Lecture 4: Artificial Neural Networks: Multilayer Networks and BP

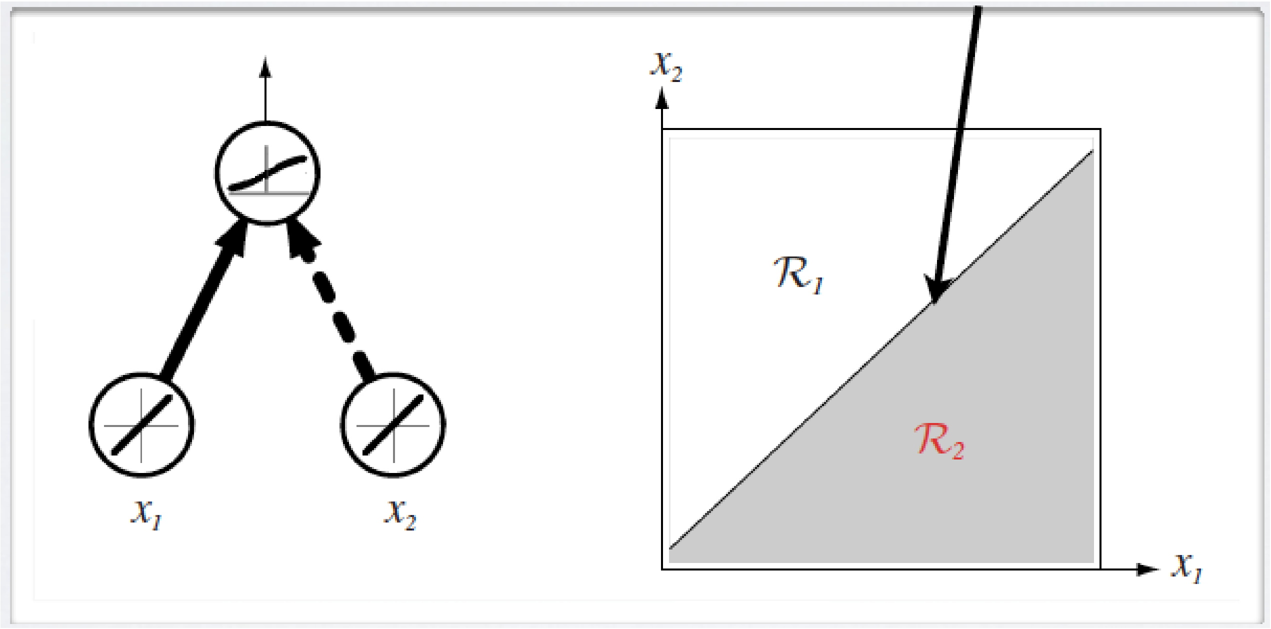
Xuming He
SIST, ShanghaiTech
Fall, 2019

Outline

- Multi-layer neural networks
  - Limitations of single layer networks
  - Networks with single hidden layer
  - Sequential network architecture
- Inference and learning
  - Forward and Backpropagation
  - Examples: one-layer network
  - General BP algorithm

Capacity of single neuron

- Binary classification
  - A neuron estimates $P(y=1|\mathbf{x}) = \sigma(\mathbf{w}^\intercal \mathbf{x})$
  - Its decision boundary is linear, determined by its weights



Capacity of single neuron

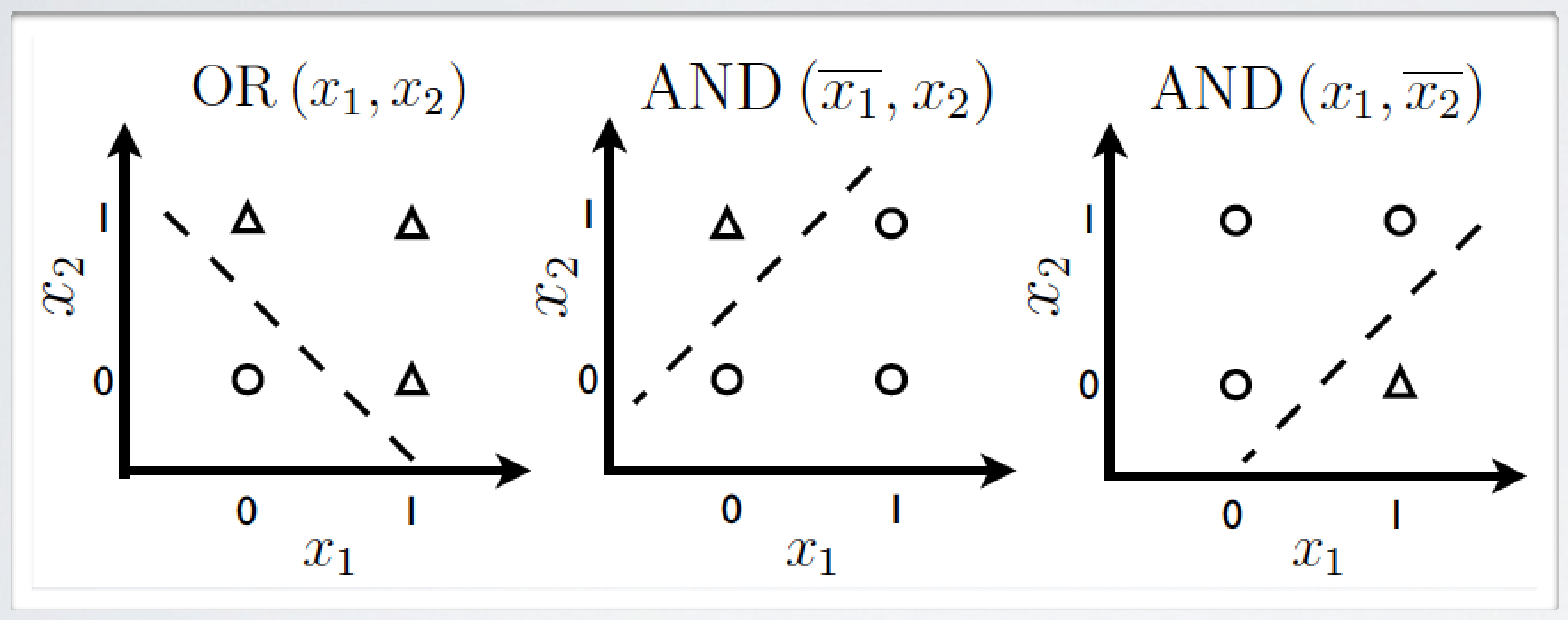
- Can solve linearly separable problems

$$\mathcal{D} = \mathcal{D}^{+} \cup \mathcal{D}^{-}$$

$$\exists\ \mathbf{w}^{*}, \mathbf{w}^{*\mathsf{T}}\mathbf{x} > 0,\ \forall \mathbf{x} \in \mathcal{D}^{+}$$

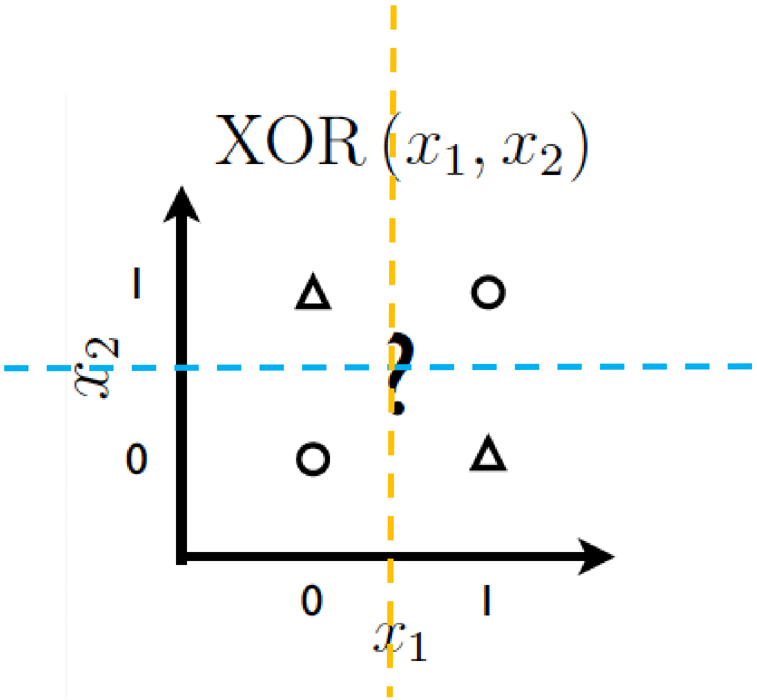
$$\mathbf{w}^{*\mathsf{T}}\mathbf{x} < 0,\ \forall \mathbf{x} \in \mathcal{D}^{-}$$

  - Examples



Capacity of single neuron

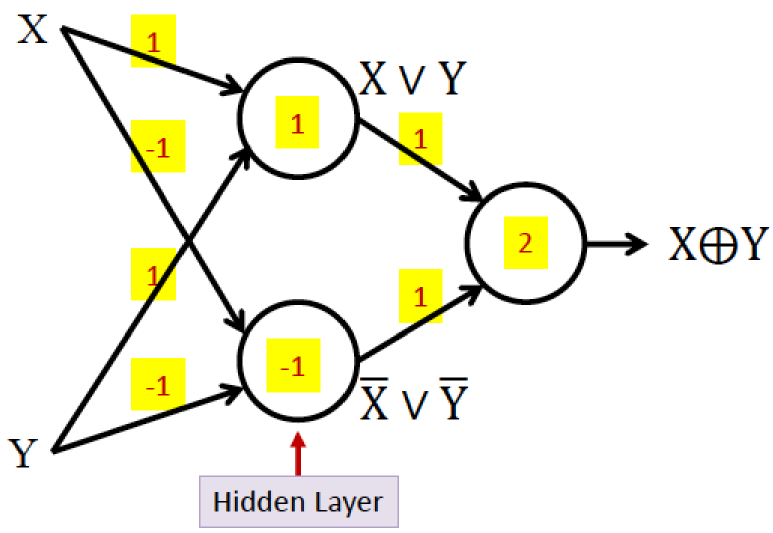
- Can't solve non linearly separable problems



- Can we use multiple neurons to achieve this?

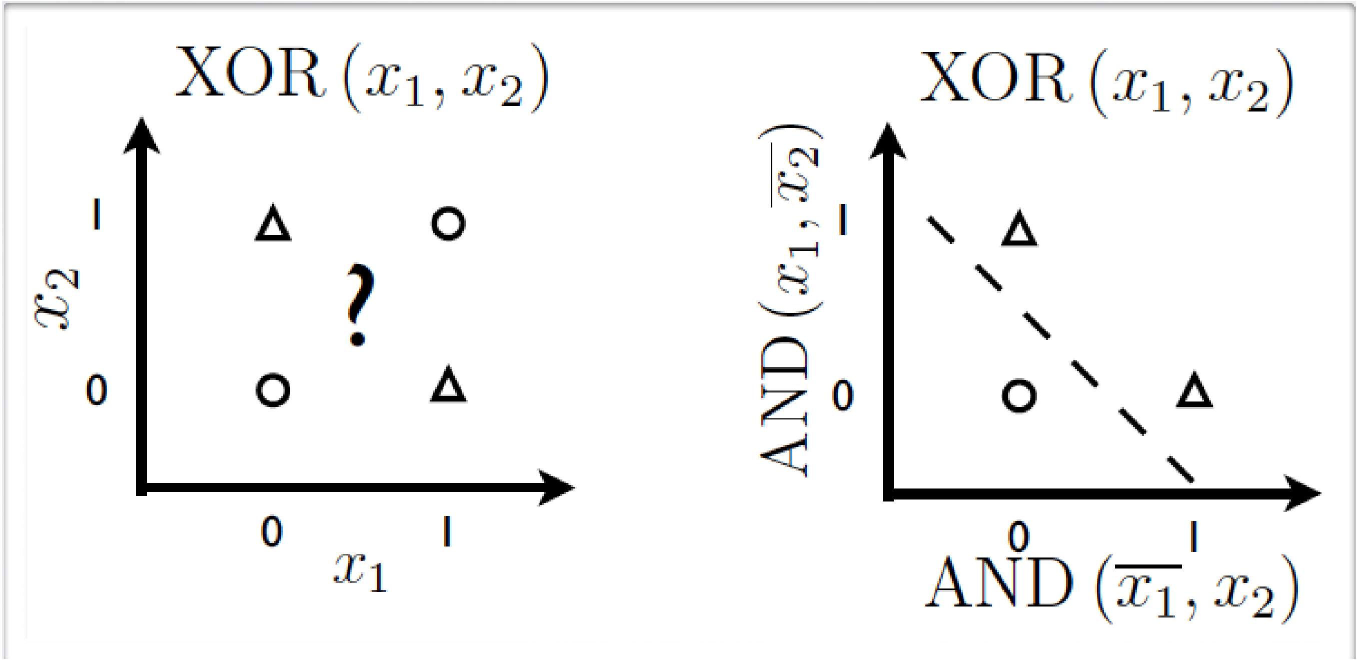
Capacity of single neuron

- Can't solve non linearly separable problems
- Unless the input is transformed in a better representation



Capacity of single neuron

- Can't solve non linearly separable problems



- Unless the input is transformed in a better representation

Adding one more layer

- Single hidden layer neural network
  - 2-layer neural network: ignoring input units

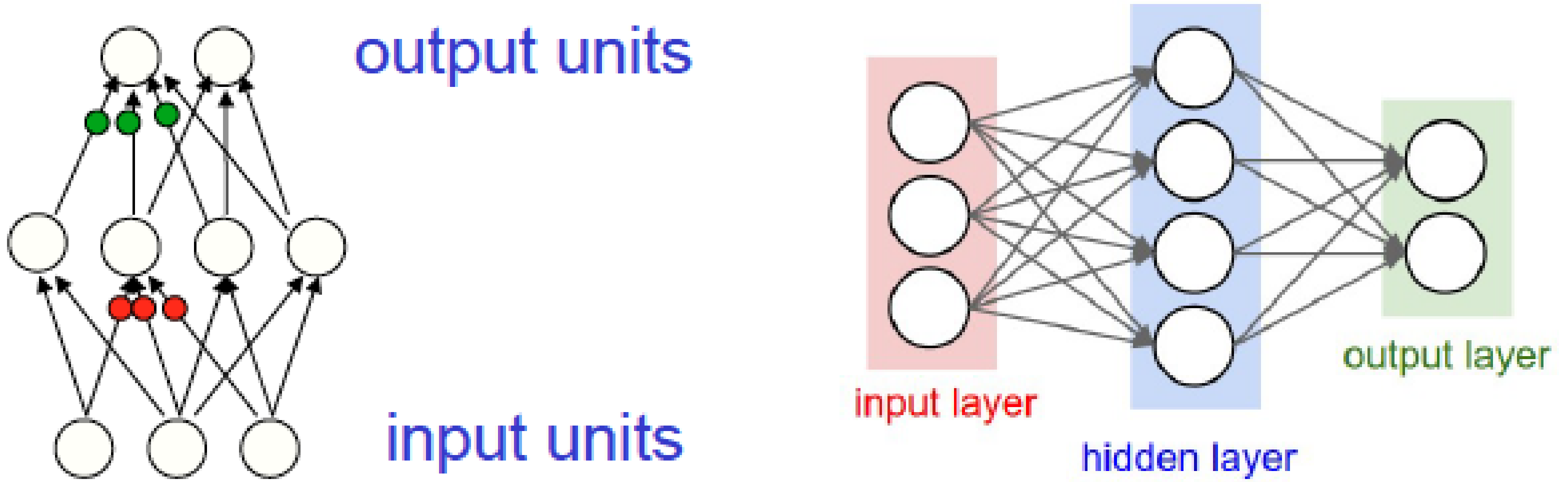
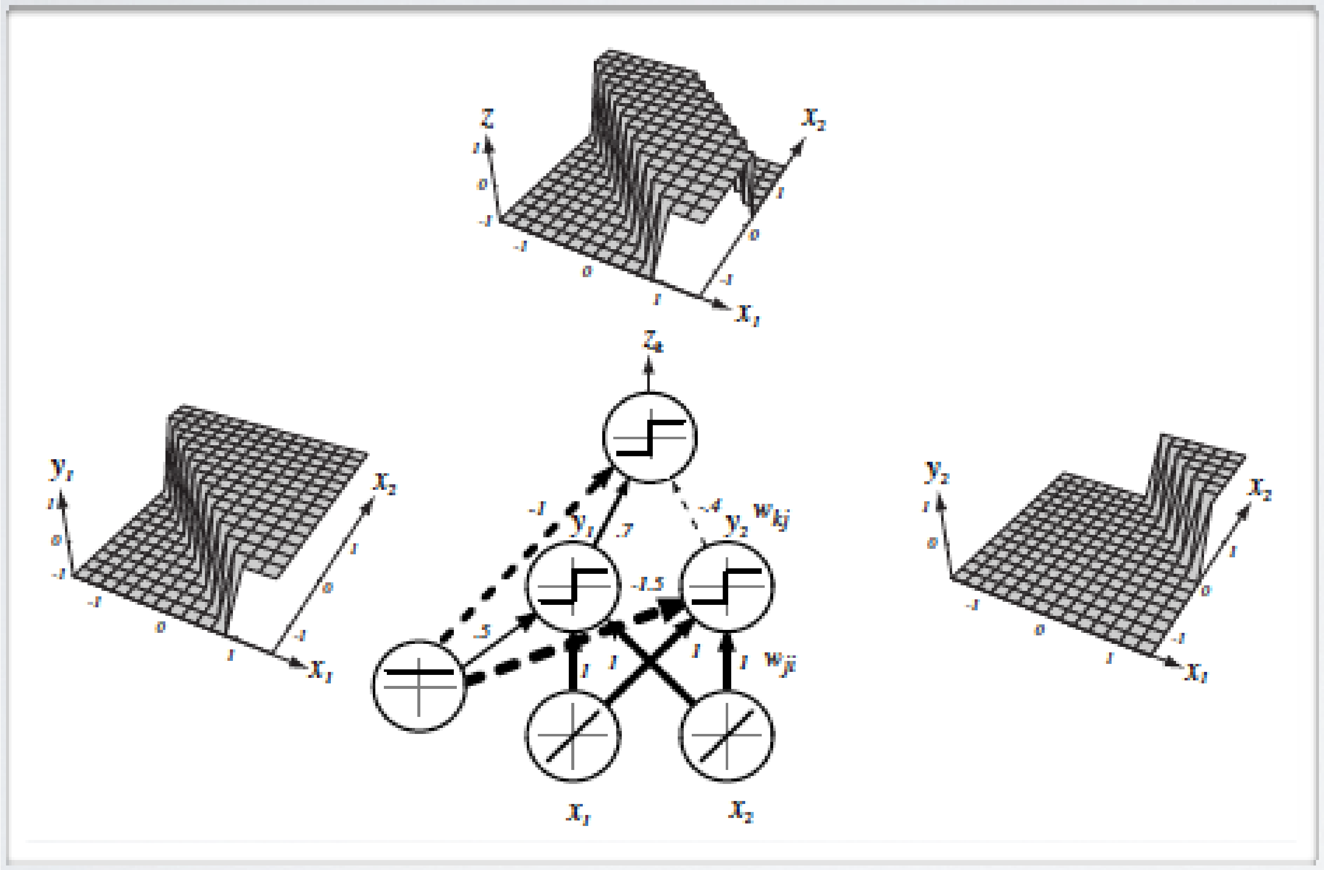


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Q: What if using linear activation in hidden layer?

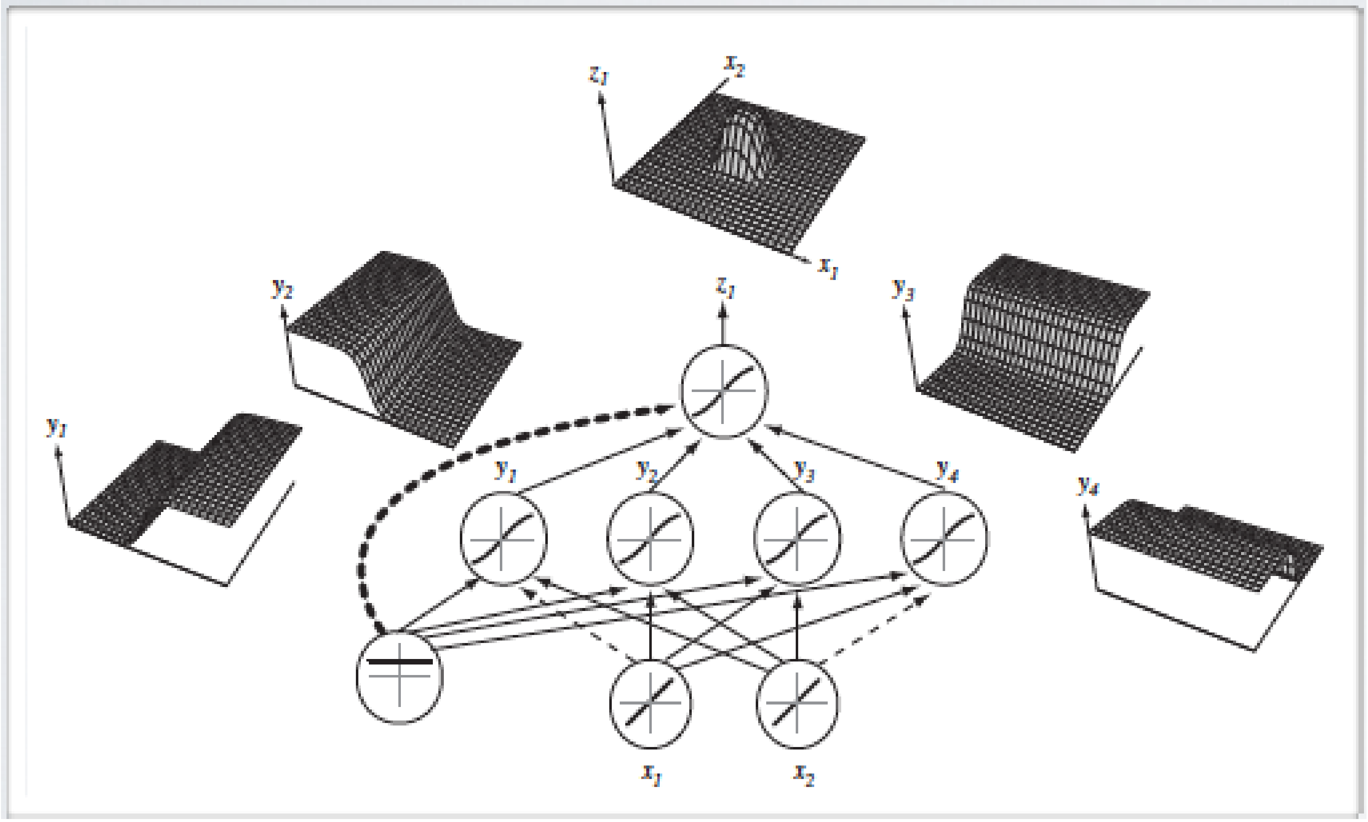
Capacity of neural network

- Single hidden layer neural network
  - Partition the input space into regions



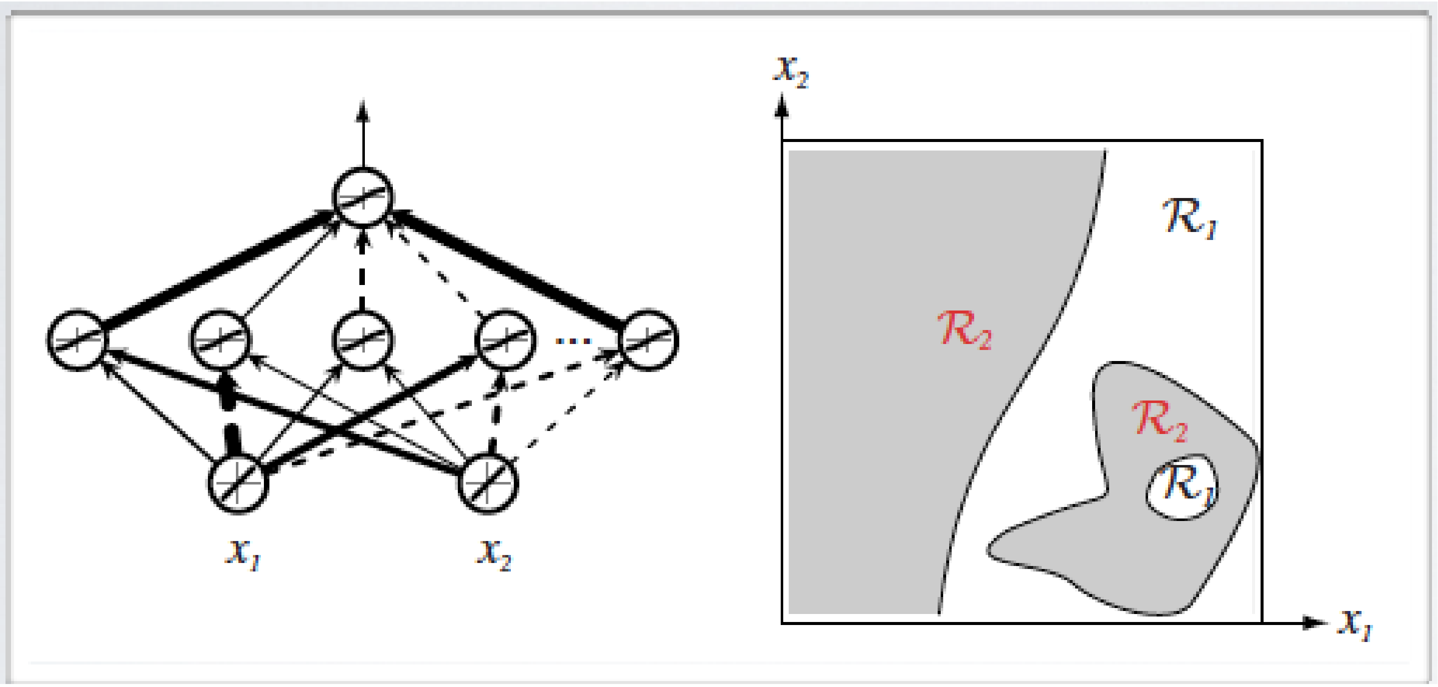
Capacity of neural network

- Single hidden layer neural network
  - Form a stump/delta function



Capacity of neural network

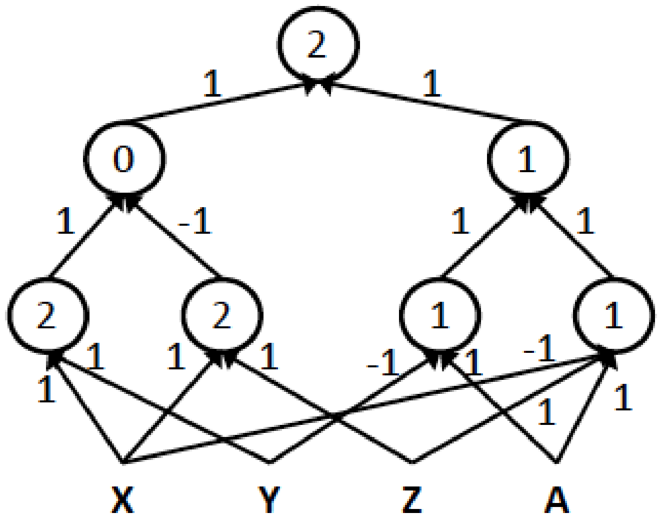
- Single hidden layer neural network



Multi-layer perceptron

- Boolean case
  - Multilayer perceptrons (MLPs) can compute more complex Boolean functions
  - MLPs can compute ***any*** Boolean function
    - Since they can emulate individual gates
  - MLPs are *universal Boolean functions*

$$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$$



Capacity of neural network

- Universal approximation
  - Theorem (Hornik, 1991)

    A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.
  - The result applies for sigmoid, tanh and many other hidden layer activation functions

- Caveat: good result but not useful in practice
  - How many hidden units?
  - How to find the parameters by a learning algorithm?

General neural network

- Multi-layer neural network

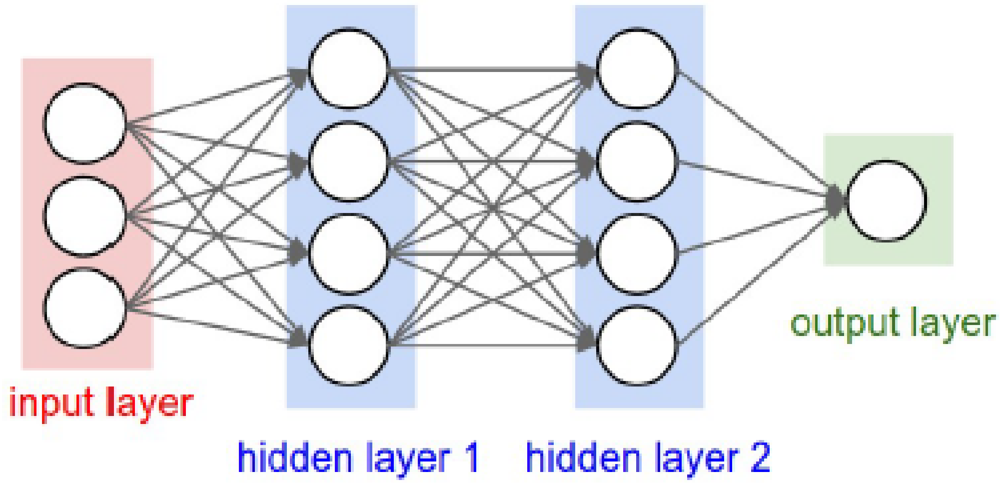
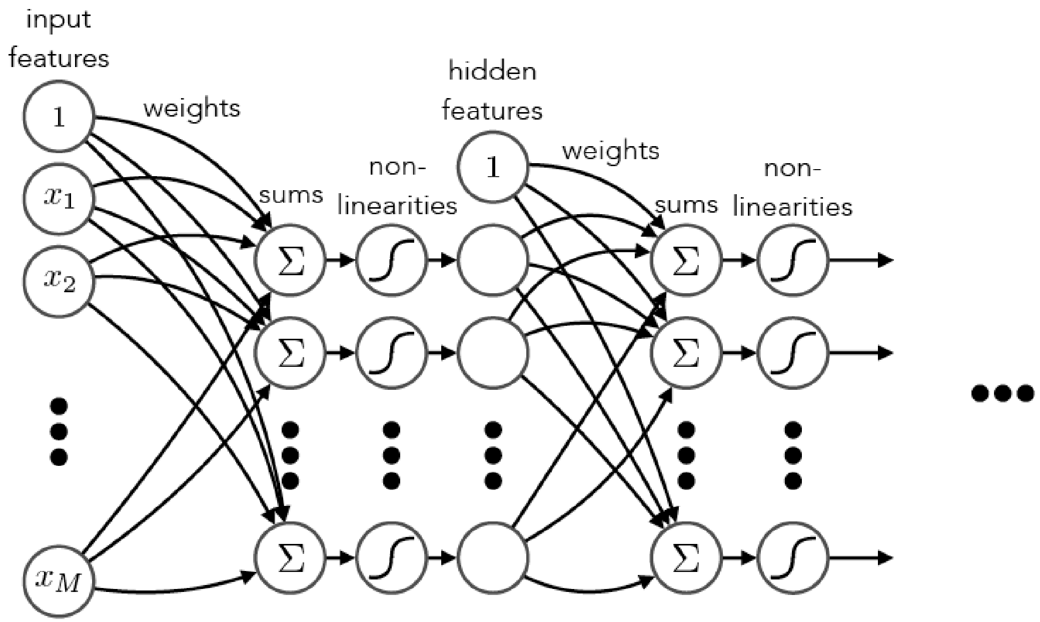


Figure : A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; a N-layer neural network:
  - $N - 1$ layers of hidden units
  - One output layer

Multilayer networks



network: *sequence of parallelized weighted sums and non-linearities*

DEFINE $\mathbf{x}^{(0)} \equiv \mathbf{x}$, $\mathbf{x}^{(1)} \equiv \mathbf{h}$, ETC.

1st layer

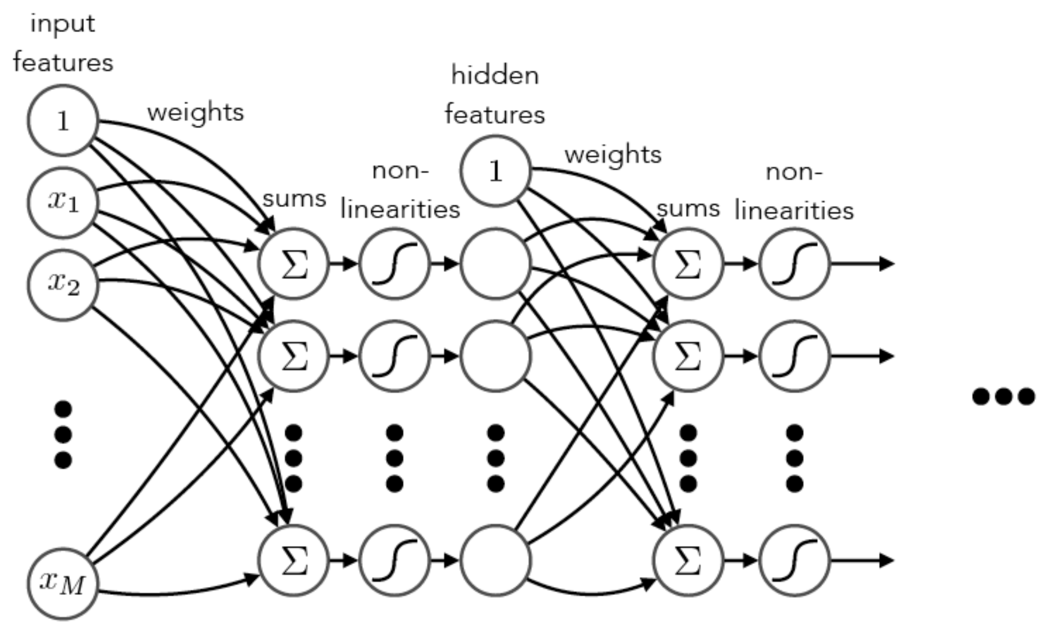
$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

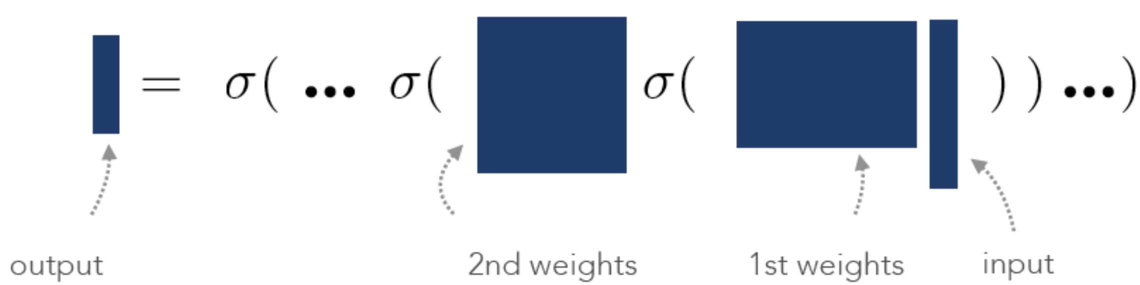
$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

…

Multilayer networks

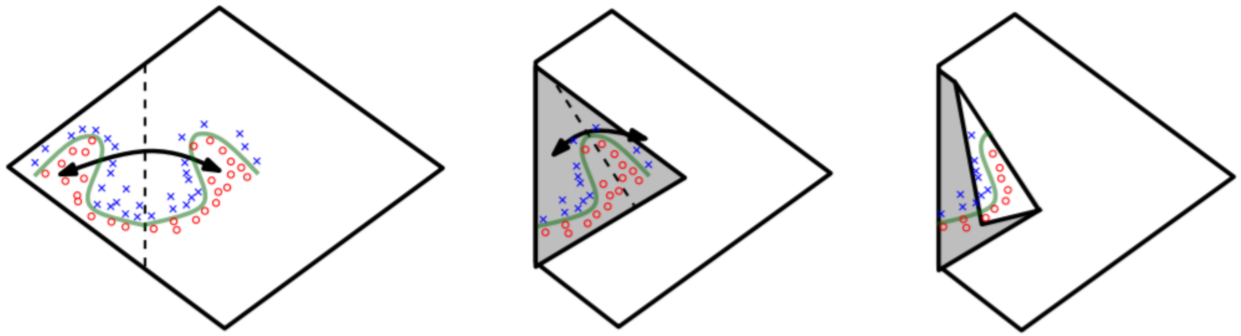


network: *sequence of parallelized weighted sums and non-linearities*



Why more layers (deeper)?

- A deep architecture can represent certain functions more compactly
  - (Montufar et al., NIPS'14)
    - Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow one.

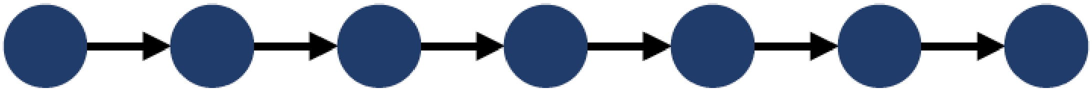


Why more layers (deeper)?

- A deep architecture can represent certain functions more compactly
  - Example: Boolean functions
    - There are Boolean functions which require an exponential number of hidden units in the single layer case
    - require a polynomial number of hidden units if we can adapt the number of layers
  - Example: multivariate polynomials (Rolnick & Tegmark, ICLR'18)
    - Total number of neurons m required to approximate natural classes of multivariate polynomials of n variables
    - grows only linearly with n for deep neural networks, but grows exponentially when merely a single hidden layer is allowed.

Other network connectivity

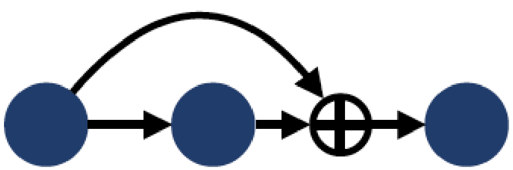
sequential connectivity: *information must flow through the entire sequence to reach the output*



information may not be able to propagate easily

→ *make shorter paths to output*

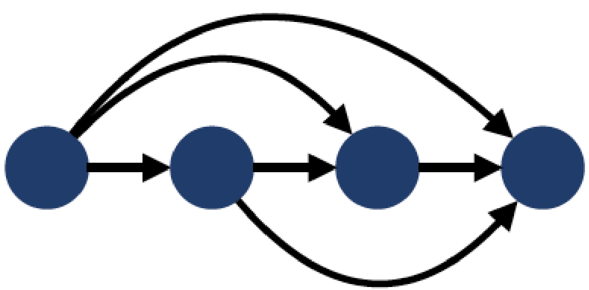
residual & highway connections



Deep residual learning for image recognition, He et al., 2016

Highway networks, Srivastava et al., 2015

dense (concatenated) connections



Densely connected convolutional networks, Huang et al., 2017

Outline

- Multi-layer neural networks
  - Limitations of single layer networks
  - Neural networks with single hidden layer
  - Sequential network architecture and variants
- Inference and learning
  - Forward and Backpropagation
  - Examples: one-layer network
  - General BP algorithm

Acknowledgement: Rich Zemel@UofT & Feifei Li's cs231n notes

Computation in neural network

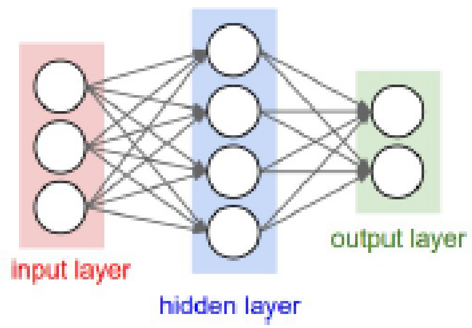
- We only need to know two algorithms
  - Inference/prediction: simply forward pass
  - Parameter learning: needs backward pass
- Basic fact:
  - A neural network is a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

  - All the f functions are linear + (simple) nonlinear (differentiable a.e.) operators

Inference example: Forward Pass

- What does the network compute?



  - Output of the network can be written as:

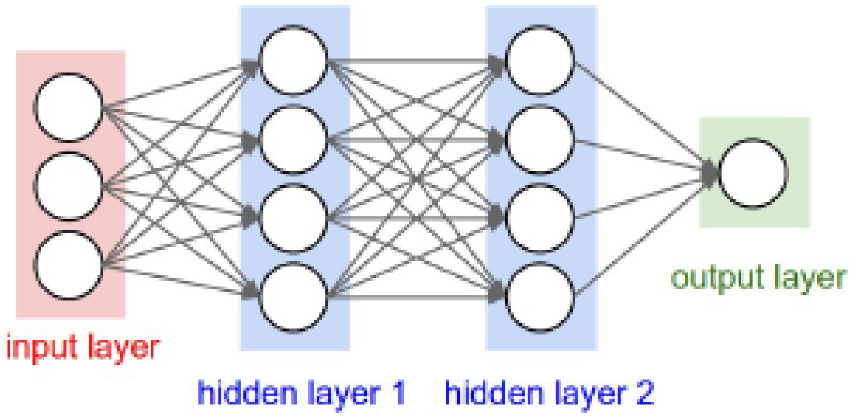
$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(\mathbf{x}) = g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})$$

(j indexing hidden units, k indexing the output units, D number of inputs)

Forward Pass in Python

- Example code for a forward pass for a 3-layer network in Python:




```python
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- Can be implemented efficiently using matrix operations

Parameter learning: Backward Pass

- Supervised learning framework
  - Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \mathrm{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

    where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

  - Define a loss function, eg:
    - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
    - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$
  - Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

    where η is the learning rate (and E is error/loss)

Backward pass

- Backpropagation
  - An efficient method for computing gradients in NNs
  - A neural network as a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \ldots f_1(\mathbf{w}_1, \mathbf{x}) \ldots))$$

and the loss $\mathcal{L}$ is a function of the network output

$\longrightarrow$ use <u>chain rule</u> to calculate gradients

Review: Chain rule

- Formal definition

For any nested function $y = f(g(x))$

$$\boxed{\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}}$$

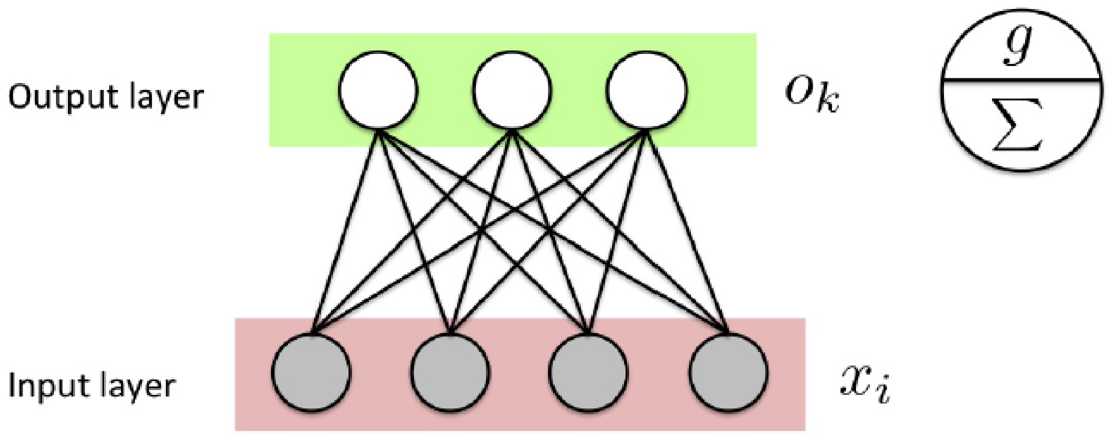
Check - we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$$

$$y = f(z) \Rightarrow \Delta y = \frac{dy}{dz} \Delta z = \frac{dy}{dz} \frac{dg(x)}{dx} \Delta x$$

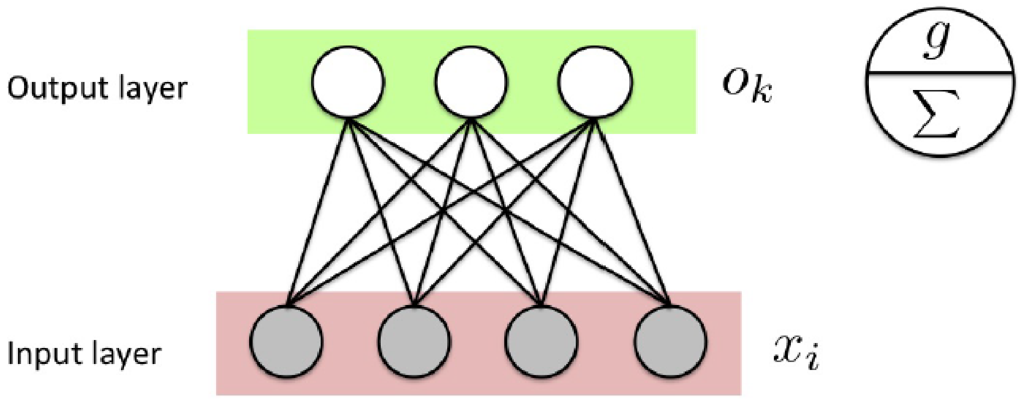
Example: Single Layer Network

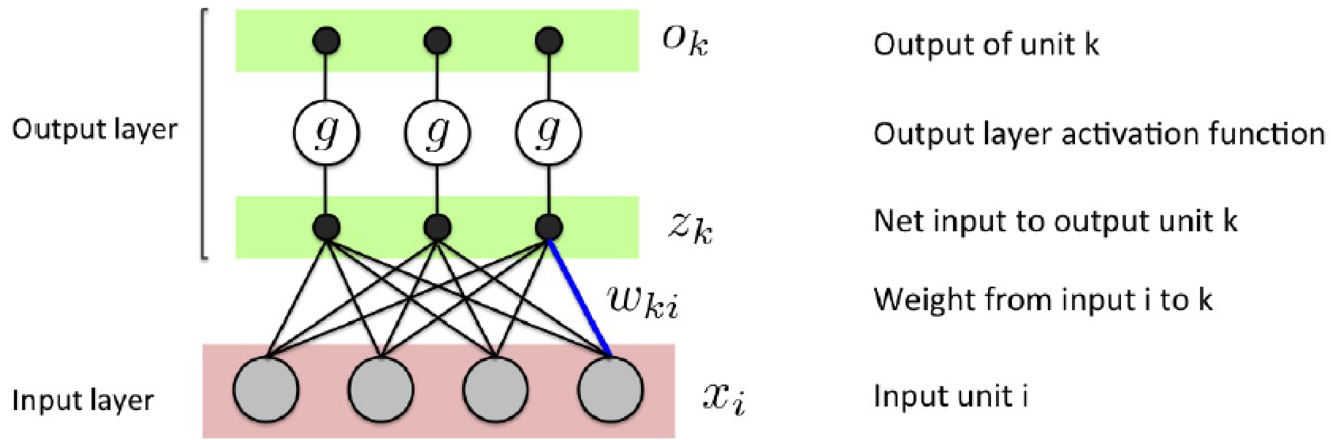
- Let's take a single layer network



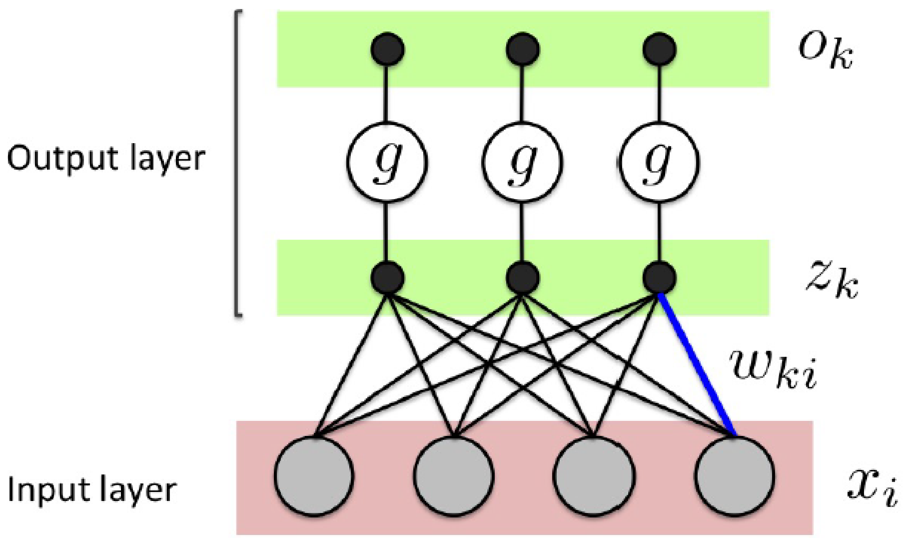
Example: Single Layer Network

- Let's take a single layer network and draw it a bit differently





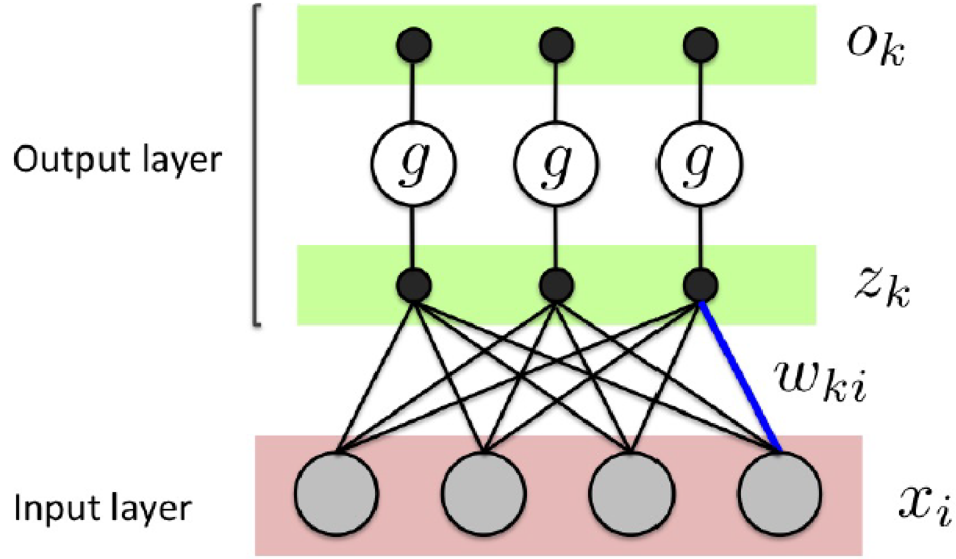
Example: Single Layer Network



- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} =$$

Example: Single Layer Network

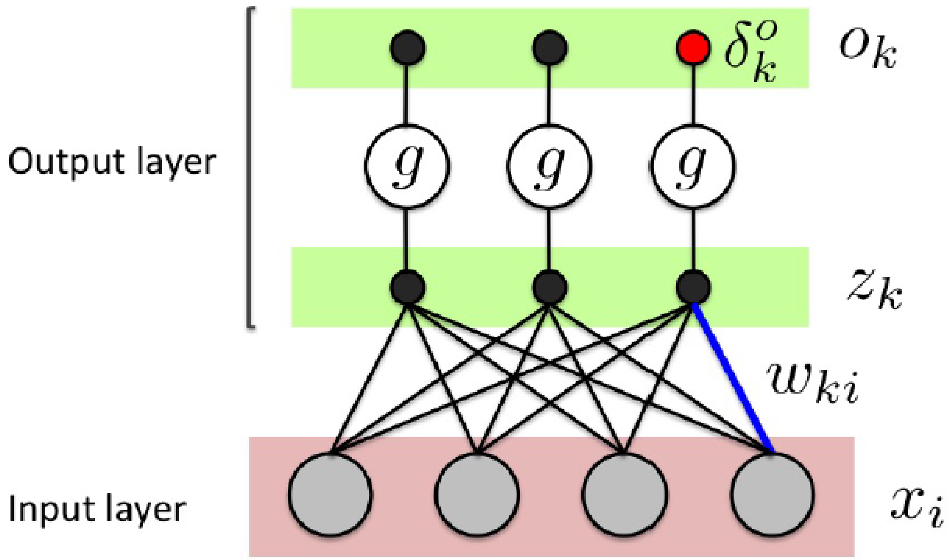


- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

- Error gradient is computable for any continuous activation function $g()$, and any continuous error function

Example: Single Layer Network

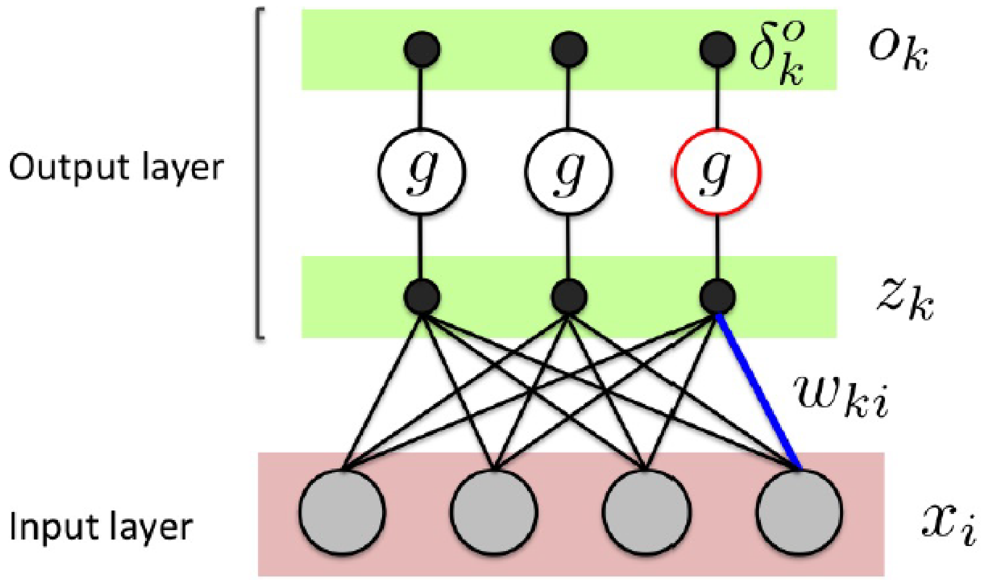


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \underbrace{\frac{\partial E}{\partial o_k}}_{\delta_k^o} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

Example: Single Layer Network

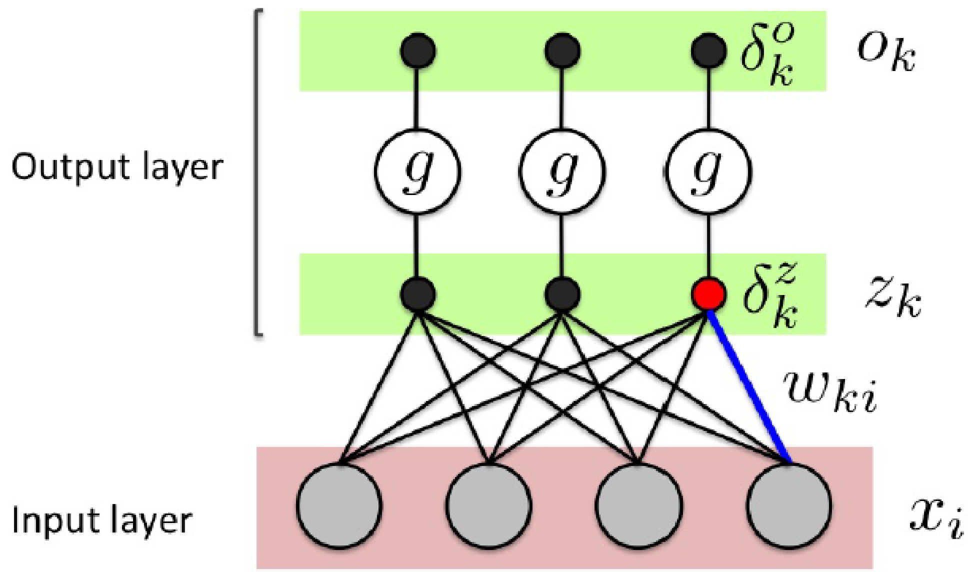


$$\frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^o \frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Example: Single Layer Network

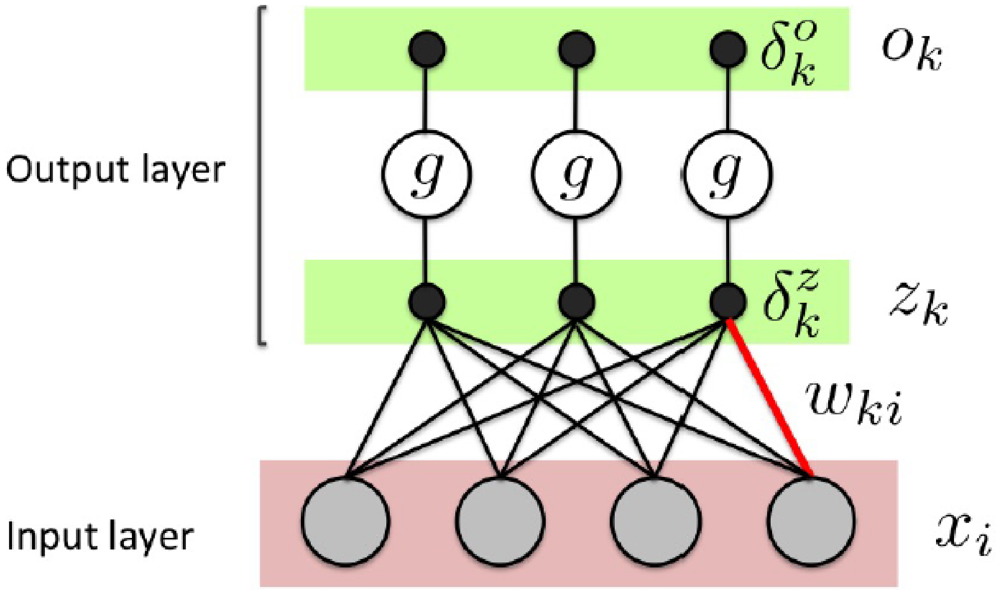


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \underbrace{\delta_k^o \cdot \frac{\partial o_k}{\partial z_k}}_{\delta_k^z}\frac{\partial z_k}{\partial w_{ki}}$$

Example: Single Layer Network



$$\frac{\partial z_k}{\partial w_{ki}}$$

- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Gradient descent iteration

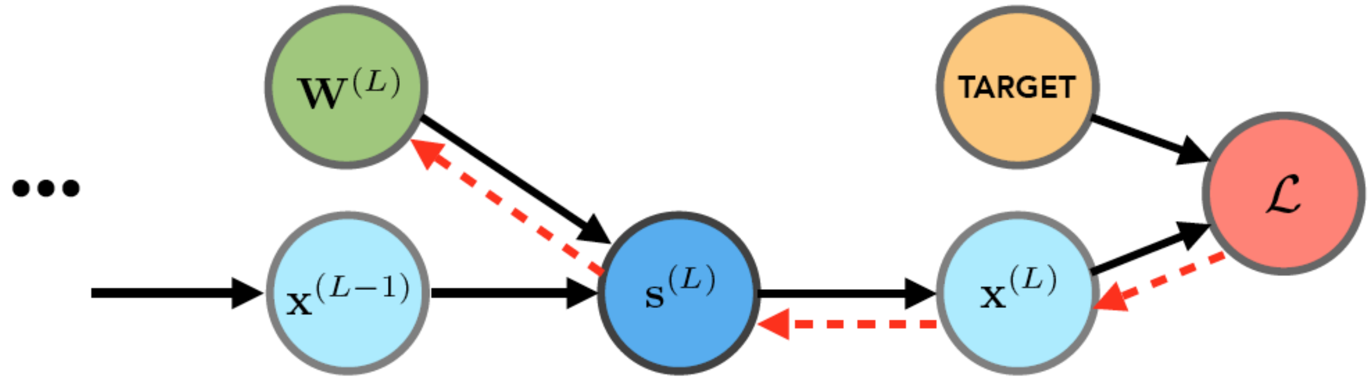
- Forward pass

1st layer	2nd layer		Loss
$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$ $\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$	$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$	•••	$\mathcal{L}$

- Backward pass

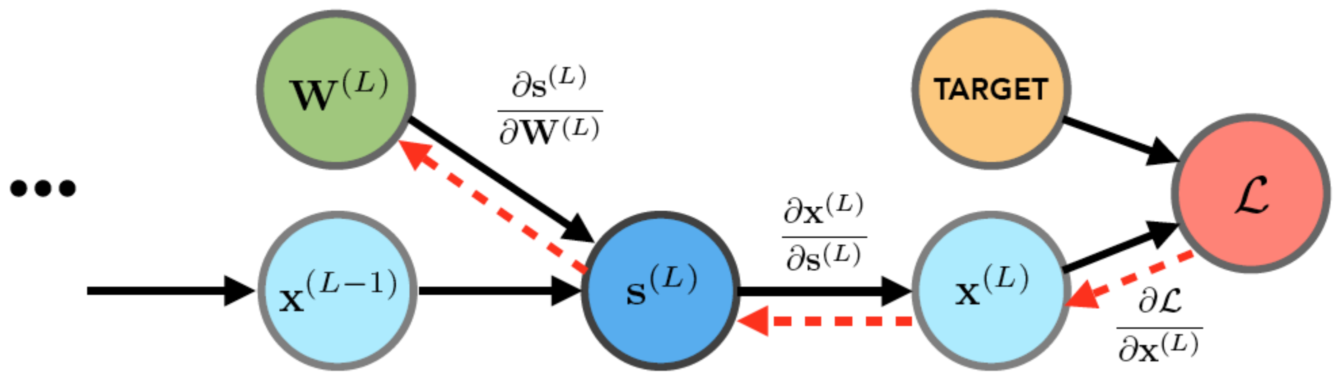
calculate $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$ let's start with the final layer: $\nabla_{W^{(L)}}\mathcal{L}$

to determine the chain rule ordering, we'll draw the dependency graph



Gradient descent iteration

- Backward pass



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

derivative of the non-linearity

$$\frac{\partial}{\partial \mathbf{W}^{(L)}} (\mathbf{W}^{(L)\intercal} \mathbf{x}^{(L-1)}) = \mathbf{x}^{(L-1)\intercal}$$

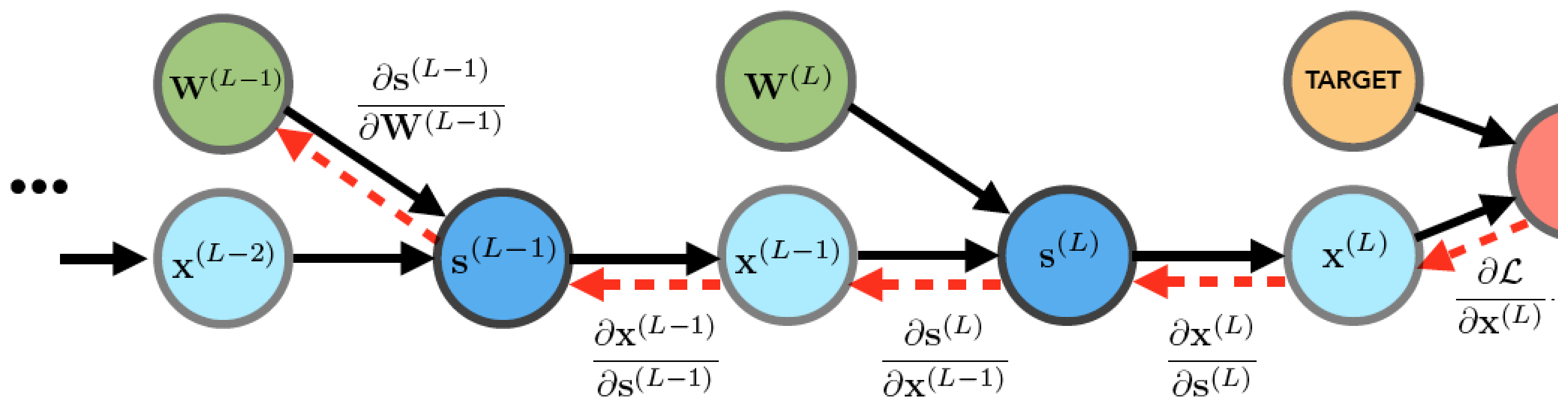
note $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

Gradient descent iteration

- Backward pass

now let's go back one more layer…

again we'll draw the dependency graph:



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

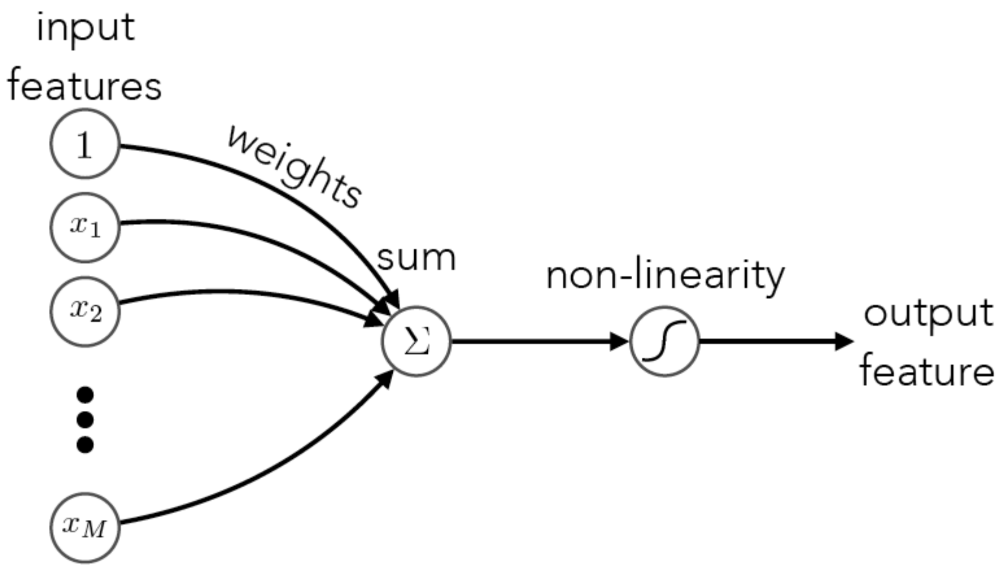
An implementation perspective

- Example: Univariate logistic least square model

$$s = wx + b$$

$$y = \sigma(s)$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$



Univariate chain rule

- A structured way to implement it
  - The goal is to write a program that efficiently computes the derivatives

Computing the loss:

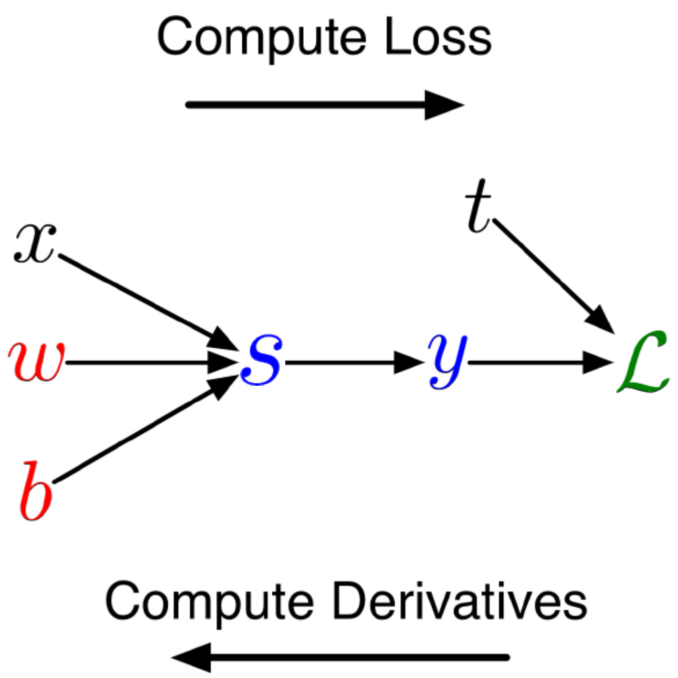
$$s = wx + b$$
$$y = \sigma(s)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\frac{d\mathcal{L}}{dy} = y - t$$
$$\frac{d\mathcal{L}}{ds} = \frac{d\mathcal{L}}{dy}\sigma'(s)$$
$$\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{ds}x$$
$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{ds}$$

Computation graph

- Represent the computations using a computation graph
  - Nodes: inputs & computed quantities
  - Edges: which nodes are computed directly as function of which other nodes



Univariate chain rule

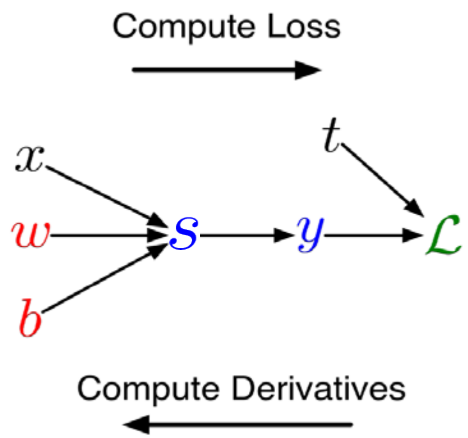
- A shorthand notation
  - Use $\delta_y := d\mathcal{L}/dy$, called the error signal
  - Note that the error signals are values computed by the program

Computing the loss:

$$s = wx + b$$
$$y = \sigma(s)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

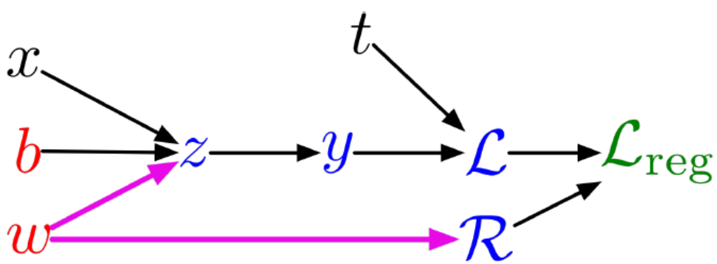
$$\delta_y = y - t$$
$$\delta_s = \delta_y \sigma'(s)$$
$$\delta_w = \delta_s x$$
$$\delta_b = \delta_s$$



Multivariate chain rule

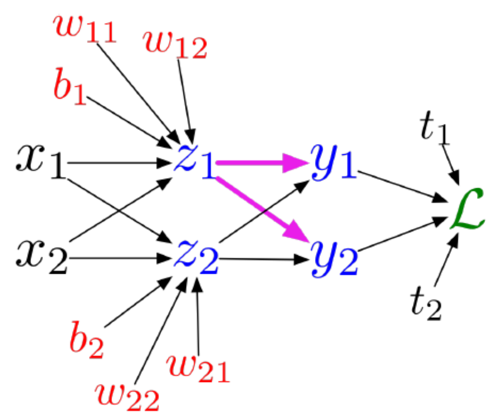
- The computation graph has fan-out > 1

L_2-Regularized regression



$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Multiclass logistic regression



$$z_\ell = \sum_j w_{\ell j} x_j + b_\ell$$
$$y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}}$$
$$\mathcal{L} = -\sum t_k \log y_k$$

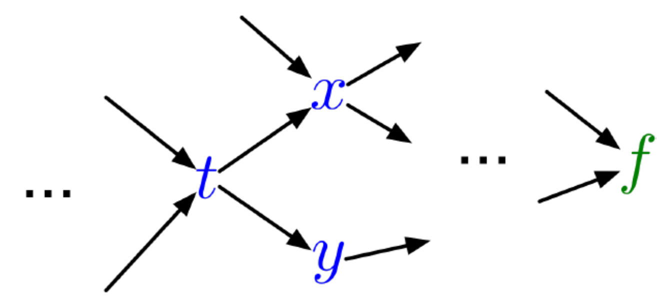
Multivariable chain rule

- Recall the distributed chain rule

Mathematical expressions
to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed
by our program



- The shorthand notation:

$$\delta_t = \delta_x \frac{dx}{dt} + \delta_y \frac{dy}{dt}$$

General Backpropagation

- Given a computation graph

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

v_N denotes the variable we're trying to compute derivatives of (e.g. loss)

forward pass

For $i = 1, \ldots, N$

Compute v_i as a function of $\mathrm{Pa}(v_i)$

backward pass

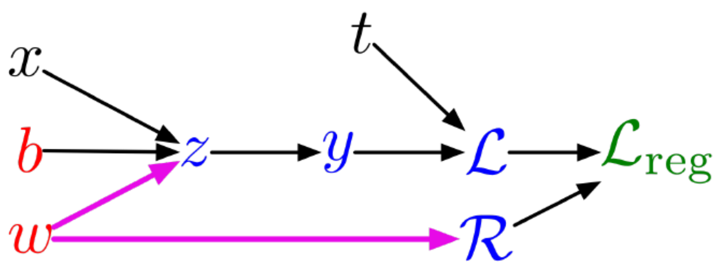
$\delta_{v_N} = 1$

For $i = N-1, \ldots, 1$

$\delta_{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \delta_{v_j} \frac{\partial v_j}{\partial v_i}$

General Backpropagation

- Example: univariate logistic least square regression



Forward pass:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\mathrm{reg}} = \mathcal{L} + \lambda\mathcal{R}$$

Backward pass:

$\delta_{\mathcal{L}_{\mathrm{reg}}} =$

$\delta_{\mathcal{R}} =$

$=$

$\delta_{\mathcal{L}} =$

$=$

$\delta_{y} =$

$=$

$\delta_{z} =$

$=$

$\delta_{w} =$

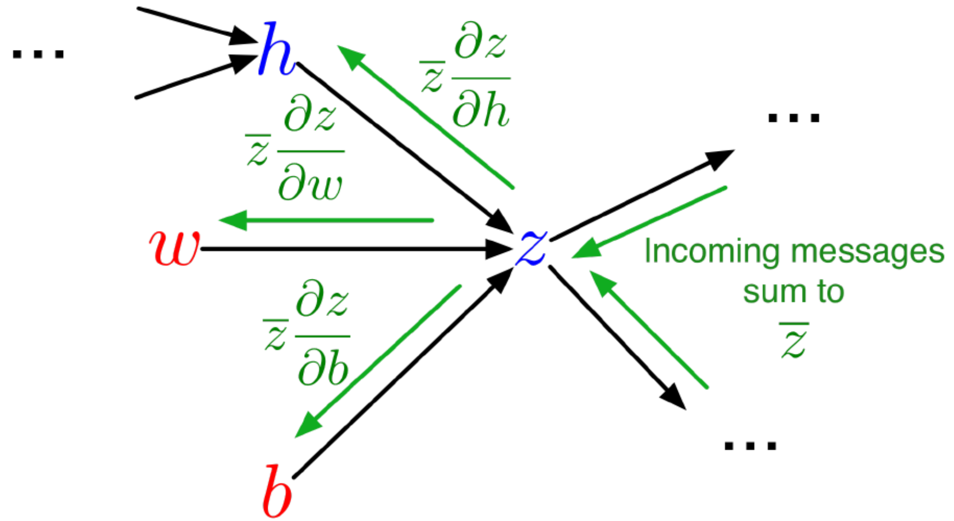
$=$

$\delta_{b} =$

$=$

General Backpropagation

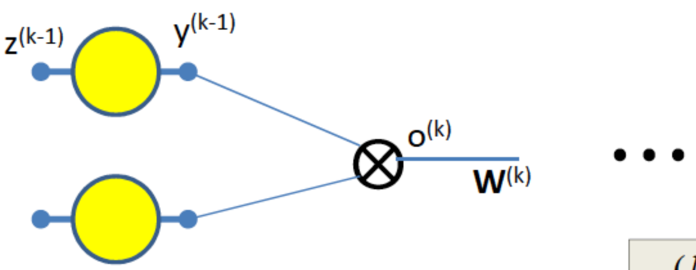
- Backprop as message passing:



  - Each node receives a set of messages from its children, which are aggregated into its error signal, then it passes messages to its parents
  - Modularity: each node only has to know how to compute derivatives w.r.t. its arguments – local computation in the graph

Patterns in backward flow

- Multiplicative node

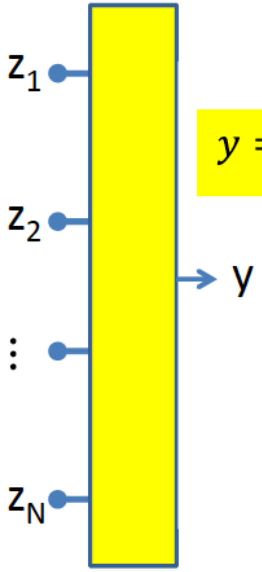


Forward: $\boxed{o_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)}}$

$$\frac{\partial L}{\partial y_j^{(k-1)}} = \frac{\partial L}{\partial o_i^{(k)}} \frac{\partial o_i^{(k)}}{\partial y_j^{(k-1)}} = y_l^{(k-1)} \frac{\partial L}{\partial o_i^{(k)}}$$

Patterns in backward flow

- Max node



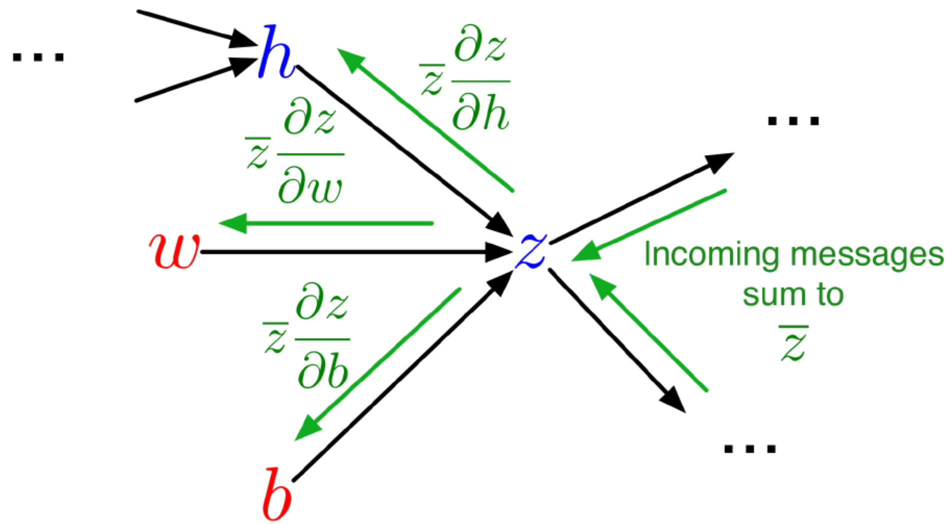
$y = \max_j z_j$

$$\frac{\partial y}{\partial z_i} = \begin{cases} 1, & i = \operatorname*{argmax}_j z_j \\ 0, & otherwise \end{cases}$$

- Vector equivalent of subgradient
  - 1 w.r.t. the largest incoming input
    - Incremental changes in this input will change the output
  - 0 for the rest
    - Incremental changes to these inputs will not change the output

Computation cost

- Forward pass: one add-multiply operation per weight
- Backward pass: two add-multiply operations per weight



- For a multilayer network, the cost is linear in the number of layers, quadratic in the number of units per layer

Backpropagation

- Backprop is used to train the majority of neural nets
  - Even generative network learning, or advanced optimization algorithms (second-order) use backprop to compute the update of weights
- However, backprop seems biologically implausible
  - No evidence for biological signals analogous to error derivatives
  - All the existing biologically plausible alternatives learn much more slowly on computers.
  - So how on earth does the brain learn???

Summary

- Multi-layer neural networks
- Inference and learning
  - Forward and Backpropagation

- Next time …
  - CNN