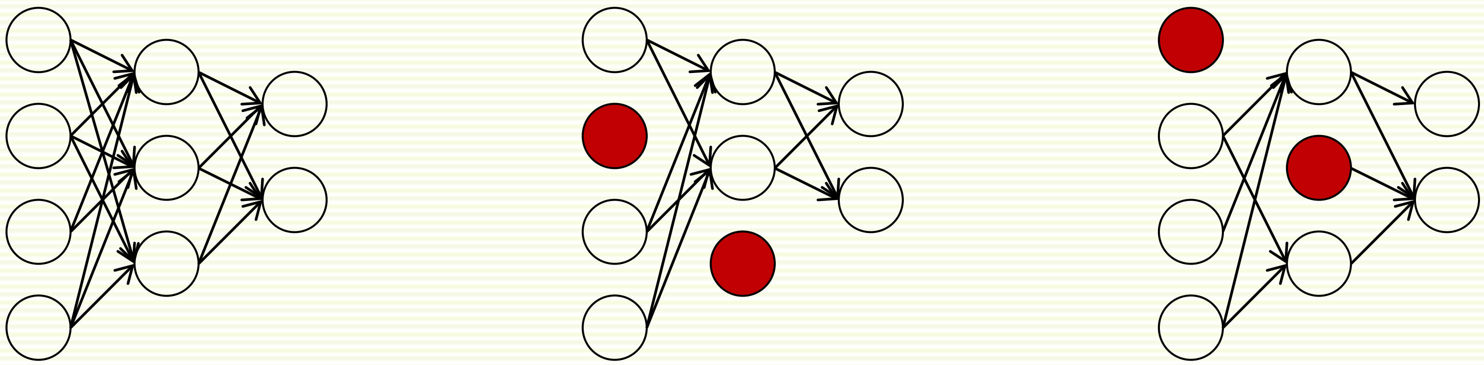
Dropout

- During training, keep each unit active with probability **p**
  - otherwise set to 0
  - **p** = 0.5 is common



- During training, sampling a subset of 2^n networks possible
- Extreme ensemble training
  - training each member of ensemble only on a small batch of examples

Dropout



standard net
used at test time

net with dropout,
first iteration

net with dropout,
second iteration

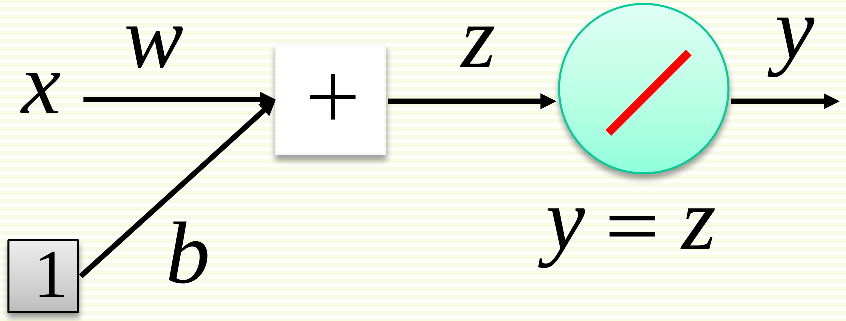
- At test time, no dropout is applied, the whole “ensemble” is active
- Scale units by **p** at test time, since all units are active now
- Or, better, scale units by 1/**p** at training time
- Dropout is usually applied to fully connected layers

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

- Set the learning rate carefully
- Toy example



- Optimal weights: w = 1, b = 0
- Gradient descent

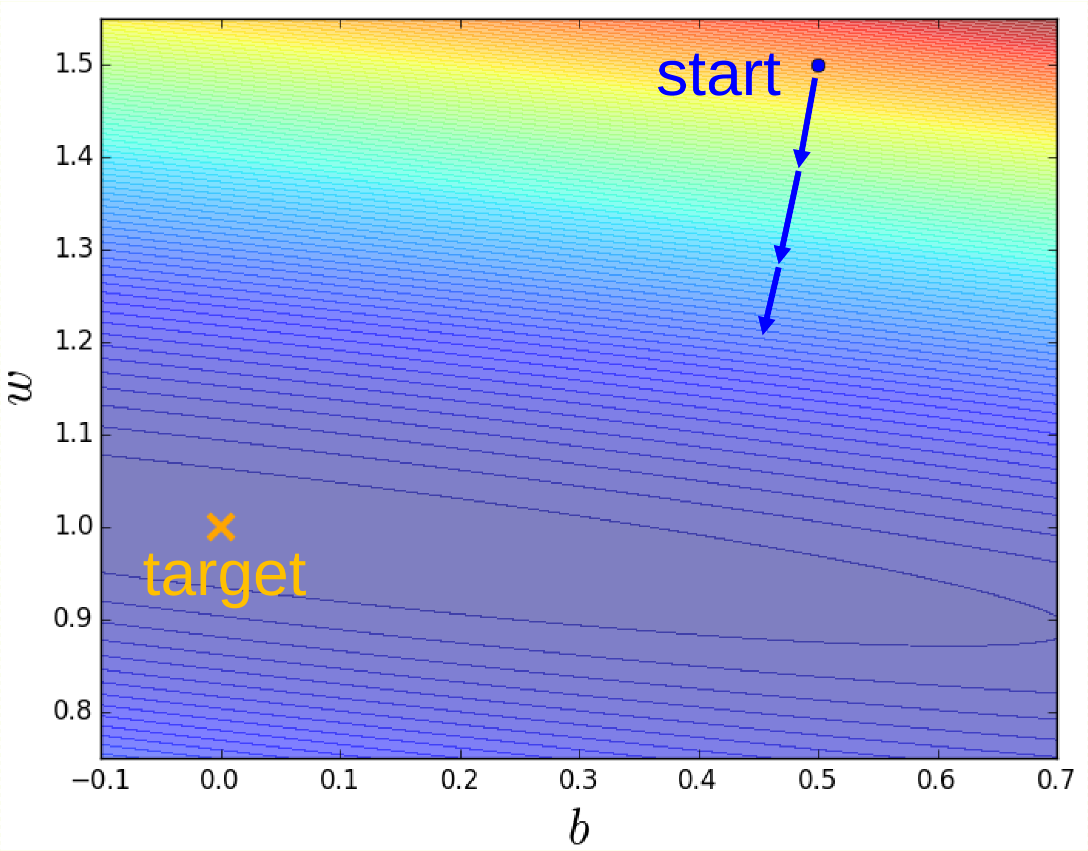
$$\mathbf{w}^{\mathbf{t}} = \mathbf{w}^{\mathbf{t}-1} - \alpha \nabla \mathbf{L}\left(\mathbf{w}^{\mathbf{t}-1}\right)$$

- Training Data (20 examples)

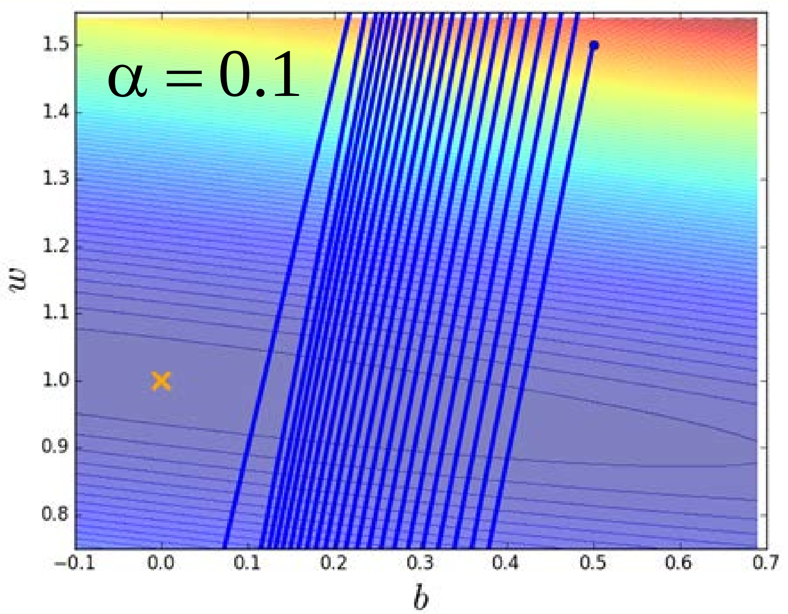
x = [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
y = [0.1, 0.4, 0.9, 1.6, 2.2, 2.5, 2.8, 3.5, 3.9, 4.7, 5.1, 5.3, 6.3, 6.5, 6.7, 7.5, 8.1, 8.5, 8.9, 9.5]

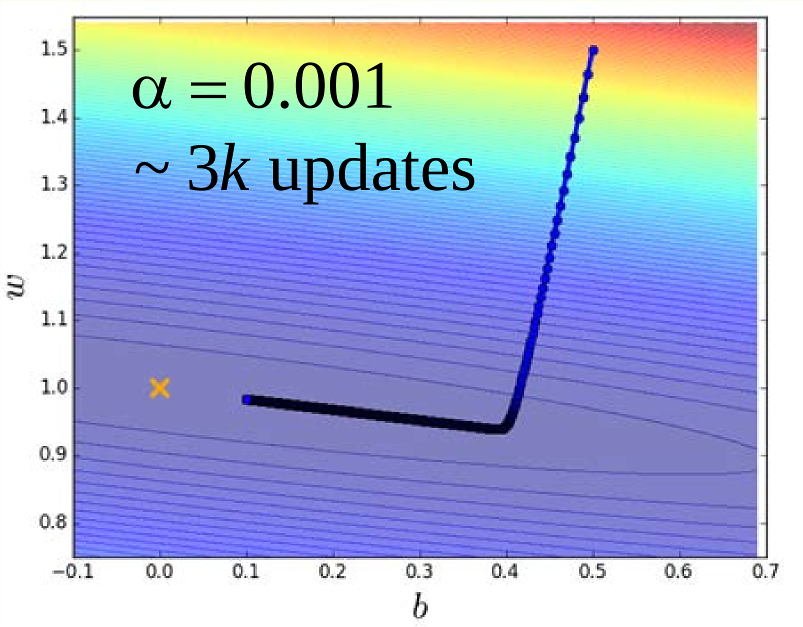
Practical Training Tips: Learning Rate

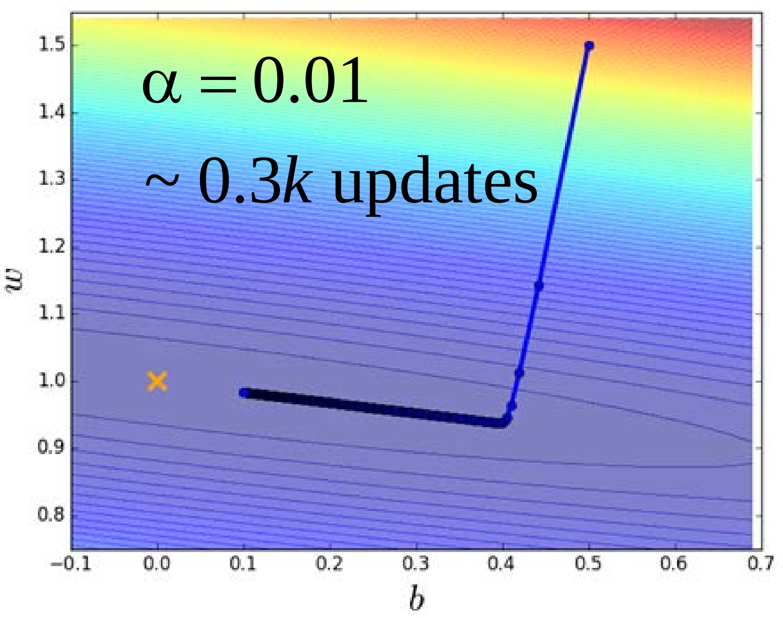
- Surface of the loss function **L(w,b)**



Practical Training Tips: Learning Rate





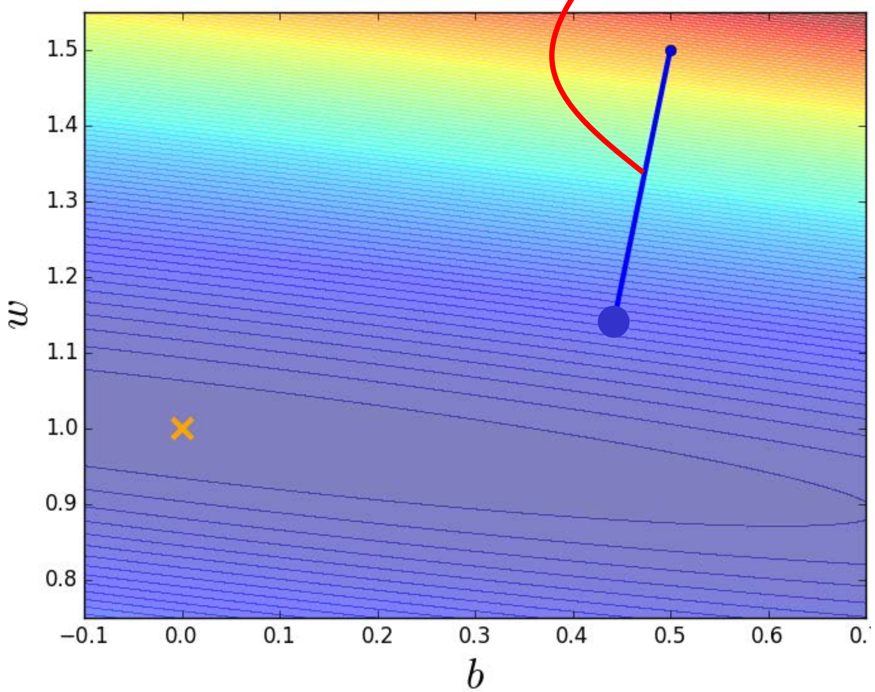


Practical Training Tips: Learning Rate

- 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

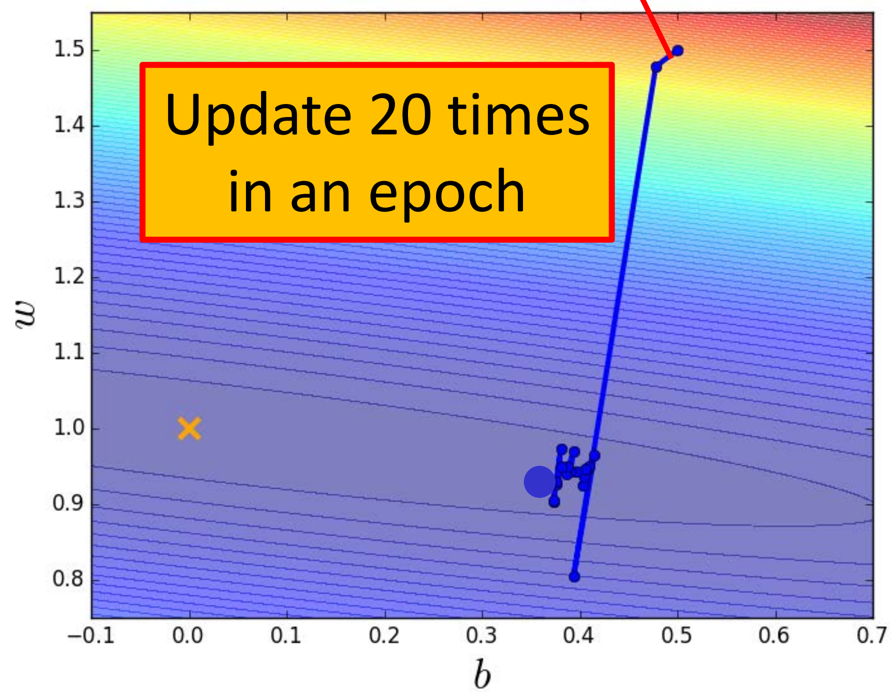
Practical Training Tips: Batch Size

see all examples



Gradient descent

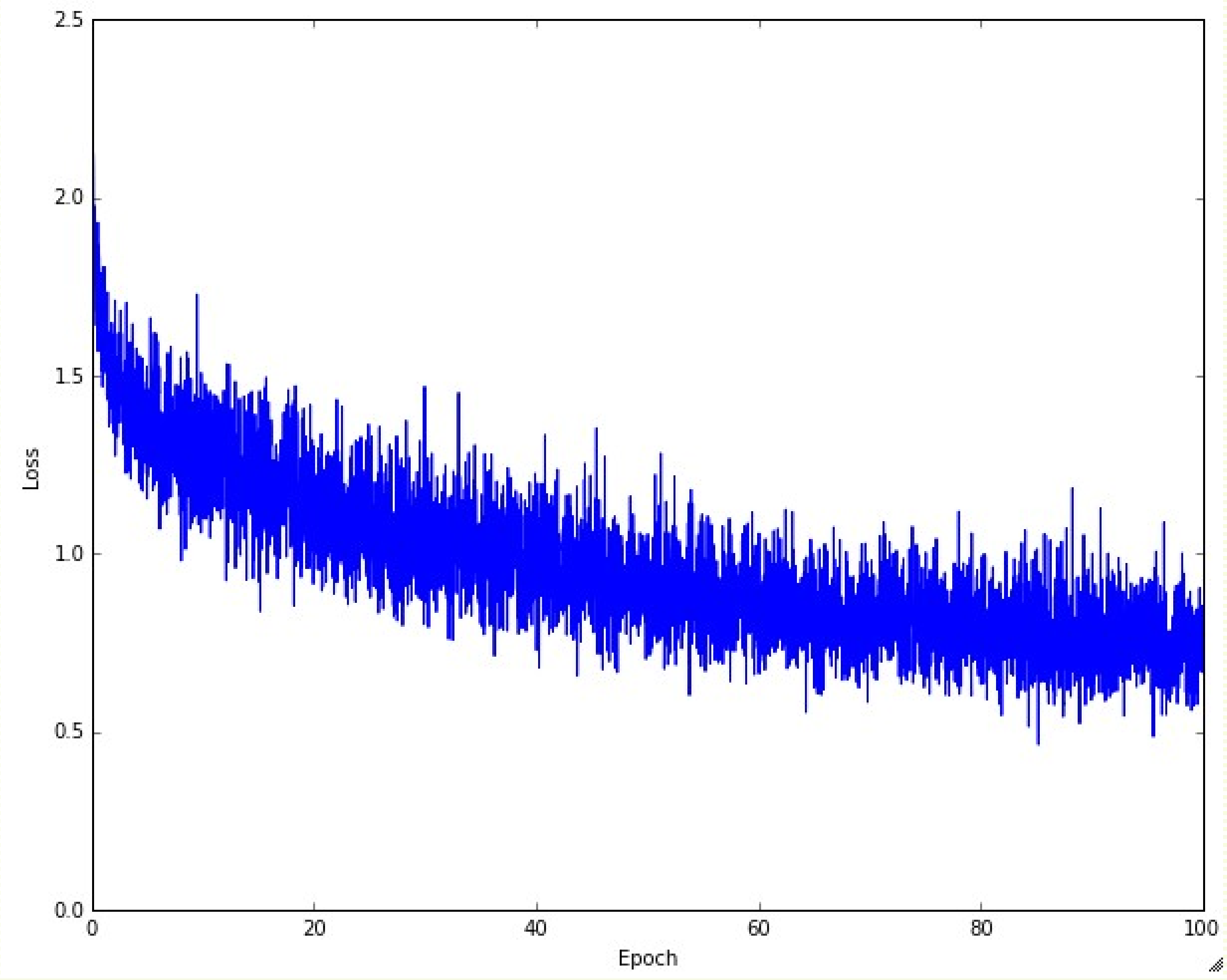
see only one example



Stochastic gradient descent,
1 epoch

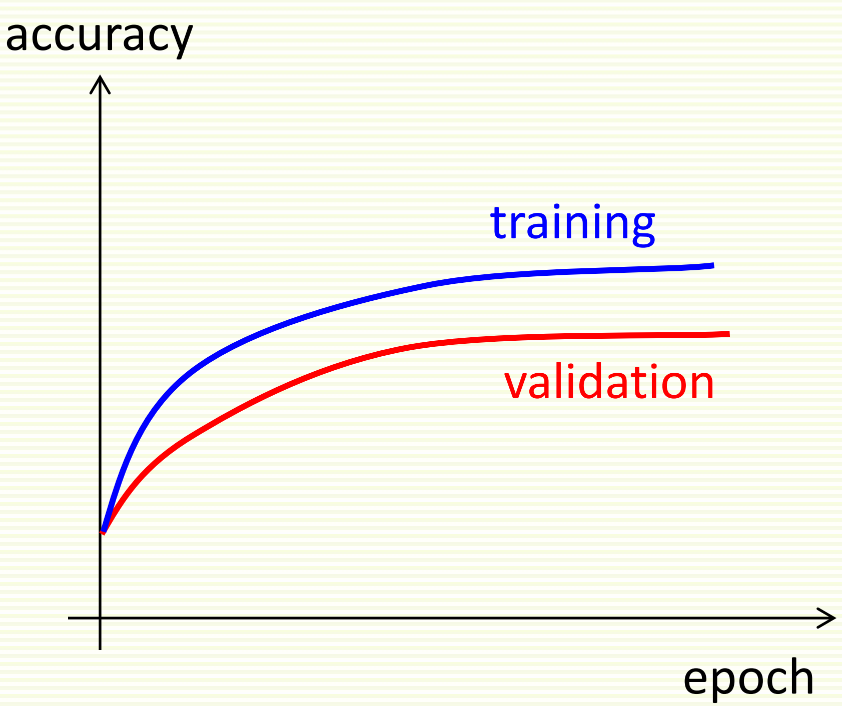
Practical Training Tips: Batch Size

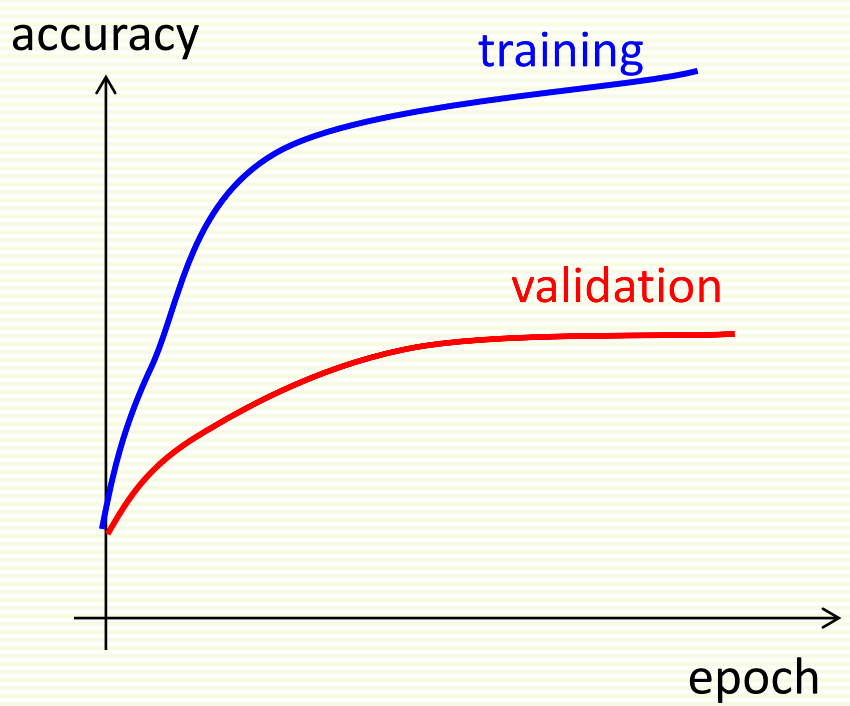
- Track number of epoch vs. Loss
- If the line is too wiggly, batch size might be too small



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

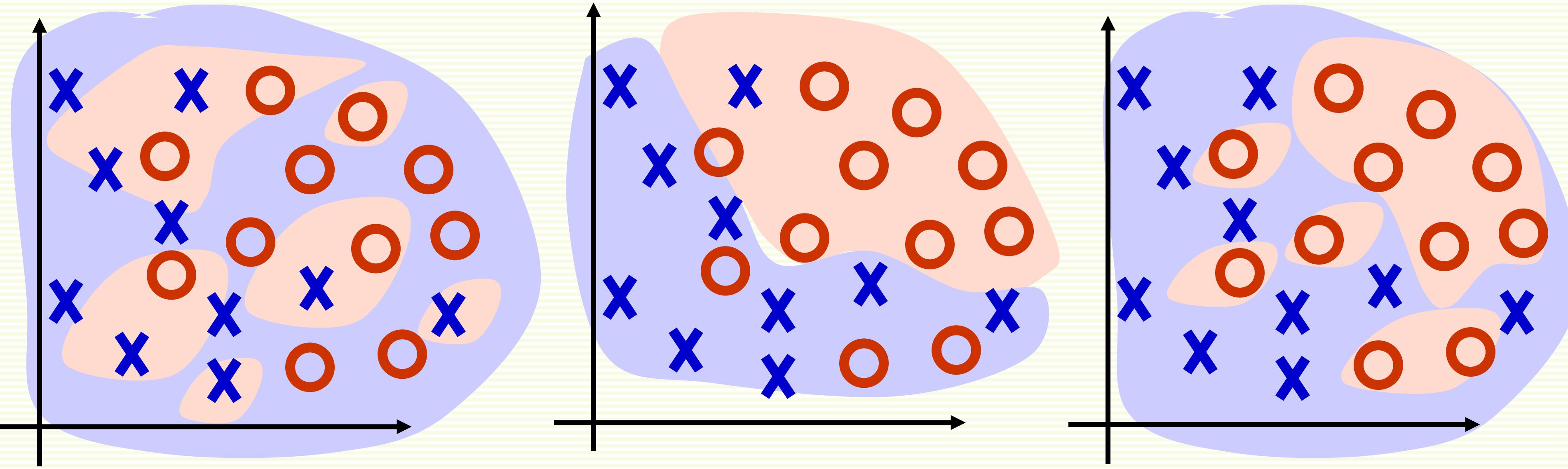
- Add temporal average direction in which weights have been moving recently
- Parameter vector will build up velocity in direction that has consistent gradient
- Helps avoid local minima and speed up descent in flat (plateau) regions
- Previous direction: $\Delta\mathbf{w}^t = \mathbf{w}^t - \mathbf{w}^{t-1}$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

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

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 cross 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 epoch
  - random search might be better than grid search