# Lecture 3: Basic Neural Networks: multi-layer neural networks

Xuming He SIST, ShanghaiTech Fall, 2020



#### Announcement

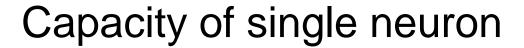
- Tutorial and TA office hour
  - □ Location: 教学中心101
  - □ Tutorial: 2,4, 8,11 周的周二 20:00 20:30
  - □ Office hour: 3, 6, 9, 12 周的周二 20:00 21:00
- Quiz 1 results are out
  - Check with TAs if you have any question
- A1 is out
  - □ Beijing Time: CST 2020/09/30 23:59:59
- Reference reading is listed at the end of lecture slides.



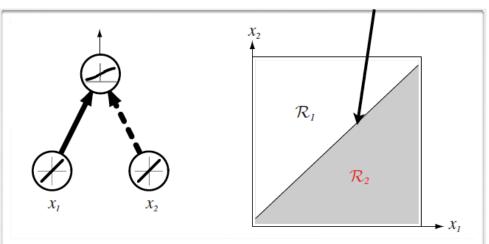
#### **Outline**

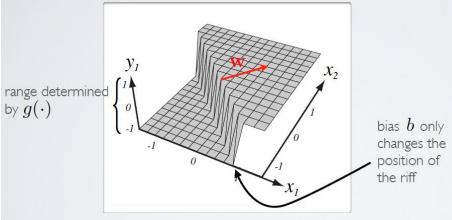
- Multi-layer neural networks
  - Limitations of single layer networks
  - Networks with single hidden layer
  - Sequential network architecture and variants
- Inference and learning
  - Forward and Backpropagation
  - Examples: one-layer network
  - General BP algorithm

Acknowledgement: Hugo Larochelle's, Mehryar Mohri@NYU's & Yingyu Liang@Princeton's course notes



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

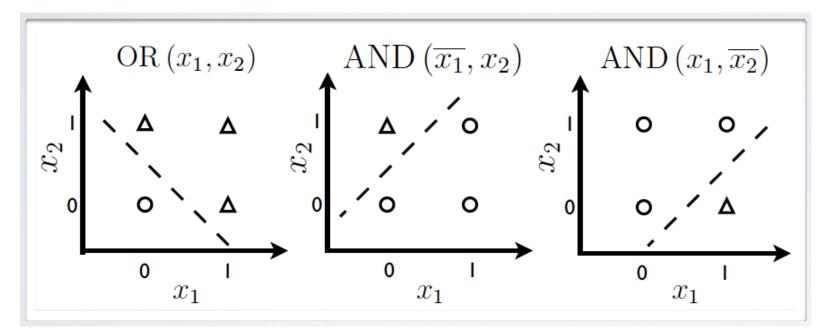




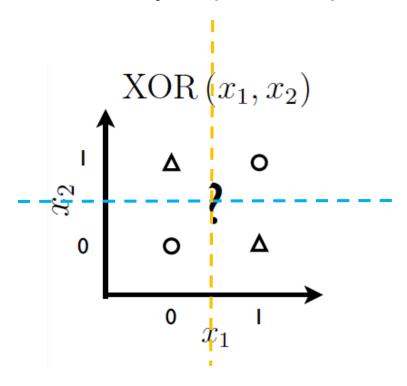
Can solve linearly separable problems

$$\mathcal{D} = \mathcal{D}^{+} \cup \mathcal{D}^{-}$$
$$\exists \mathbf{w}^{*}, \mathbf{w}^{*T}\mathbf{x} > 0, \ \forall \mathbf{x} \in \mathcal{D}^{+}$$
$$\mathbf{w}^{*T}\mathbf{x} < 0, \ \forall \mathbf{x} \in \mathcal{D}^{-}$$

Examples

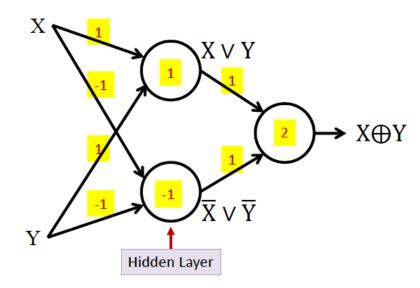


Can't solve non linearly separable problems

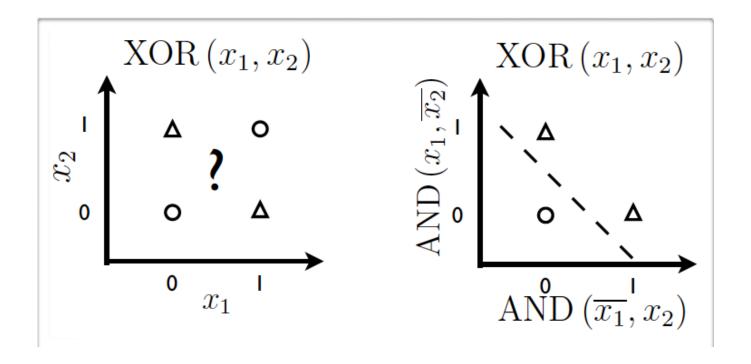


Can we use multiple neurons to achieve this?

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



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

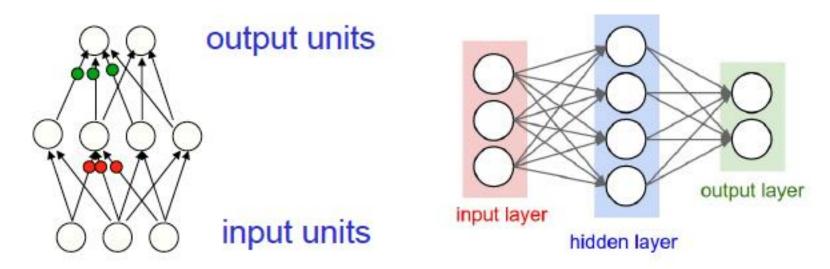


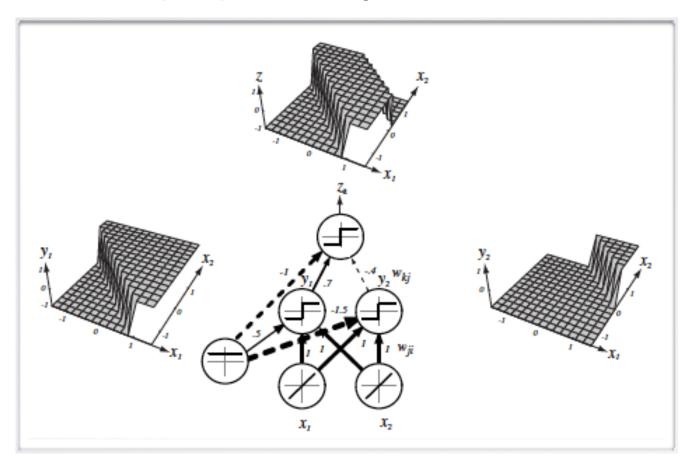
Figure: Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

Q: What if using linear activation in hidden layer?



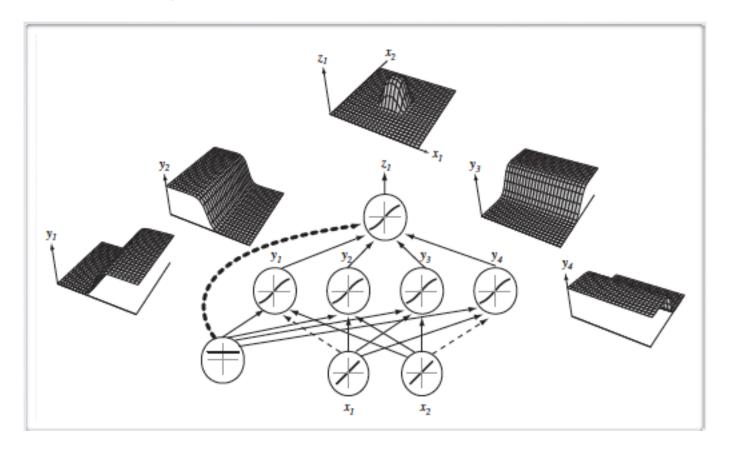
#### Capacity of neural network

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



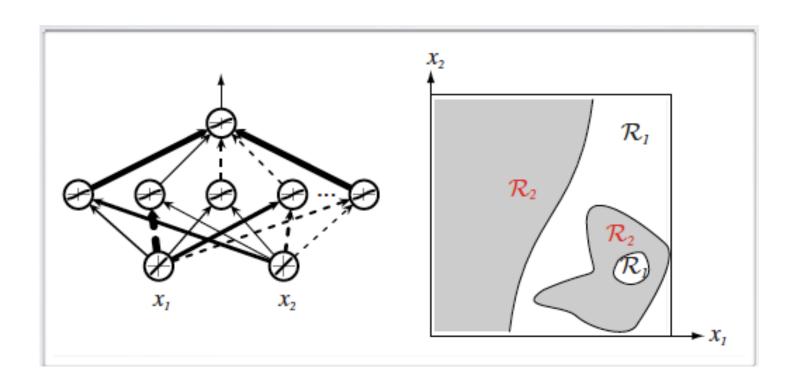


- Single hidden layer neural network
  - □ Form a stump/delta function



### Capacity of neural network

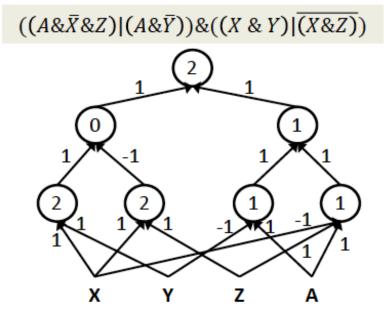
Single hidden layer neural network





#### 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





#### 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

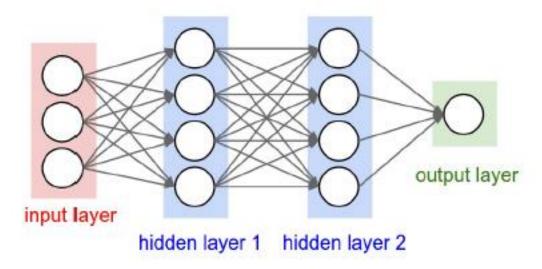
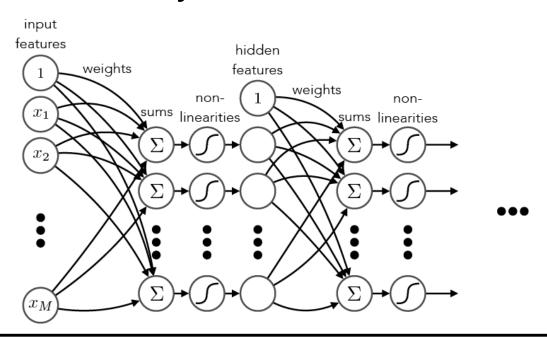


Figure: A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

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

### Multilayer networks

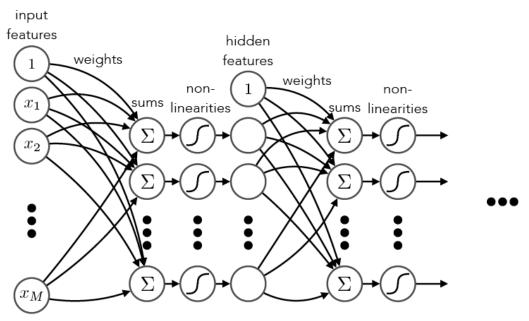


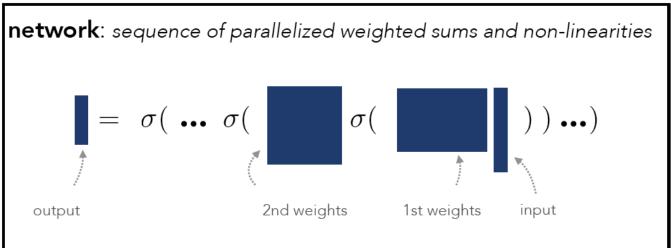
network: sequence of parallelized weighted sums and non-linearities

define 
$$\mathbf{x}^{(0)} \equiv \mathbf{x}$$
,  $\mathbf{x}^{(1)} \equiv \mathbf{h}$ , etc.

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)} \mathbf{T} \mathbf{x}^{(0)}$$
  $\mathbf{s}^{(2)} = \mathbf{W}^{(2)} \mathbf{T} \mathbf{x}^{(1)}$   $\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$   $\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$ 

### Multilayer networks

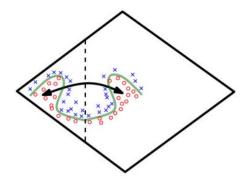


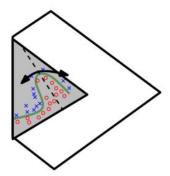


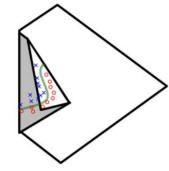


#### Why more layers (deeper)?

- 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.





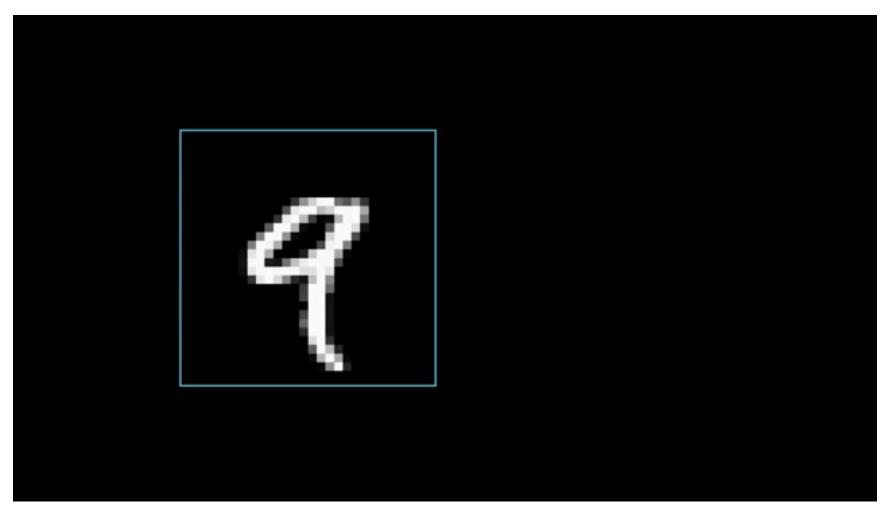




#### Why more layers (deeper)?

- 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.

### Why more layers (deeper)?



https://youtu.be/aircAruvnKk?list=PLZHQObOWTQDN U6R1\_67000Dx\_ZCJB-3pi

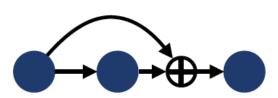
### Other network connectivity

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

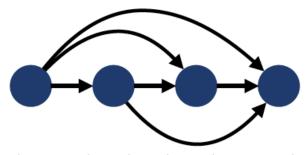


information may not be able to propagate easily make shorter paths to output

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



#### Computation in neural network

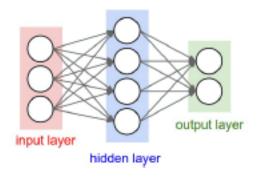
- We only need to know two algorithms
  - □ Inference/prediction: simply forward pass
  - □ Parameter learning: needs backward pass
- Basic fact:
  - □ A neural network is a function of composed operations

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

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

#### Inference example: Forward Pass

What does the network compute?



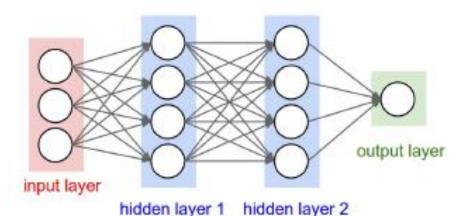
Output of the network can be written as:

$$h_j(x) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$
  
 $o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x) w_{kj})$ 

(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:



# 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)

Can be implemented efficiently using matrix operations

out = np.dot(W3, h2) + b3 # output neuron (Ix1)



#### Parameter learning: Backward Pass

- Supervised learning framework
  - Find weights:

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

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  $\eta$  is the learning rate (and E is error/loss)



#### Backward pass

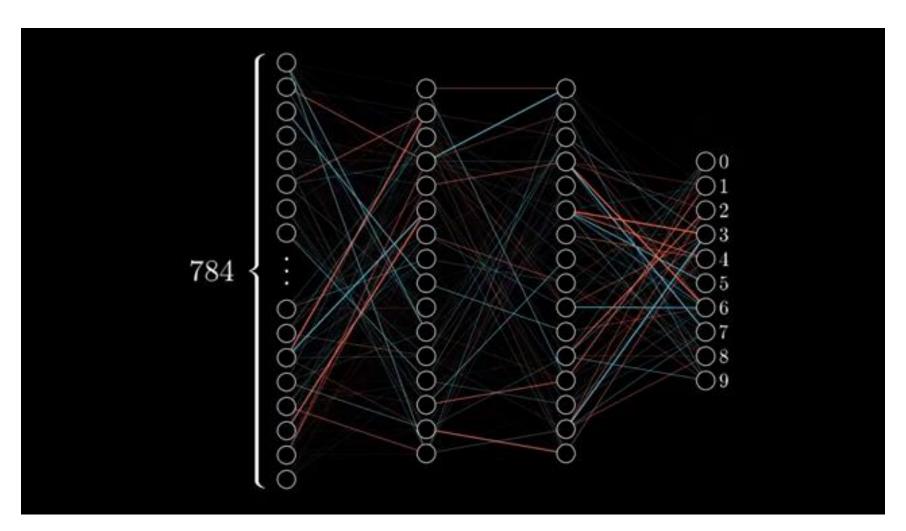
- Backpropagation
  - □ An efficient method for computing gradients in NNs
  - □ A neural network as a function of composed operations

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

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

→ use <u>chain rule</u> to calculate gradients

### Backward pass



https://www.youtube.com/watch?v=Ilg3gGewQ5U

28

#### Gradient descent iteration

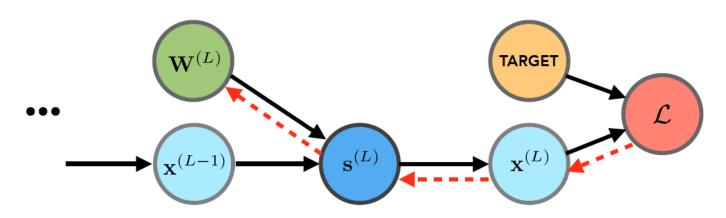
#### Forward pass

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$
  $\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$   $\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$   $\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$ 

#### Backward pass

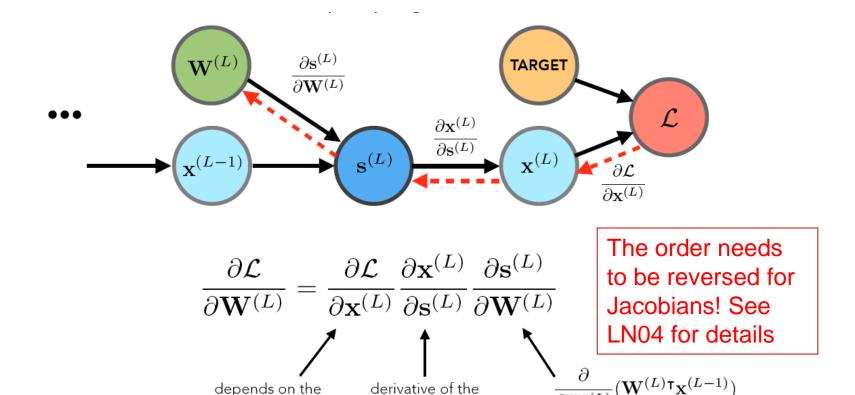
calculate  $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$  let's start with the final layer:  $\nabla_{W^{(L)}}\mathcal{L}$ 

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



#### Gradient descent iteration

#### Backward pass



non-linearity

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

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

form of the loss

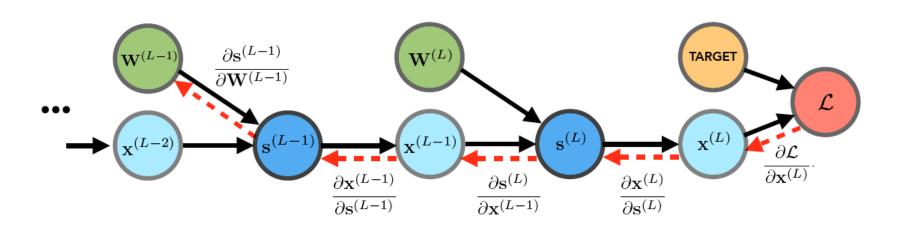
9/14/2020

#### Gradient descent iteration

#### Backward pass

now let's go back one more layer...

again we'll draw the dependency graph:

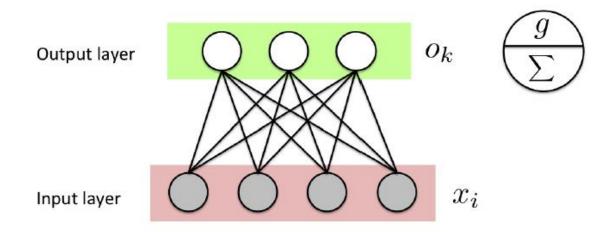


$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial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

The order needs to be reversed for Jacobians! See LN04 for details

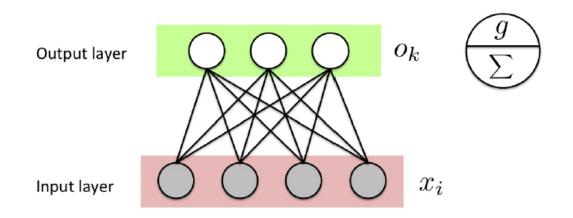


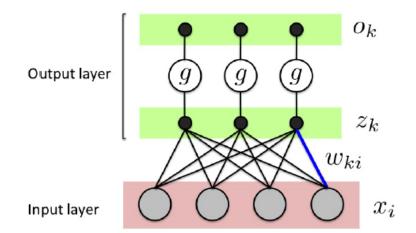
Let's take a single layer network





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





Output of unit k

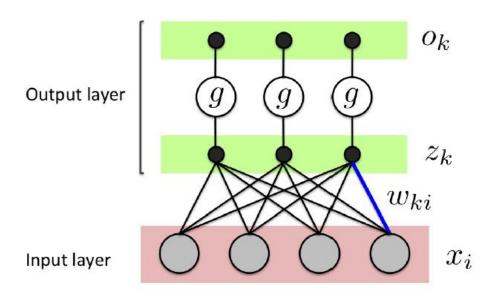
Output layer activation function

Net input to output unit k

Weight from input i to k

Input unit i

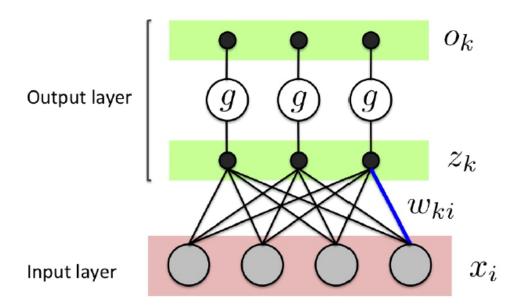




• Error gradients for single layer network:

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



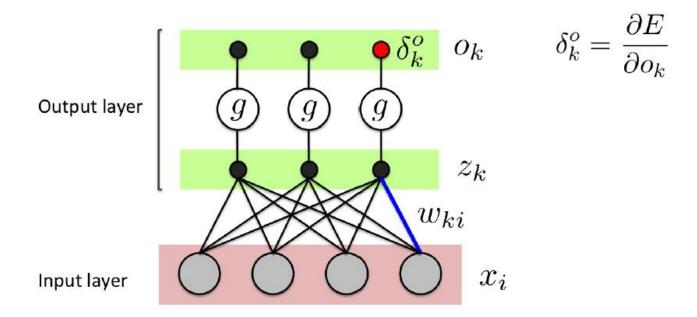


• Error gradients for single layer network:

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

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



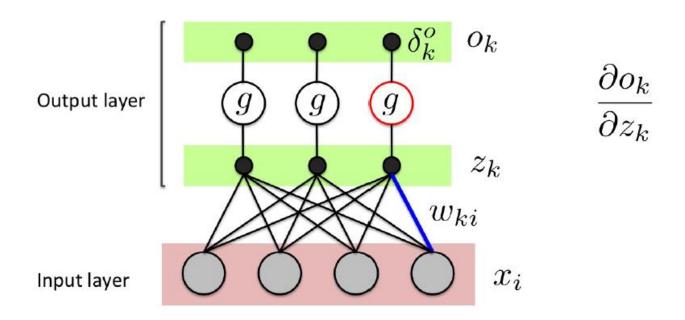


Error gradients for single layer network:

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



## Example: Single Layer Network

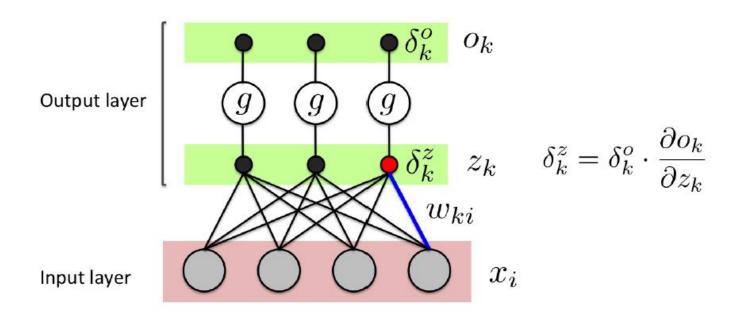


Error gradients for single layer network:

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



### Example: Single Layer Network

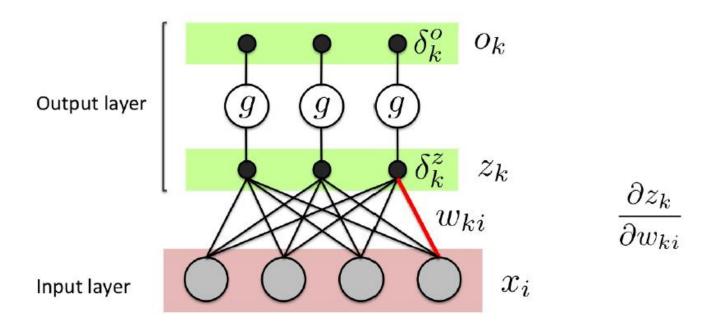


Error gradients for single layer network:

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



## Example: Single Layer Network



Error gradients for single layer network:

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



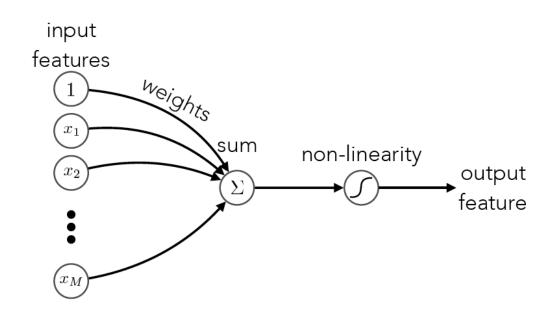
### Outline

- 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

## 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:

$$\frac{d\mathcal{L}}{dy} = y - t$$

$$\frac{d\mathcal{L}}{ds} = \frac{d\mathcal{L}}{dy}\sigma'(s)$$

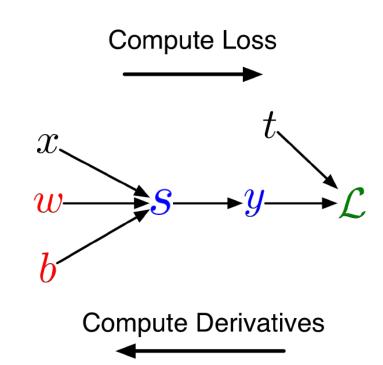
$$\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{ds}x$$

$$\frac{d\mathcal{L}}{db} = \frac{d\mathcal{L}}{ds}$$



## 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

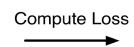
- A shorthand notation
  - $\square$  Use  $\delta_y := d\mathcal{L}/dy$  , called the error signal
  - Note that the error signals are values computed by the program

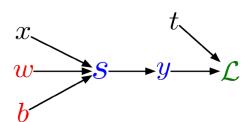
#### Computing the loss:

$$s = wx + b$$

$$y = \sigma(s)$$

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





Compute Derivatives

#### Computing the derivatives:

$$\delta_y = y - t$$

$$\delta_s = \delta_y \sigma'(s)$$

$$\delta_w = \delta_s x$$

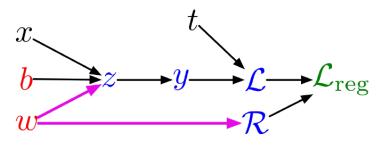
$$\delta_b = \delta_s$$



### Multivariate chain rule

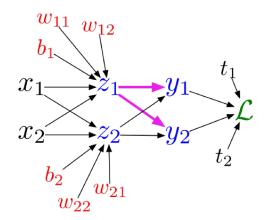
The computation graph has fan-out > 1

### L<sub>2</sub>-Regularized regression



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

#### Multiclass logistic regression

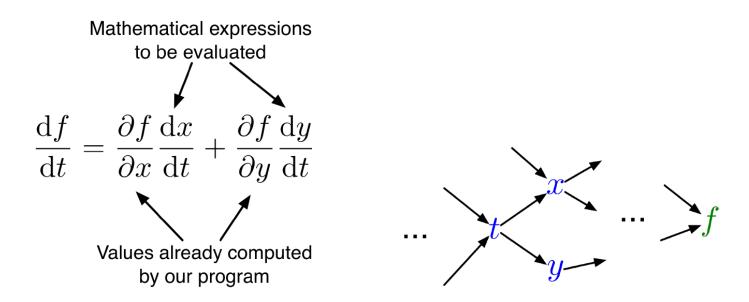


$$z_{\ell} = \sum_{j} w_{\ell j} x_{j} + b_{\ell}$$
 $y_{k} = \frac{e^{z_{k}}}{\sum_{\ell} e^{z_{\ell}}}$ 
 $\mathcal{L} = -\sum_{j} t_{k} \log y_{k}$ 



### Multivariable chain rule

#### Recall the distributed chain rule



The shorthand notation:

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



## **General Backpropagation**

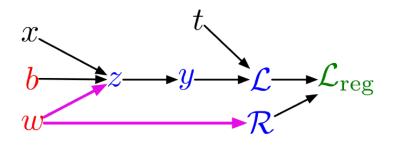
### Given a computation graph

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)

## **General Backpropagation**

Example: univariate logistic least square regression



### Forward pass:

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

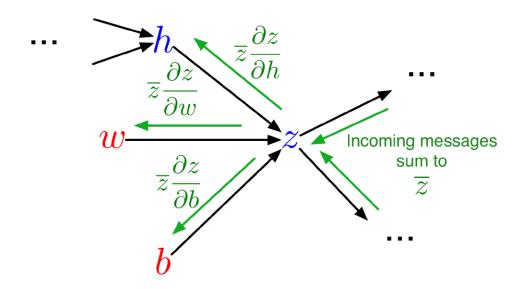
### **Backward pass:**

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



## **General Backpropagation**

Backprop as message passing:

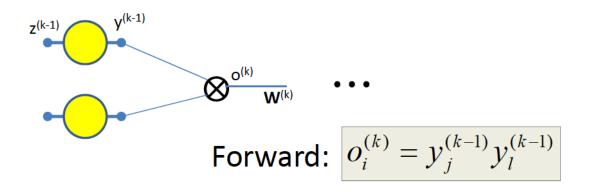


- Each node receives a set of messages from its children, which are aggregated into its error signal, then it passes messages to its parents
- Modularity: each node only has to know how to compute derivatives w.r.t. its arguments – local computation in the graph



### Patterns in backward flow

### Multiplicative node

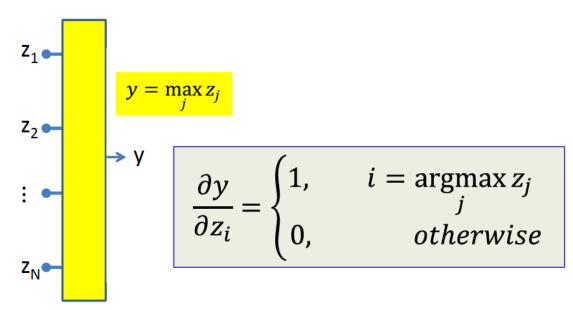


$$\frac{\partial L}{\partial y_j^{(k-1)}} = \frac{\partial L}{\partial o_i^{(k)}} \frac{\partial o_i^{(k)}}{\partial y_j^{(k-1)}} = y_l^{(k-1)} \frac{\partial L}{\partial o_i^{(k)}}$$

# .

### Patterns in backward flow

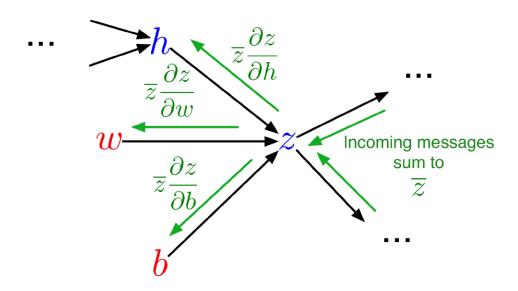
#### Max node



- 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???



## Coding examples

- Getting familiar with Pytorch
  - Python Tutorial: https://cs231n.github.io/python-numpy-tutorial/
  - PyTorch in 60 mins: https://pytorch.org/tutorials/beginner/deep\_learning\_60min\_blitz. html
- Predicting house prices
  - https://d2l.ai/chapter\_multilayer-perceptrons/kaggle-houseprice.html



## Summary

- Multi-layer neural networks
- Inference and learning
  - Forward and Backpropagation
- Next time ...
  - □ CNN
  - Quiz 2: Open book, but no electronic device is allowed.

### Reference:

- □ d2l.ai: 4.1-4.3, 4.7
- □ DLBook: Chapter 6