Lecture 2: Basic Artificial Neural Networks

Lan Xu

SIST, ShanghaiTech

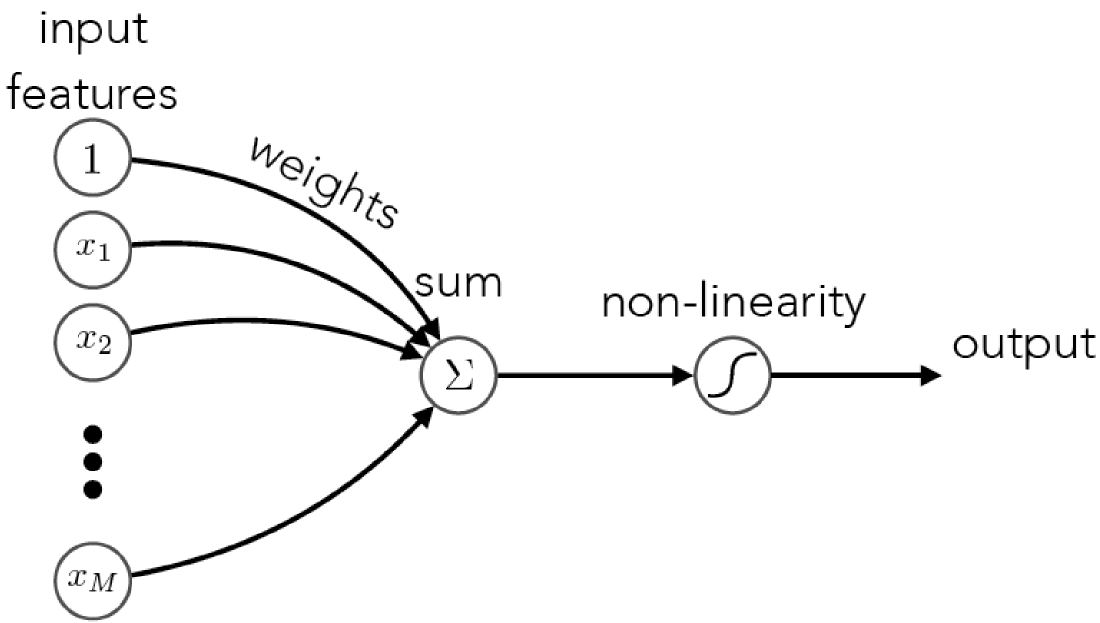
Fall, 2022

Outline

- Single layer neural networks
  - Network models
  - Example: Logistic Regression
- Multi-layer neural networks
  - Limitations of single layer networks
  - Networks with single hidden layer

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

Mathematical model of a neuron



artificial neuron: *weighted sum and non-linearity*

bias   input features

$$s = b + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^\mathsf{T}\mathbf{x}$$

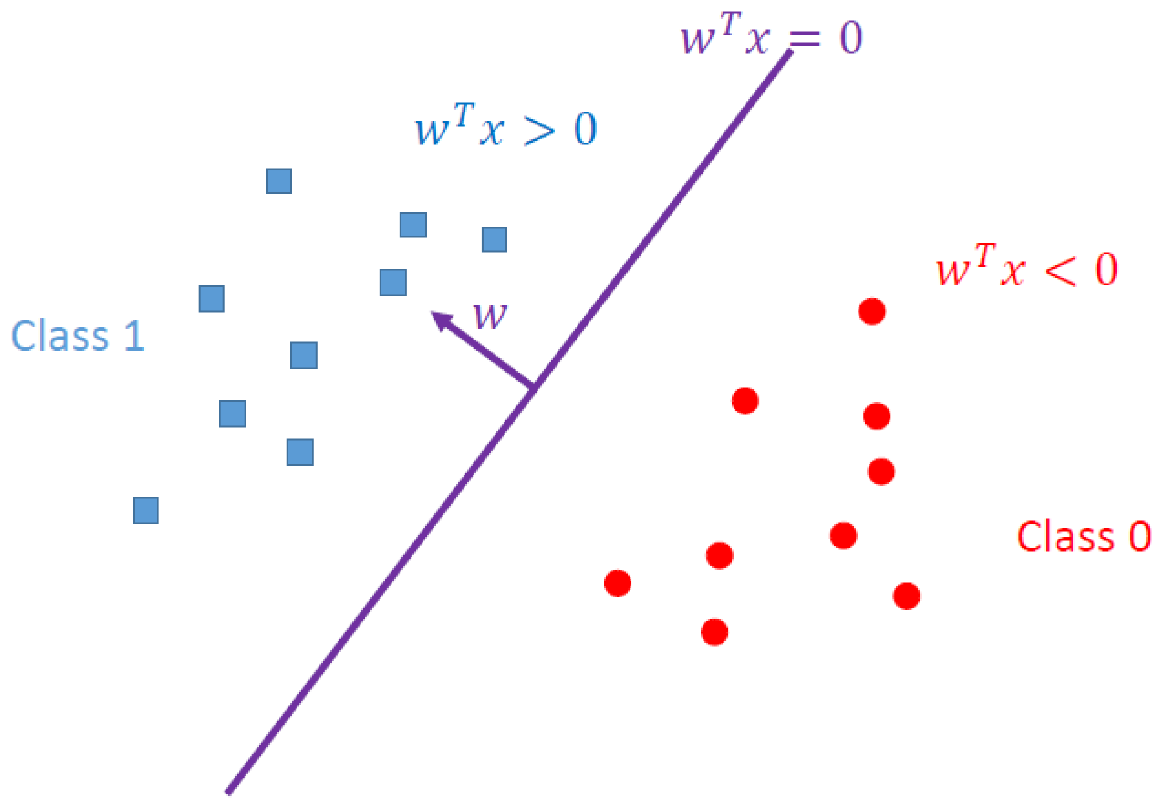
sum   weights

$$h = g(s)$$

output   non-linearity   sum

Single neuron as a linear classifier

- Binary classification

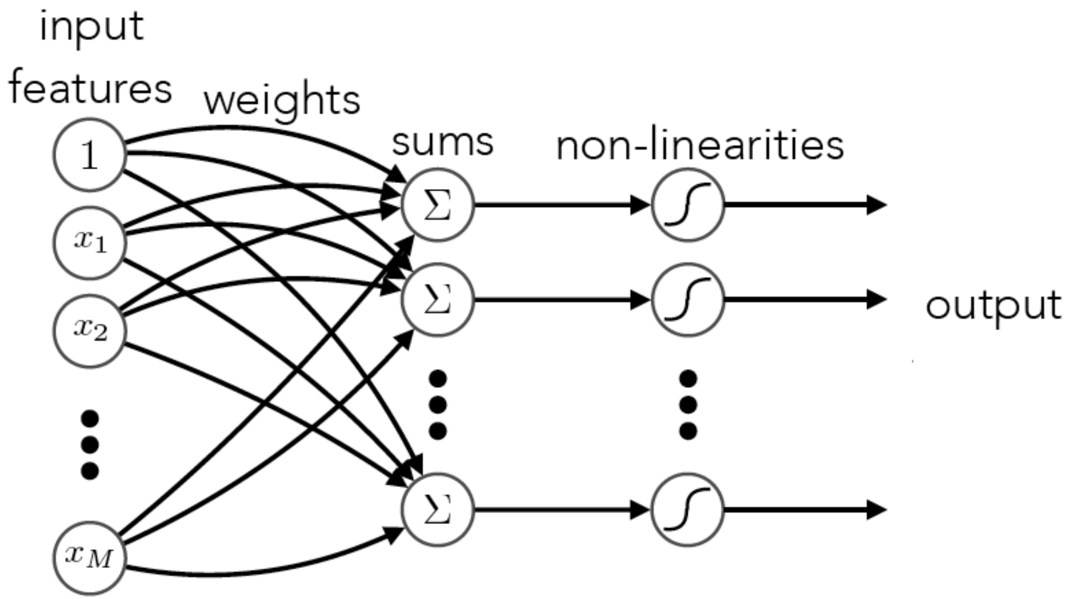


Outline

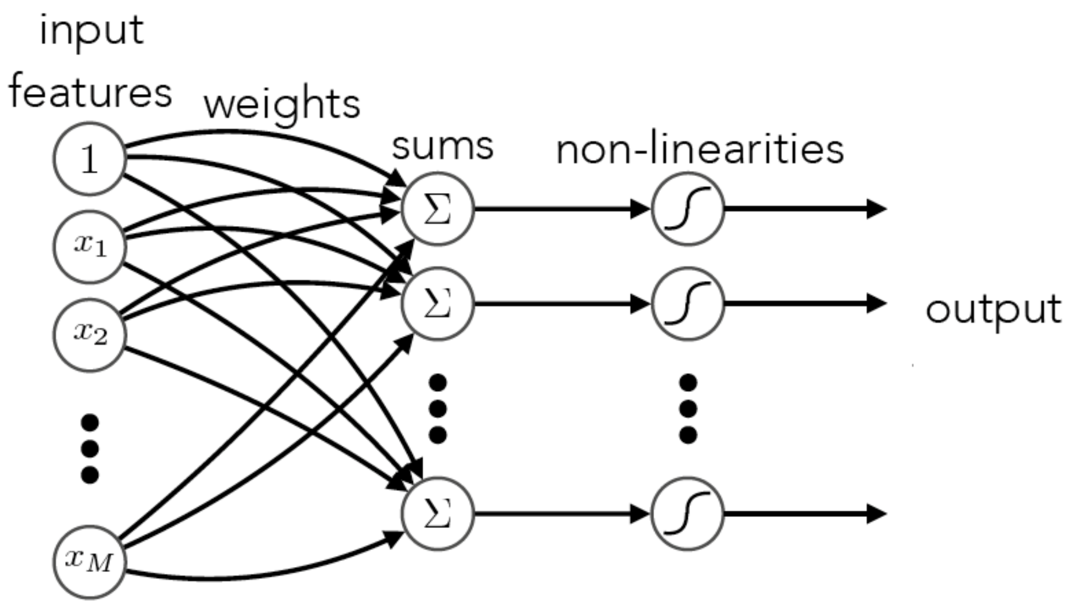
- Artificial neuron
  - Perceptron algorithm
- Single layer neural networks
  - Network models
  - Example: Logistic Regression
- Multi-layer neural networks
  - Limitations of single layer networks
  - Networks with single hidden layer

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

Single layer neural network



Single layer neural network

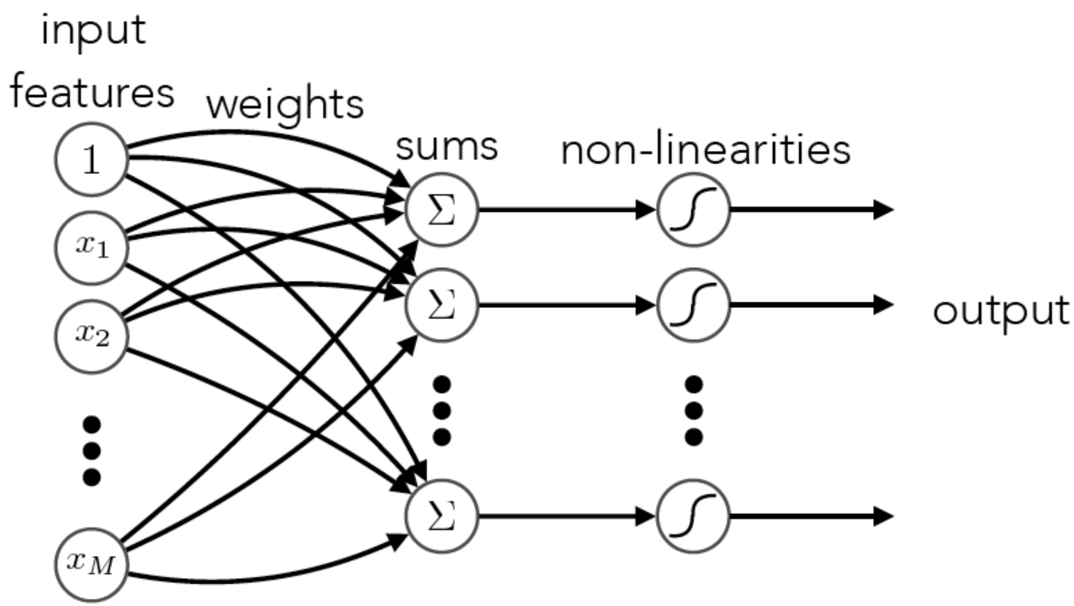


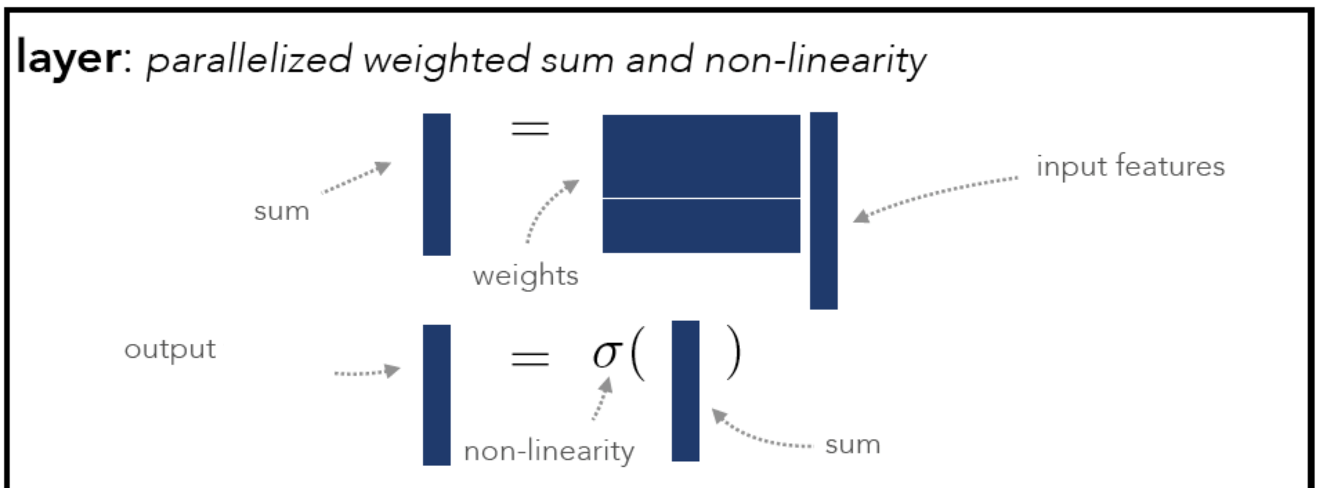
layer: *parallelized weighted sum and non-linearity*

one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x}$ $\longrightarrow$ $\mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

$$\mathbf{h} = \sigma(\mathbf{s})$$

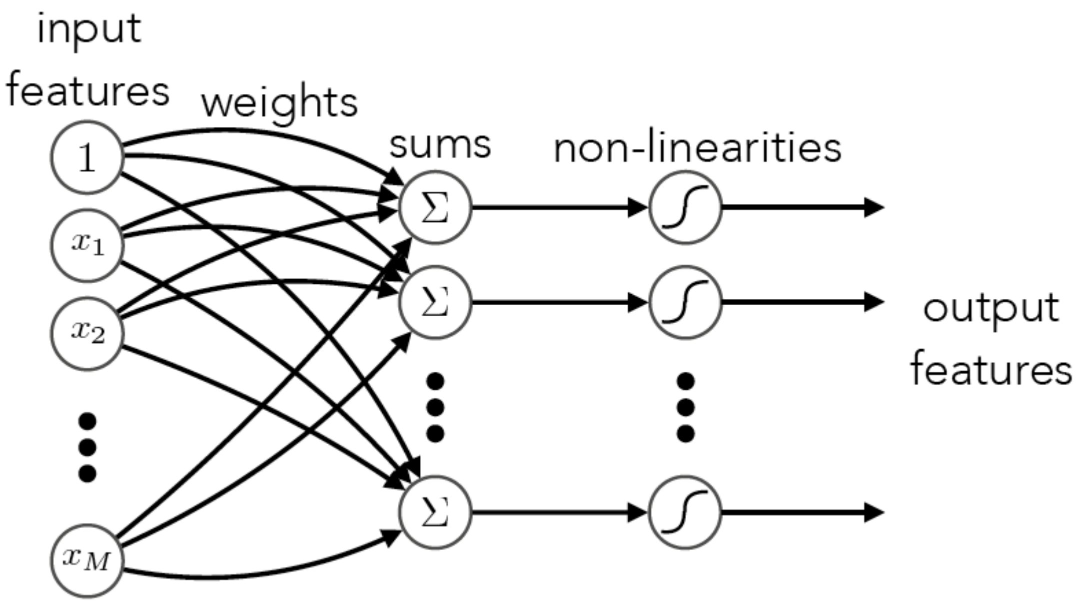
Single layer neural network





What is the output?

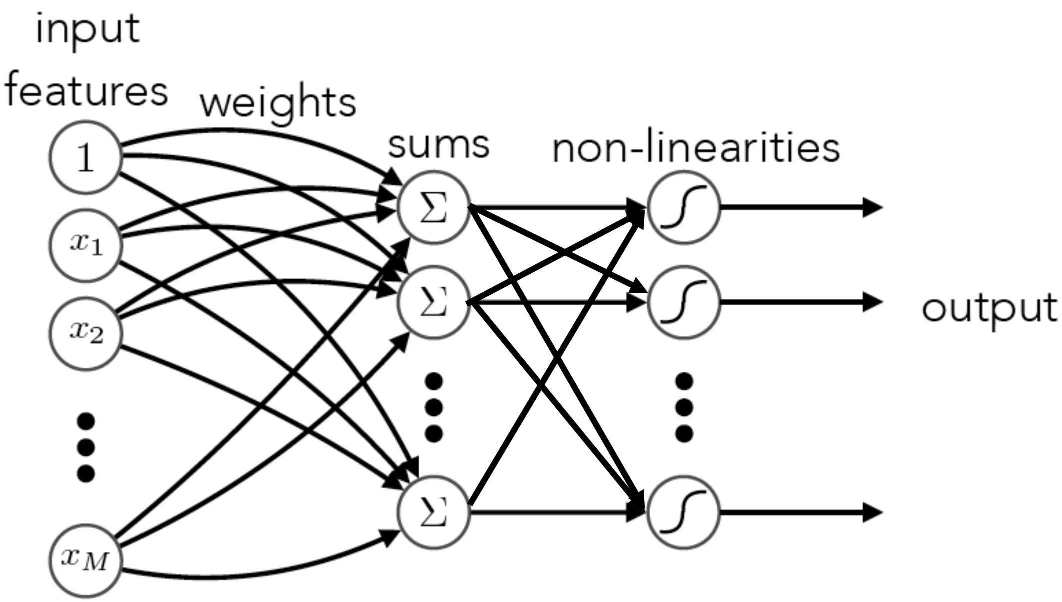
- Element-wise nonlinear functions
  - Independent feature/attribute detectors



$$\mathbf{h} = [h_j] \qquad h_j = \sigma(s_j) = \sigma(\mathbf{w}_j^\intercal \mathbf{x})$$

What is the output?

- Nonlinear functions with vector input
  - Competition between neurons

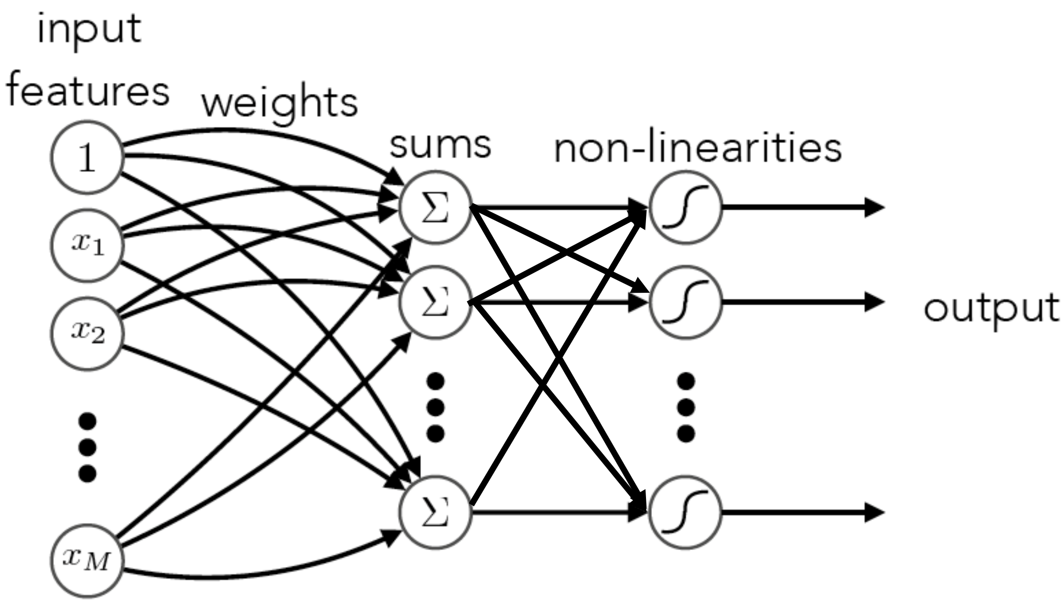


$$\mathbf{h} = [h_j]$$

$$h_j = g(\mathbf{s}) = g(\mathbf{w}_1^\intercal \mathbf{x}, \cdots, \mathbf{w}_m^\intercal \mathbf{x})$$

What is the output?

- Nonlinear functions with vector input
  - Example: Winner-Take-All (WTA)

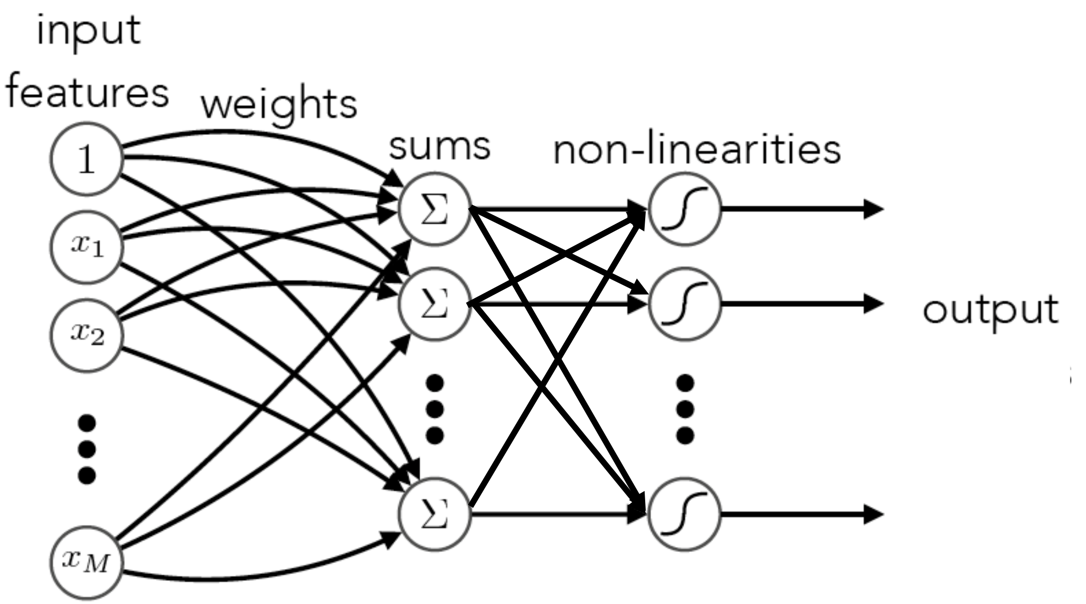


$$\mathbf{h} = [h_j]$$

$$h_j = g(\mathbf{s}) = \begin{cases} 1 & \text{if } j = \arg\max_i \mathbf{w}_i^\intercal \mathbf{x} \\ 0 & \text{if otherwise} \end{cases}$$

A probabilistic perspective

- Change the output nonlinearity



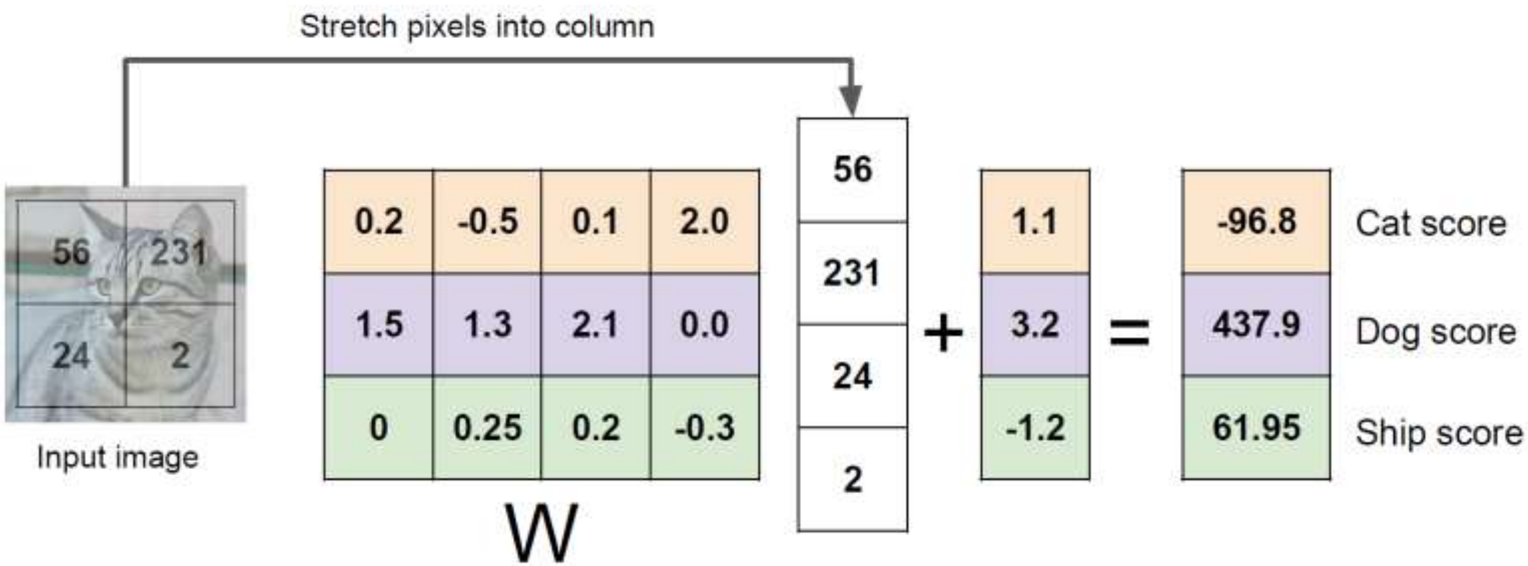
  - From WTA to Softmax function

scores = unnormalized log probabilities of the classes.

$$\boxed{P(Y=k|X=x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s = f(x_i; W)}$$

Multiclass linear classifiers

- Example with an image with 4 pixels, and 3 classes (cat/dog/ship)



- The WTA prediction: one-hot encoding of its predicted label

$$y = 1 \Leftrightarrow y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad y = 2 \Leftrightarrow y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad y = 3 \Leftrightarrow y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Probabilistic outputs

scores = unnormalized log probabilities of the classes.

$$P(Y=k|X=x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

	unnormalized log probabilities	exp →	unnormalized probabilities	normalize →	probabilities
cat	**3.2**		**24.5**		**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

How to learn a multiclass classifier?

- Define a loss function and do minimization
  - Given training data $\{(x_i, y_i): 1 \leq i \leq n\}$ i.i.d. from distribution D
  - Find $y = f(x) \in \mathcal{H}$ that minimizes $\hat{L}(f) = \frac{1}{n}\sum_{i=1}^{n} l(f, x_i, y_i)$
  - s.t. the expected loss is small

$$L(f) = \mathbb{E}_{(x,y)\sim D}[l(f, x, y)]$$

Empirical loss

Learning a multiclass linear classifier

- Design a loss function for multiclass classifiers
  - Perceptron?
    - Yes, see homework
  - Hinge loss
    - The SVM and max-margin (see CS231n)
  - Probabilistic formulation
    - Log loss and logistic regression
- Generalization issue
  - Avoid overfitting by regularization

Example: Logistic Regression

- Learning loss: negative log likelihood

scores = unnormalized log probabilities of the classes.

$$\boxed{P(Y=k|X=x_i)=\frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad \boxed{s=f(x_i;W)}$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$\boxed{L_i=-\log P(Y=y_i|X=x_i)}$$

- Learning loss: example

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

	unnormalized log probabilities	exp →	normalize → probabilities
cat	**3.2**	**24.5**	**0.13**
car	5.1	164.0	0.87
frog	-1.7	0.18	0.00

→ L_i = -log(0.13)
= **0.89**

Logistic Regression

- Learning loss: questions

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

Q: What is the min/max possible loss L_i?

	unnormalized log probabilities	exp →	unnormalized probabilities	normalize →	probabilities
cat	**3.2**		**24.5**		**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

→ L_i = -log(0.13) = **0.89**

Logistic Regression

- Learning loss: questions

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

Q2: Usually at initialization W is small so all s ≈ 0. What is the loss?

cat	**3.2**		**24.5**		**0.13**
car	5.1	exp →	164.0	normalize →	0.87
frog	-1.7		0.18		0.00

unnormalized log probabilities

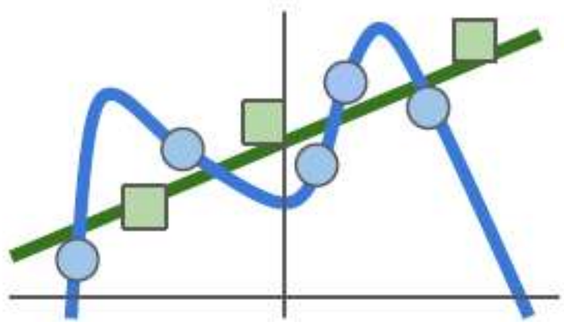
probabilities

→ L_i = -log(0.13) = **0.89**

Learning with regularization

- Constraints on hypothesis space
  - Similar to Linear Regression

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\textbf{Data loss}\text{: Model predictions should match training data}} + \underbrace{\lambda R(W)}_{\textbf{Regularization}\text{: Model should be “simple”, so it works on test data}}$$



Learning with regularization

- Regularization terms

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Max norm regularization (might see later)

- Priors on the weights
  - Bayesian: integrating out weights
  - Empirical: computing MAP estimate of W

L1 vs L2 regularization

https://www.youtube.com/watch?v=jEVh0uheCPk

L1 vs L2 regularization

- Sparsity

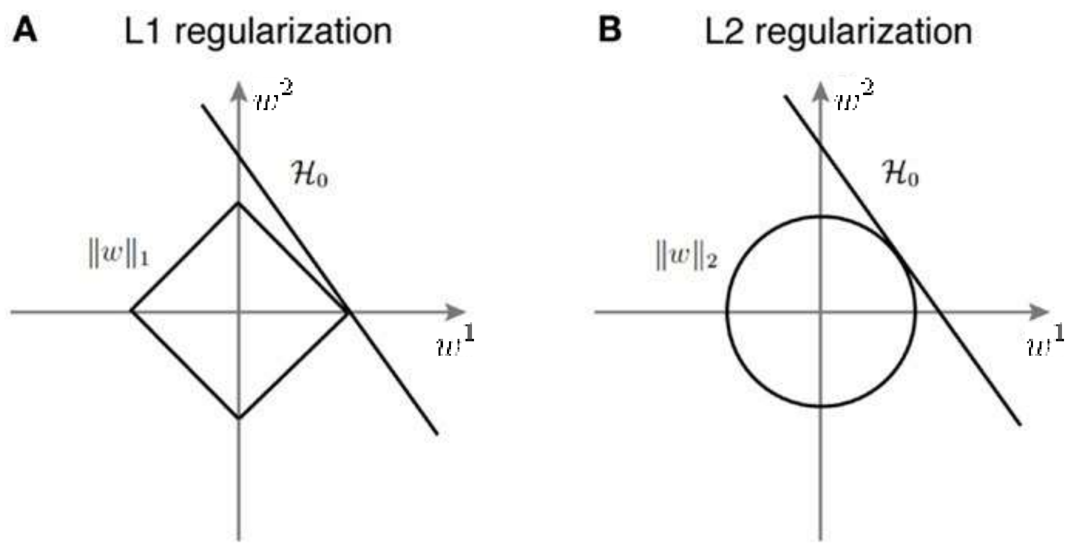
$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$
$$w_3 = [0.5, 0.5, 0, 0]$$

$$f(x) = w^\mathsf{T} x$$

$$w_1^\mathsf{T} x = w_2^\mathsf{T} x = w_3^\mathsf{T} x$$

$$\|w_1\|^2 = |w_1| = 1$$
$$\|w_2\|^2 = 4/16 = 1/4, |w_2| = 1$$
$$\|w_3\|^2 = 2/4 = 1/2, |w_3| = 1$$



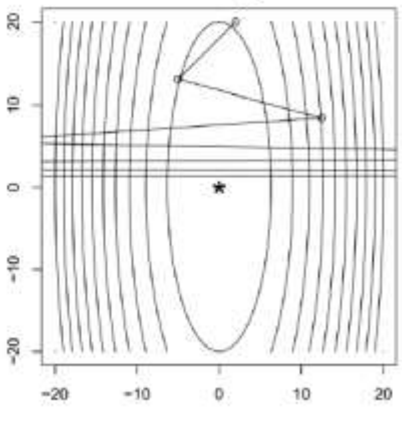
Optimization: gradient descent

- Gradient descent

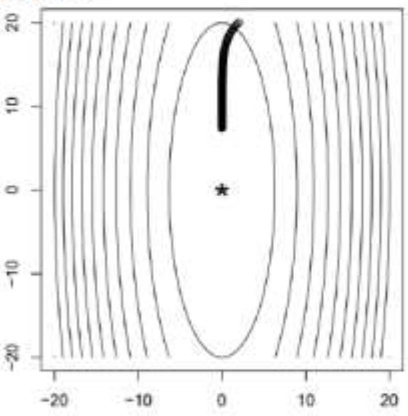
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- Learning rate matters

$\eta_t = t$, it is too big



too small η_t, after 100 iterations



Optimization: gradient descent

- Stochastic gradient descent

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

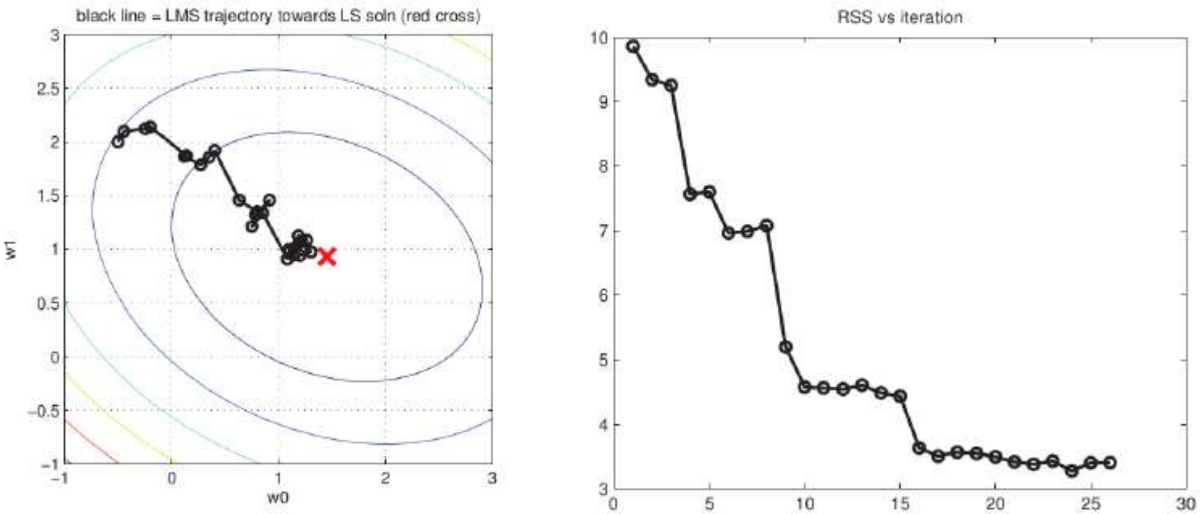
Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Optimization: gradient descent

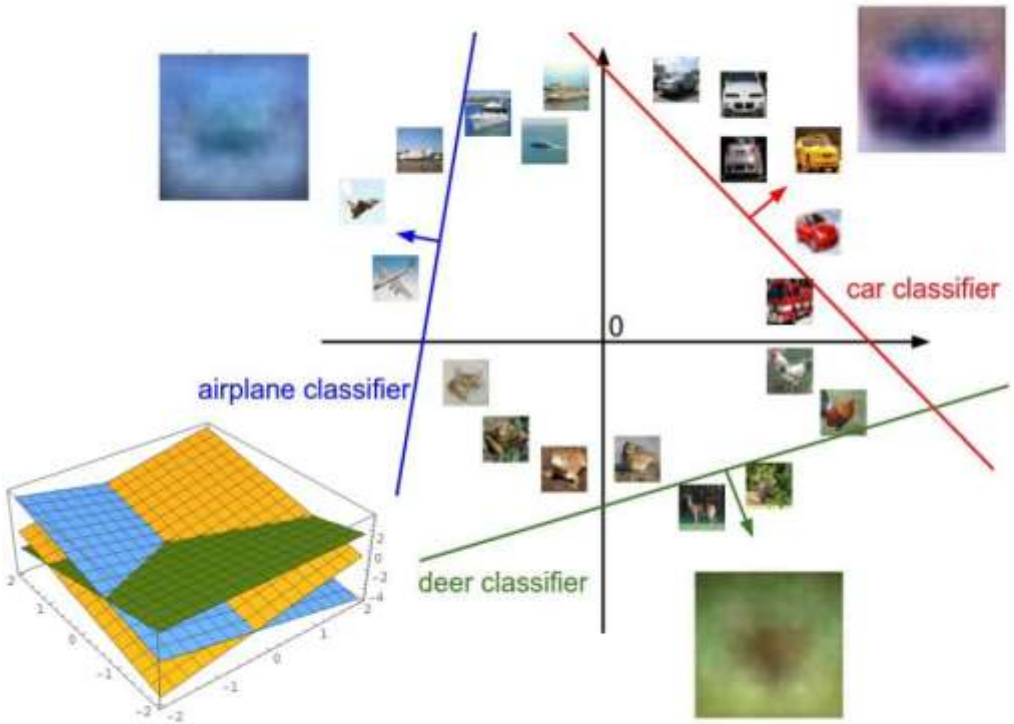
- Stochastic gradient descent



- the objective does not always decrease for each step
- comparing to GD, SGD needs more steps, but each step is cheaper
- mini-batch, say pick up 100 samples and do average, may accelerate the convergence

Interpreting network weights

- What are those weights?



$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

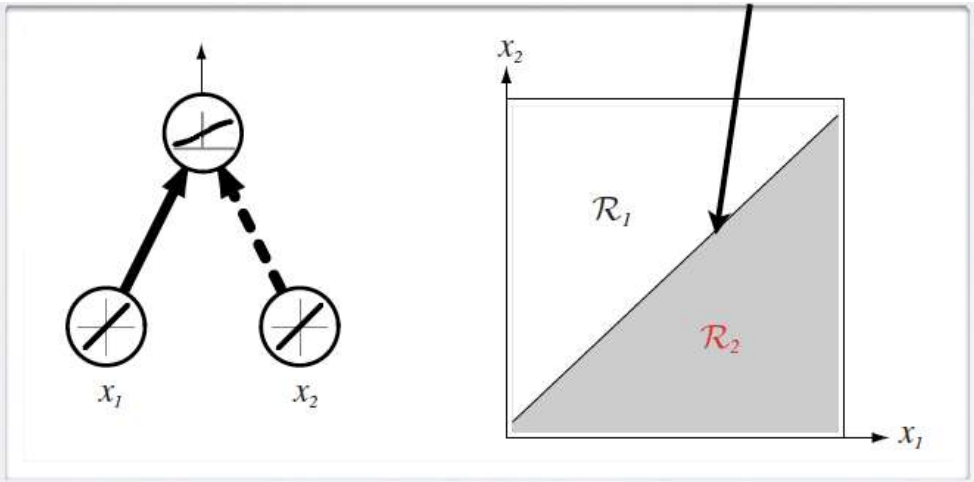
Outline

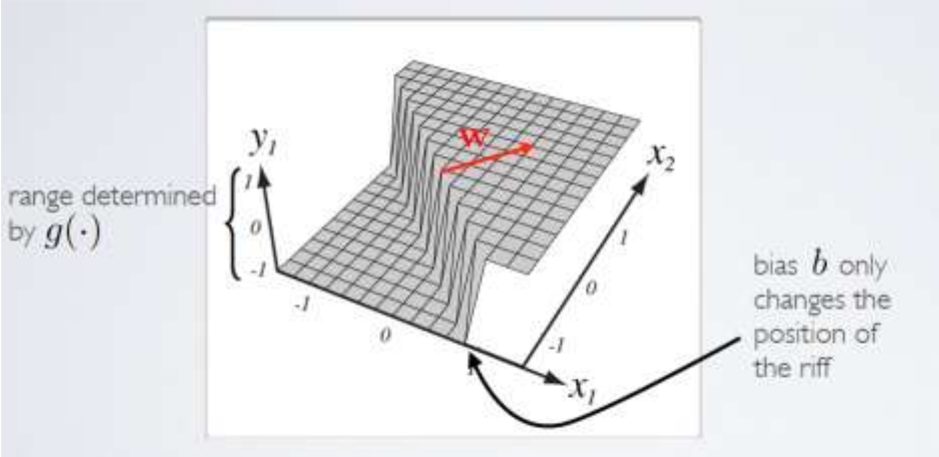
- Single layer neural networks
  - Network models
  - Example: Logistic Regression
- Multi-layer neural networks
  - Limitations of single layer networks
  - Networks with single hidden layer

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

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

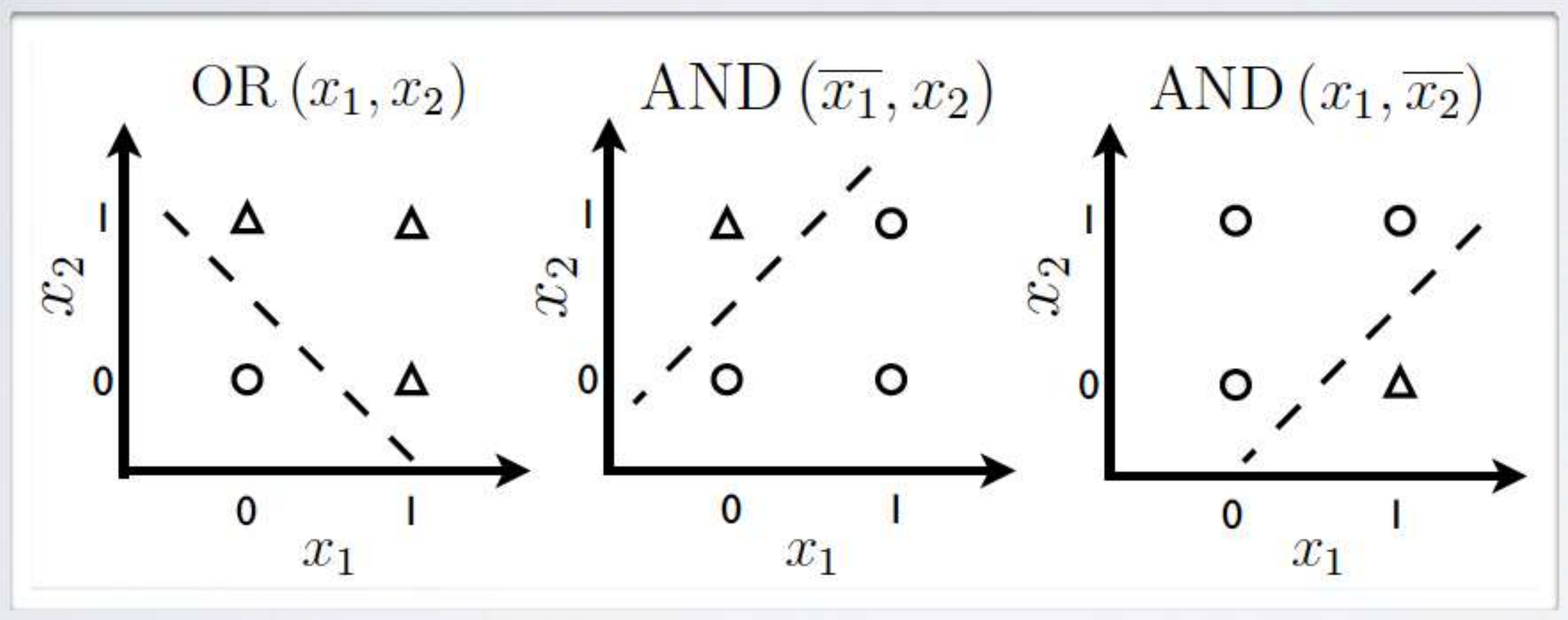
- Can solve linearly separable problems

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

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

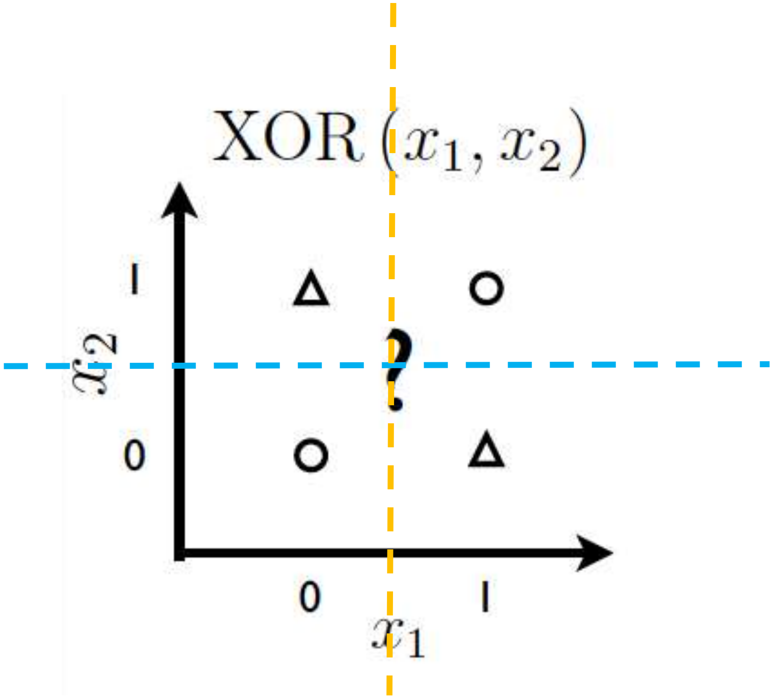
$${\mathbf{w}^*}^\mathsf{T}\mathbf{x} < 0,\ \forall \mathbf{x} \in \mathcal{D}^-$$

  - Examples



Capacity of single neuron

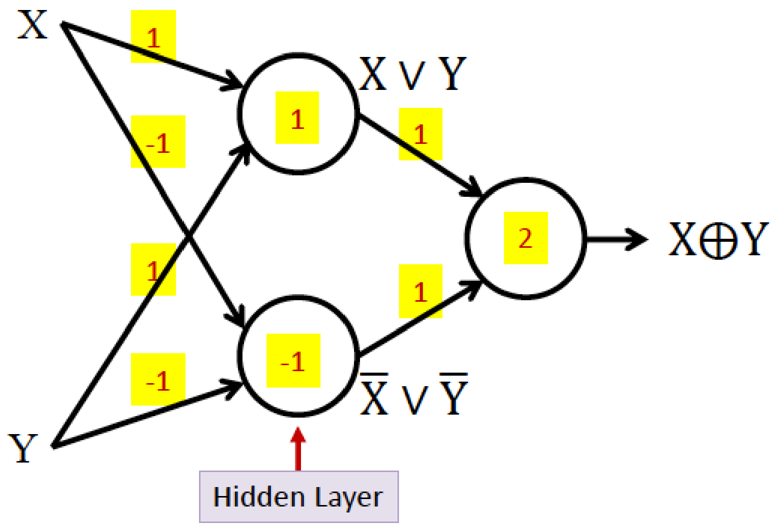
- Can't solve non linearly separable problems



- Can we use multiple neurons to achieve this?

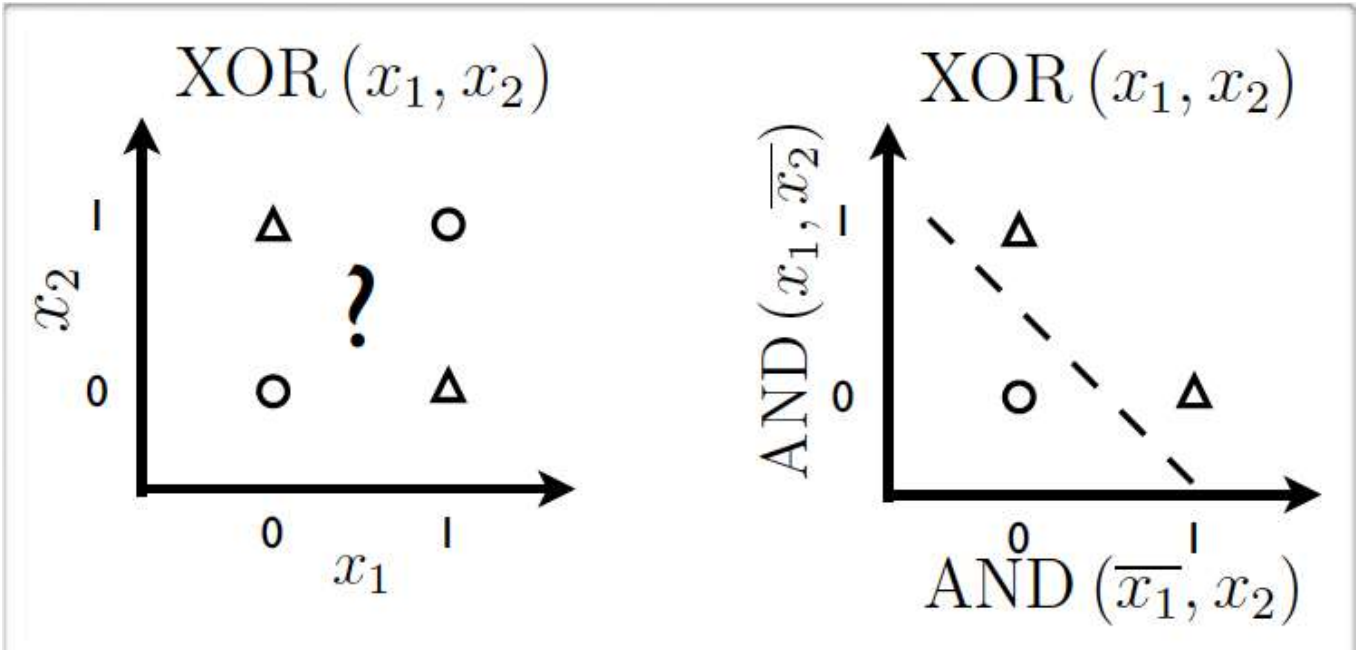
Capacity of single neuron

- 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

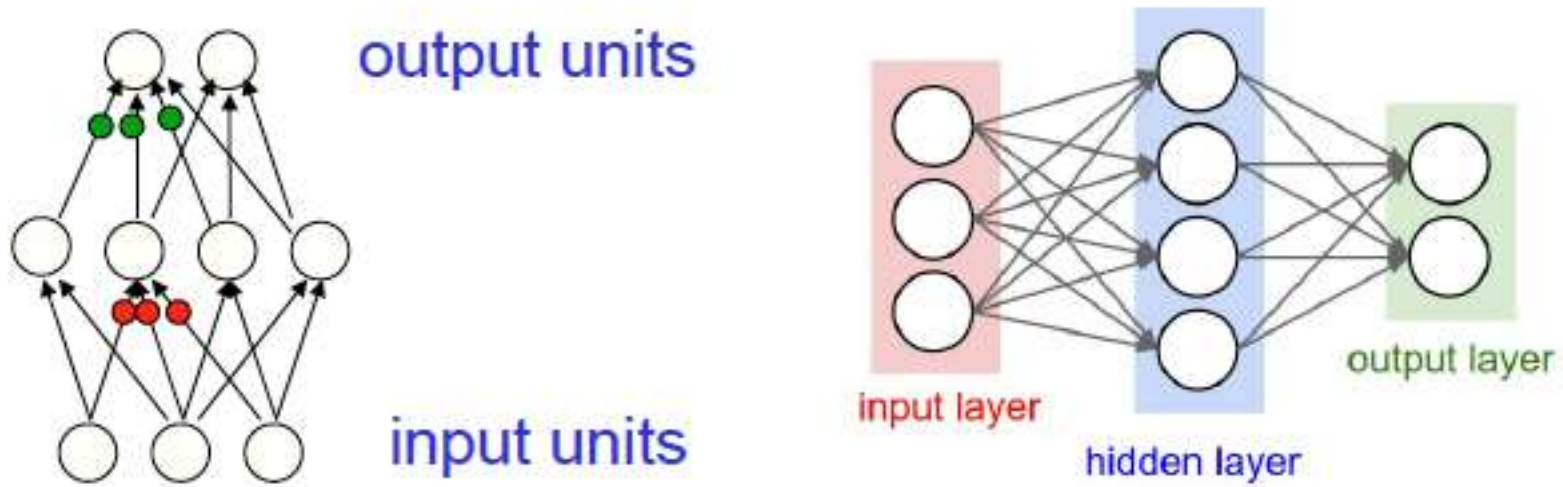
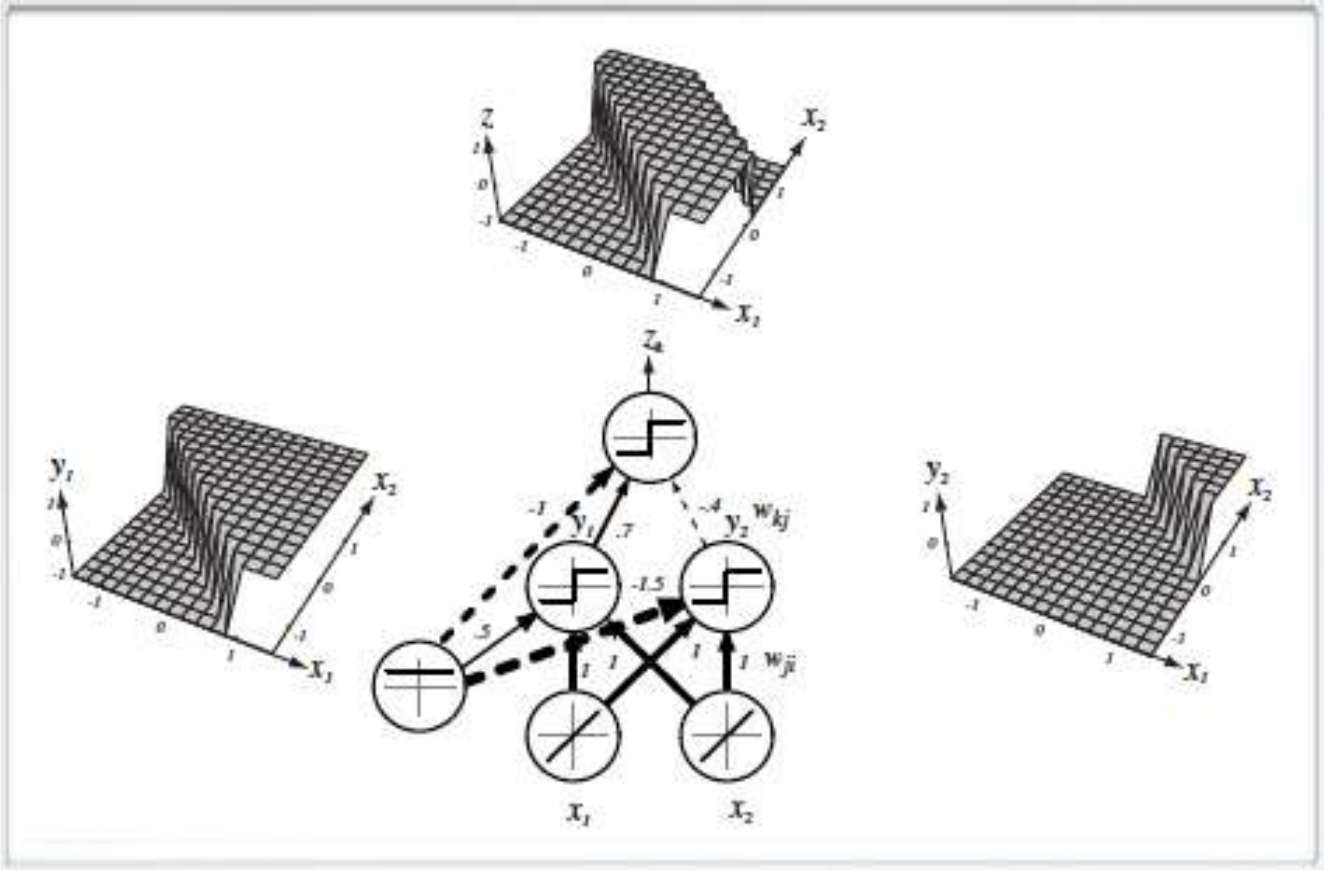


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

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

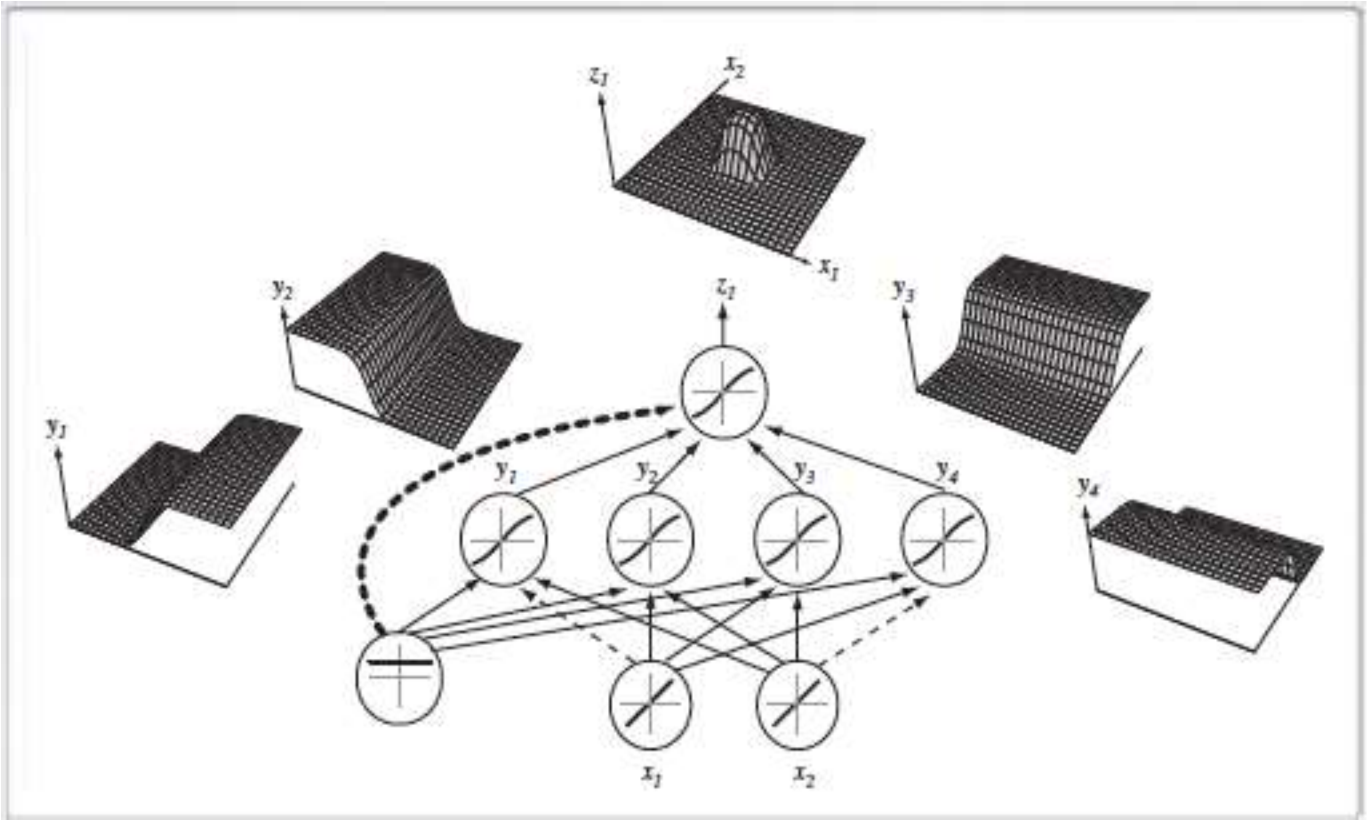
Capacity of neural network

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



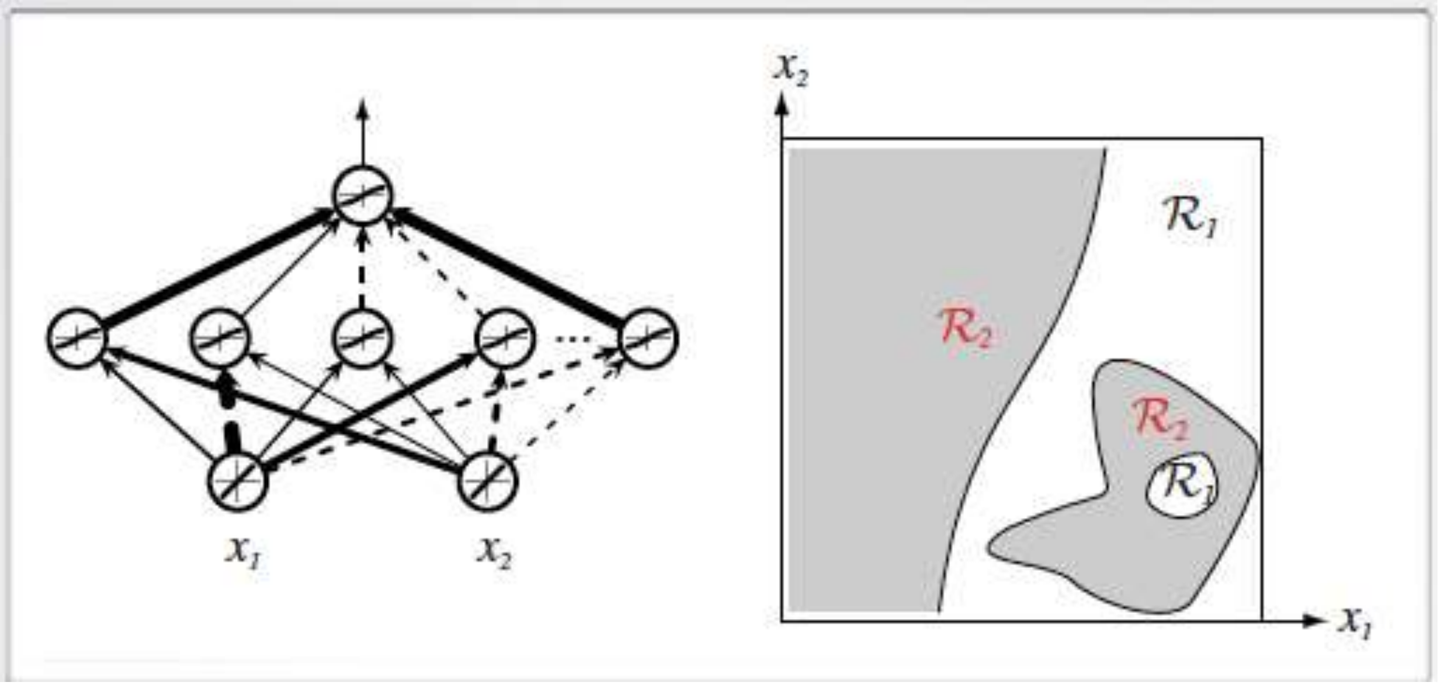
Capacity of neural network

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



Capacity of neural network

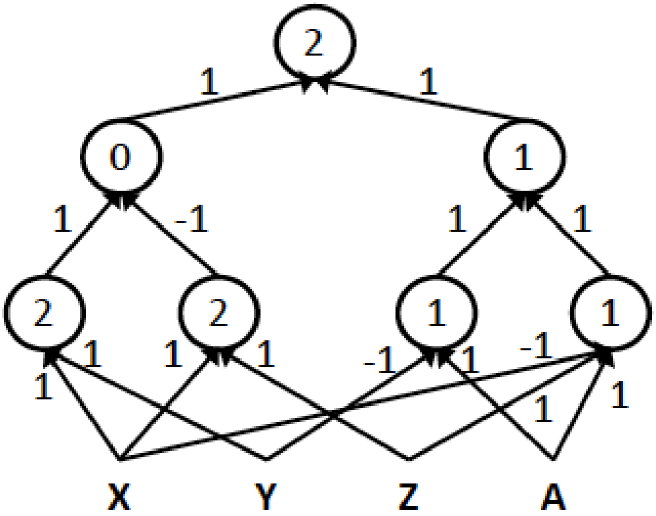
- Single hidden layer neural network



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

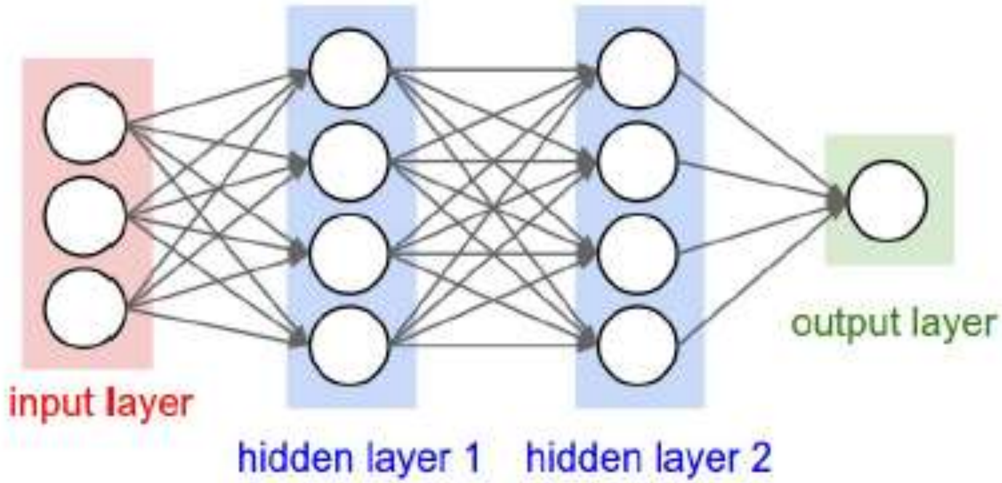
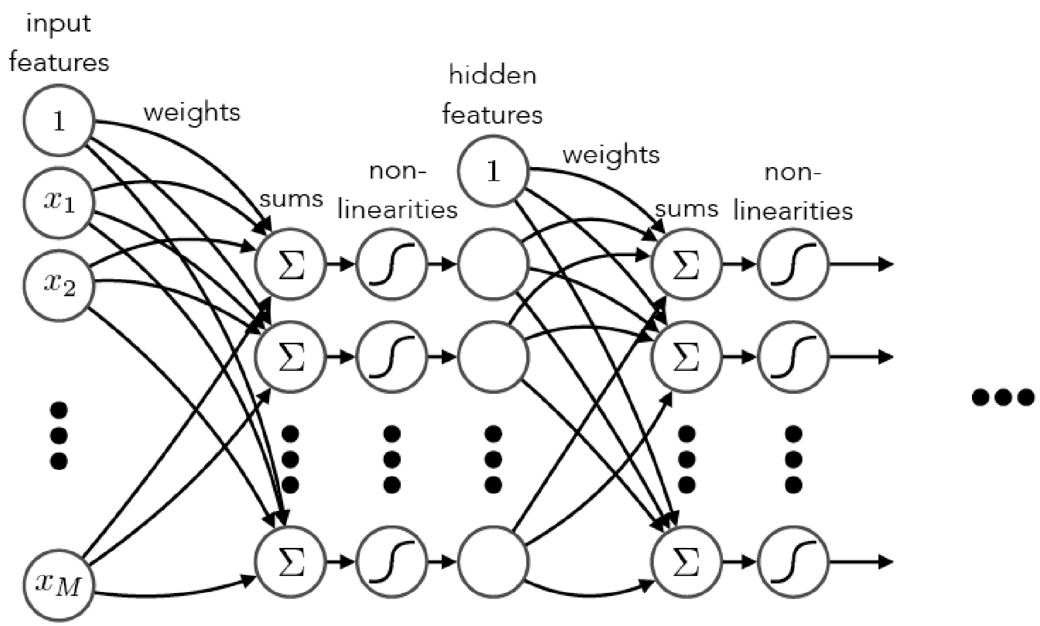


Figure : A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- 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

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

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

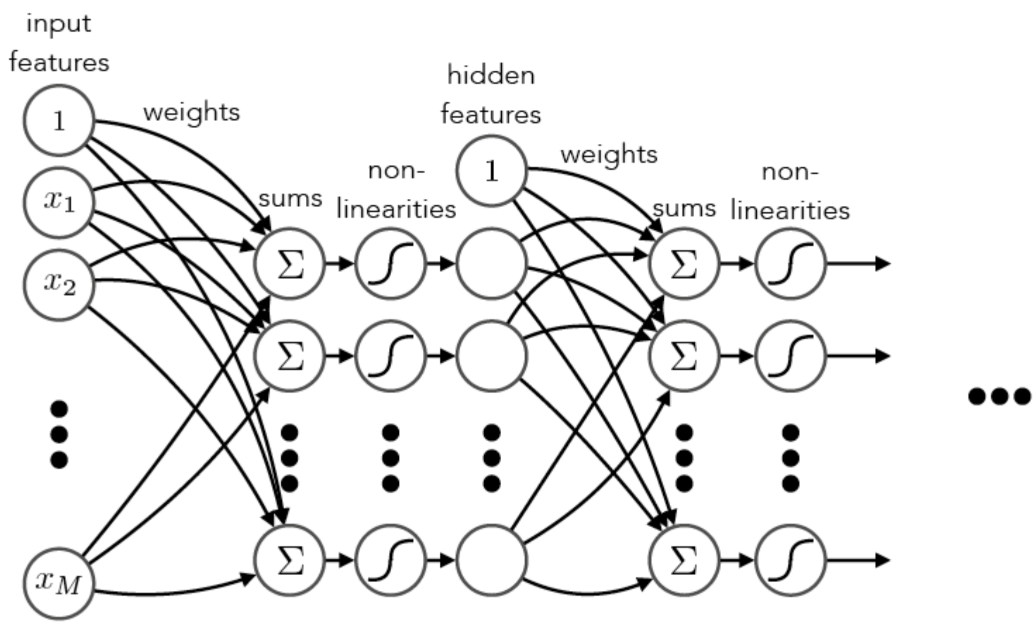
2nd layer

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

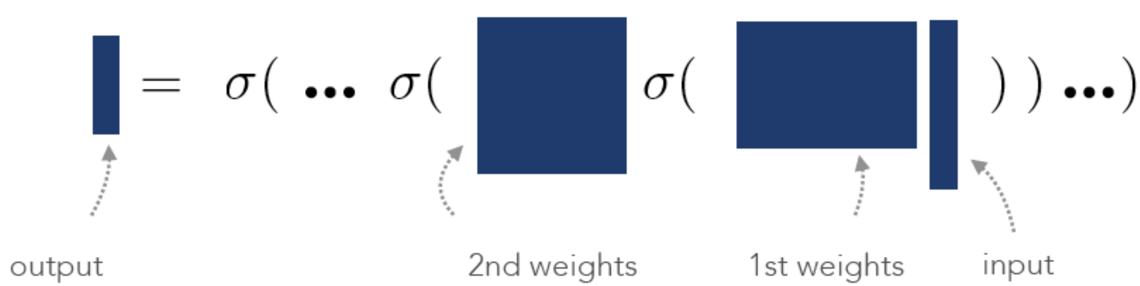
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

...

Multilayer networks

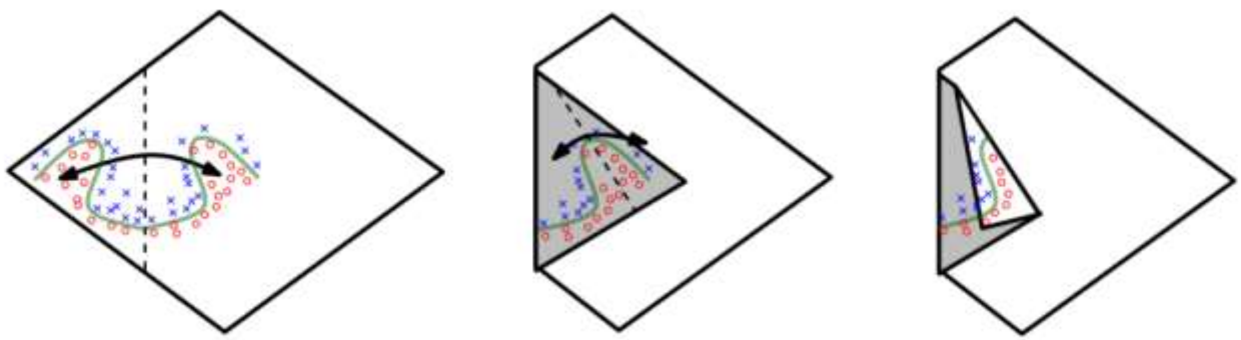


network: *sequence of parallelized weighted sums and non-linearities*



Why more layers (deeper)?

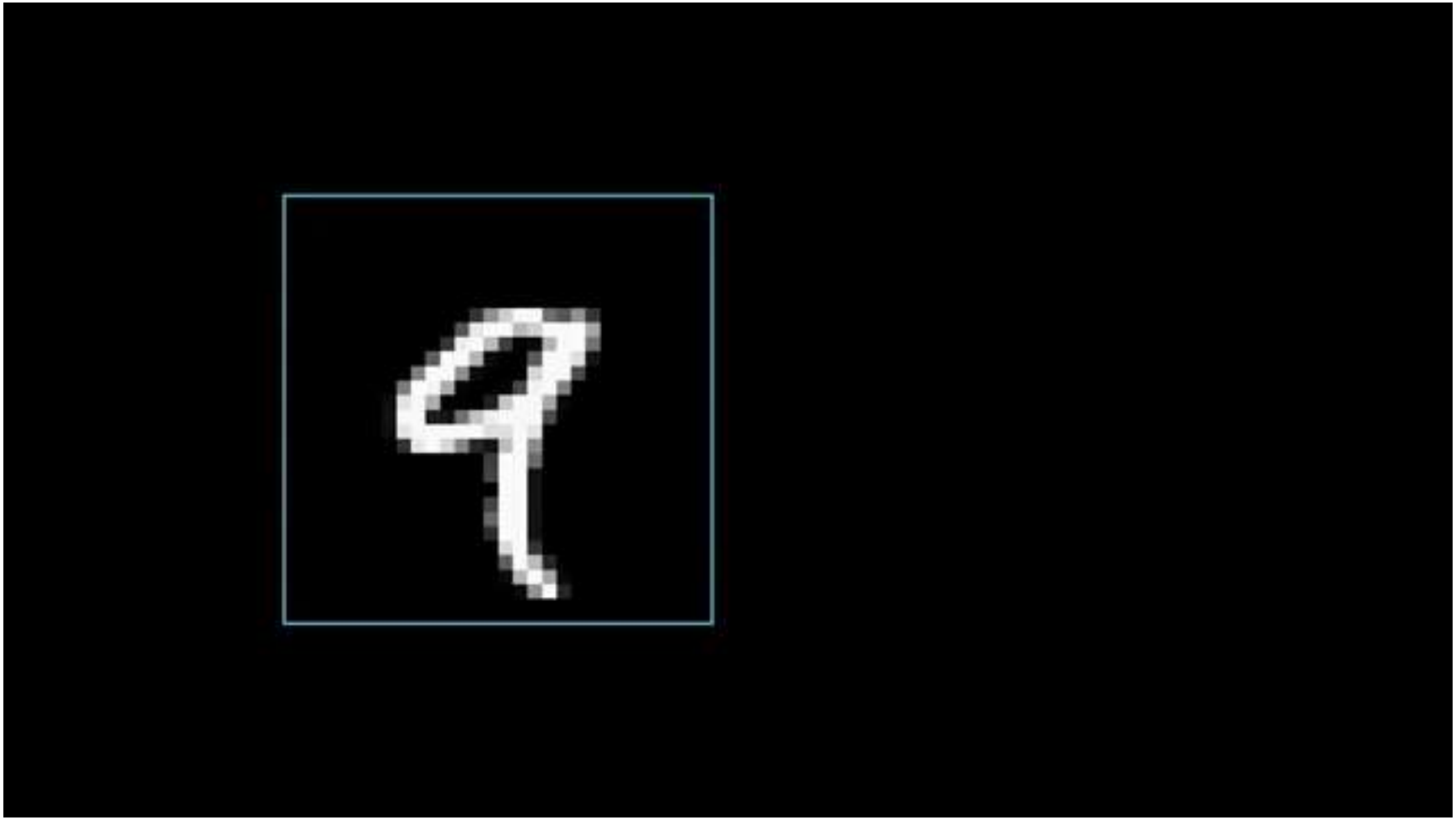
- A deep architecture can represent certain functions more compactly
  - (Montufar et al., NIPS'14)
    - Functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow one.



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.

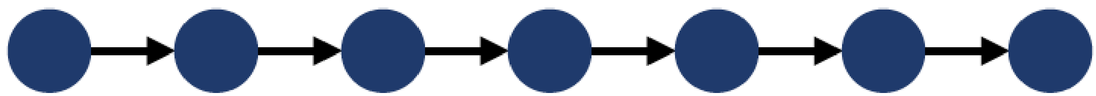
Why more layers (deeper)?



https://youtu.be/aircAruvnKk?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Other network connectivity

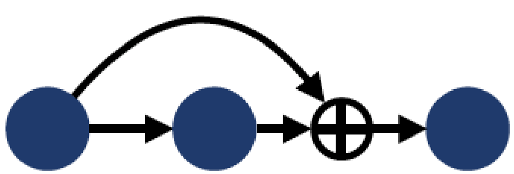
sequential connectivity: *information must flow through the entire sequence to reach the output*



information may not be able to propagate easily

→ *make shorter paths to output*

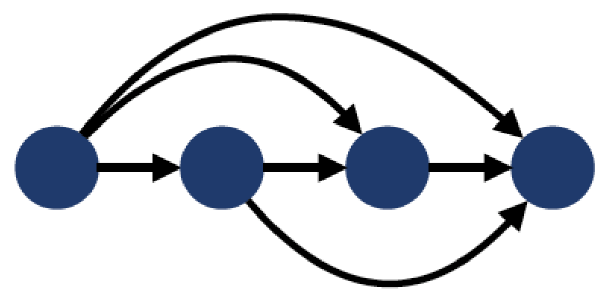
residual & highway connections



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

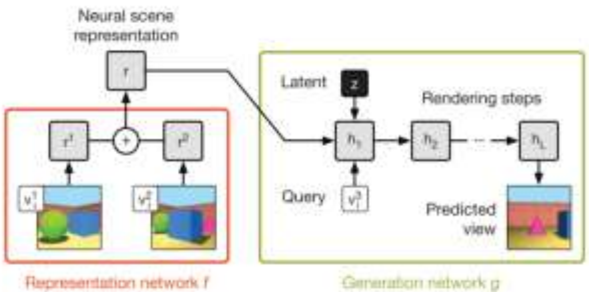
Highway networks, *Srivastava* et al., 2015

dense (concatenated) connections



Densely connected convolutional networks, Huang et al., 2017

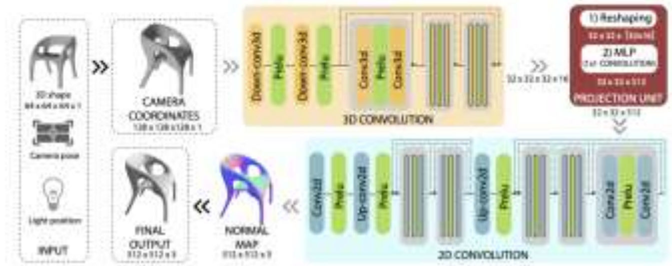
Modern MLP as Implicit Representation



Generative Query Networks
[Eslami et al. 2018]

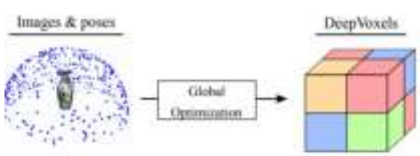
[Flynn et al., 2016; Zhou et al., 2018b;
Mildenhall et al. 2019]

Multiplane Images (MPIs)

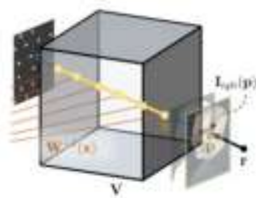


RenderNet [Nguyen-Phuoc et al. 2018]

Voxel Grids + CNN decoder

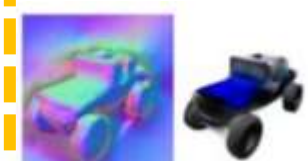


DeepVoxels
[Sitzmann et al. 2019]

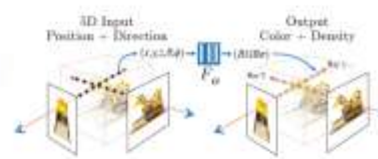


Neural Volumes
[Lombardi et al. 2019]

Voxel Grids + Ray Marching



SRN
[Sitzmann et al. 2019b]



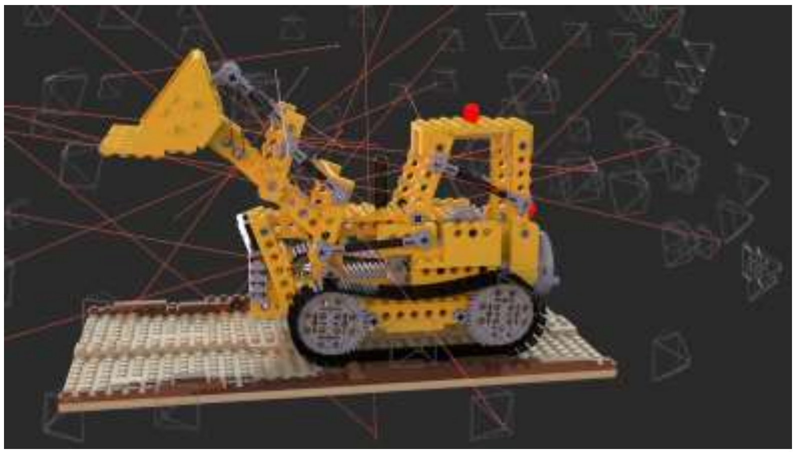
NeRF
[Mildenhall et al. 2020]

IDR
[Yariv et al. 2020]

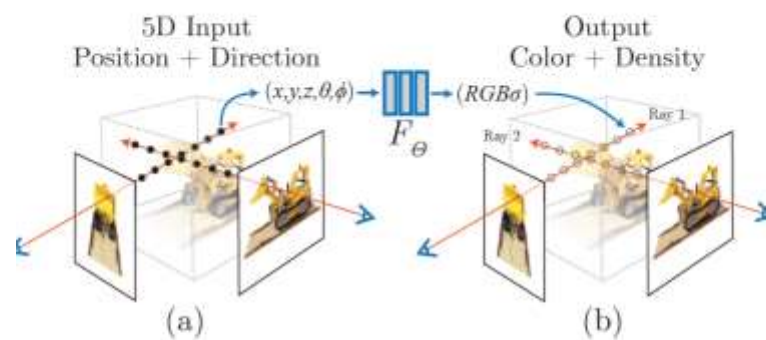
Implicit Fields

Modern MLP in NeRF

- - Color + Density
- Positional Encoding
- Volume Rendering



Representing Scenes as Neural Radiance Fields for View Synthesis, Mildenhall et al., *ECCV 2020 Oral - Best Paper Honorable Mention*



Modern MLP in NeRF

NeRF: Representing Scenes as
Neural Radiance Fields for View Synthesis

Ben Mildenhall* UC Berkeley
Pratul P. Srinivasan* UC Berkeley
Matthew Tancik* UC Berkeley
Jonathan T. Barron Google Research
Ravi Ramamoorthi UC San Diego
Ren Ng UC Berkeley

* Denotes Equal Contribution

https://youtu.be/JuH79E8rdKc

Outline

- Single layer neural networks
  - Network models; Example: Logistic Regression
- 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

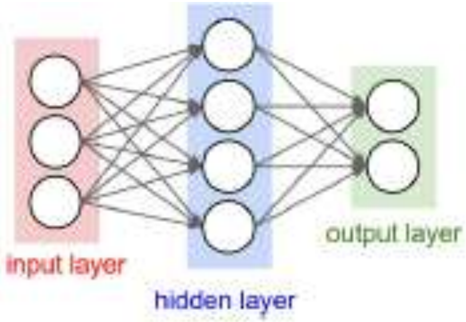
- We only need to know two algorithms
  - Inference/prediction: simply forward pass
  - Parameter learning: needs backward pass
- Basic fact:
  - A neural network is a function of composed operations

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

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

Inference example: Forward Pass

- What does the network compute?



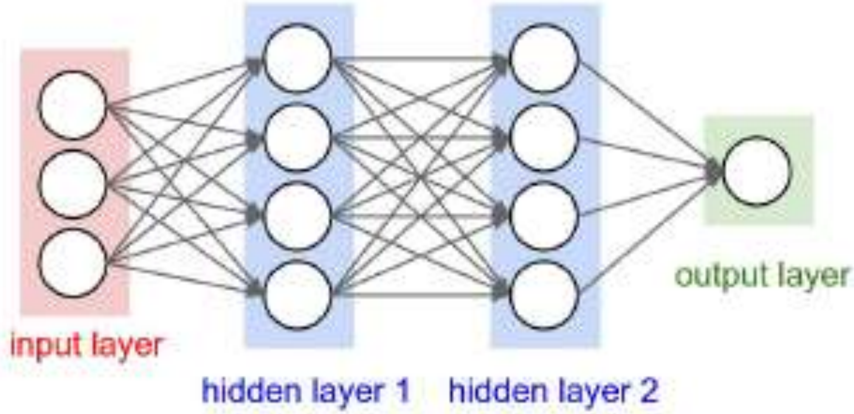
- Output of the network can be written as:

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

(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:

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

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

Backward pass

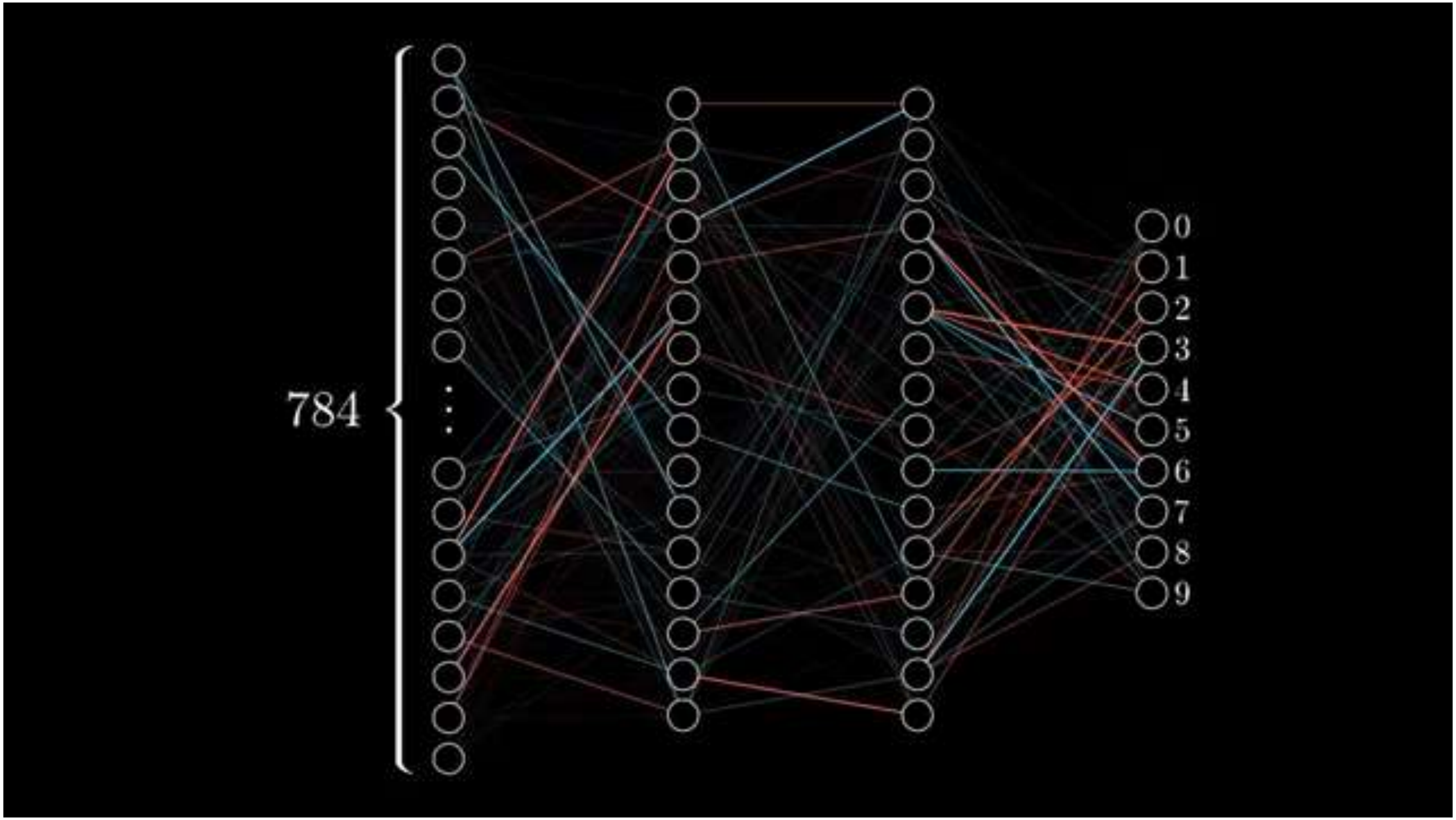
- Backpropagation
  - An efficient method for computing gradients in NNs
  - A neural network as a function of composed operations

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

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

→ use chain rule to calculate gradients

Backward pass



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

Gradient descent iteration

- Forward pass

1st layer

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

2nd layer

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

•••

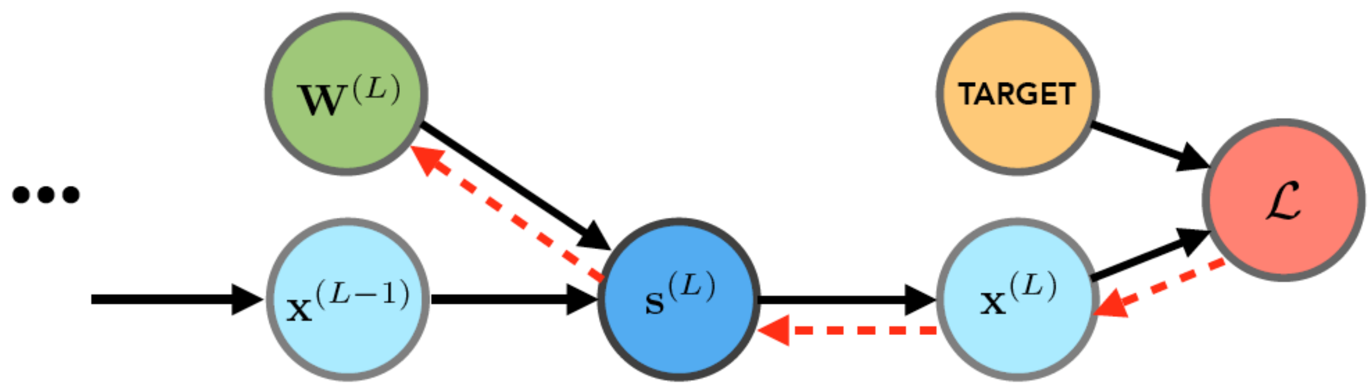
Loss

$$\mathcal{L}$$

- Backward pass

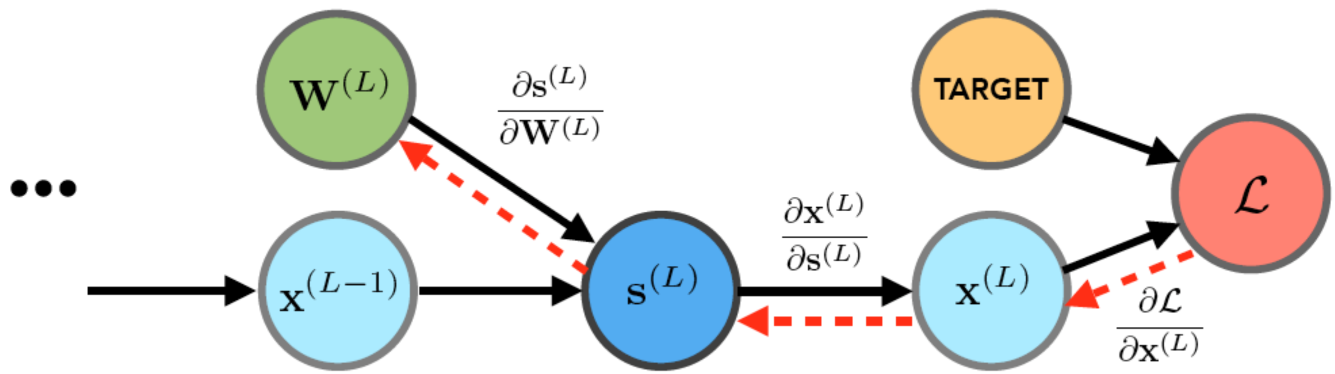
calculate $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$ let's start with the final layer: $\nabla_{W^{(L)}}\mathcal{L}$

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



Gradient descent iteration

- Backward pass



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

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

depends on the form of the loss

derivative of the non-linearity

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

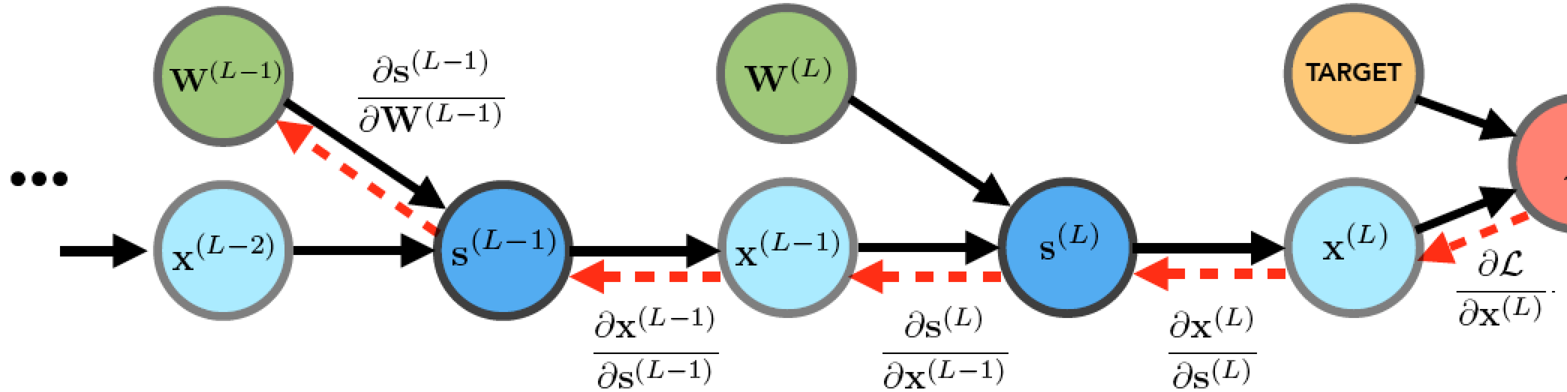
note $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

Gradient descent iteration

- Backward pass

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

again we'll draw the dependency graph:

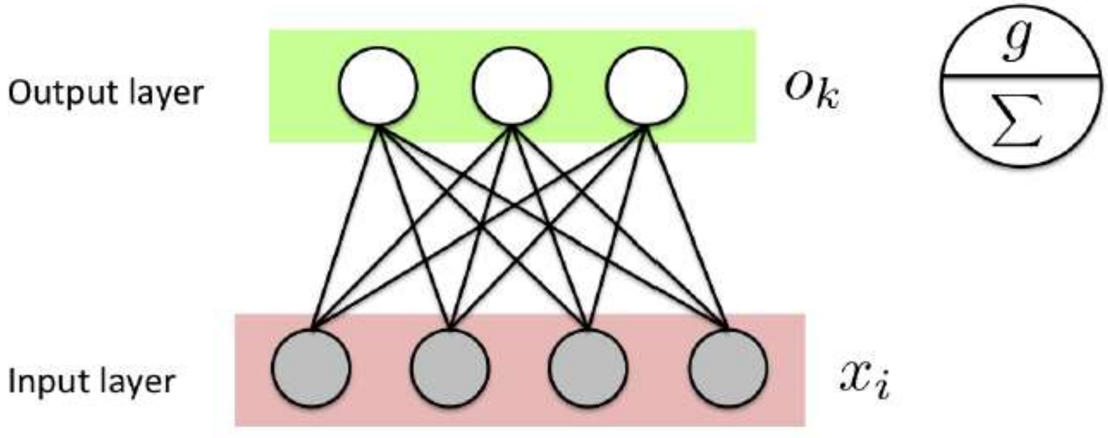


$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial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

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

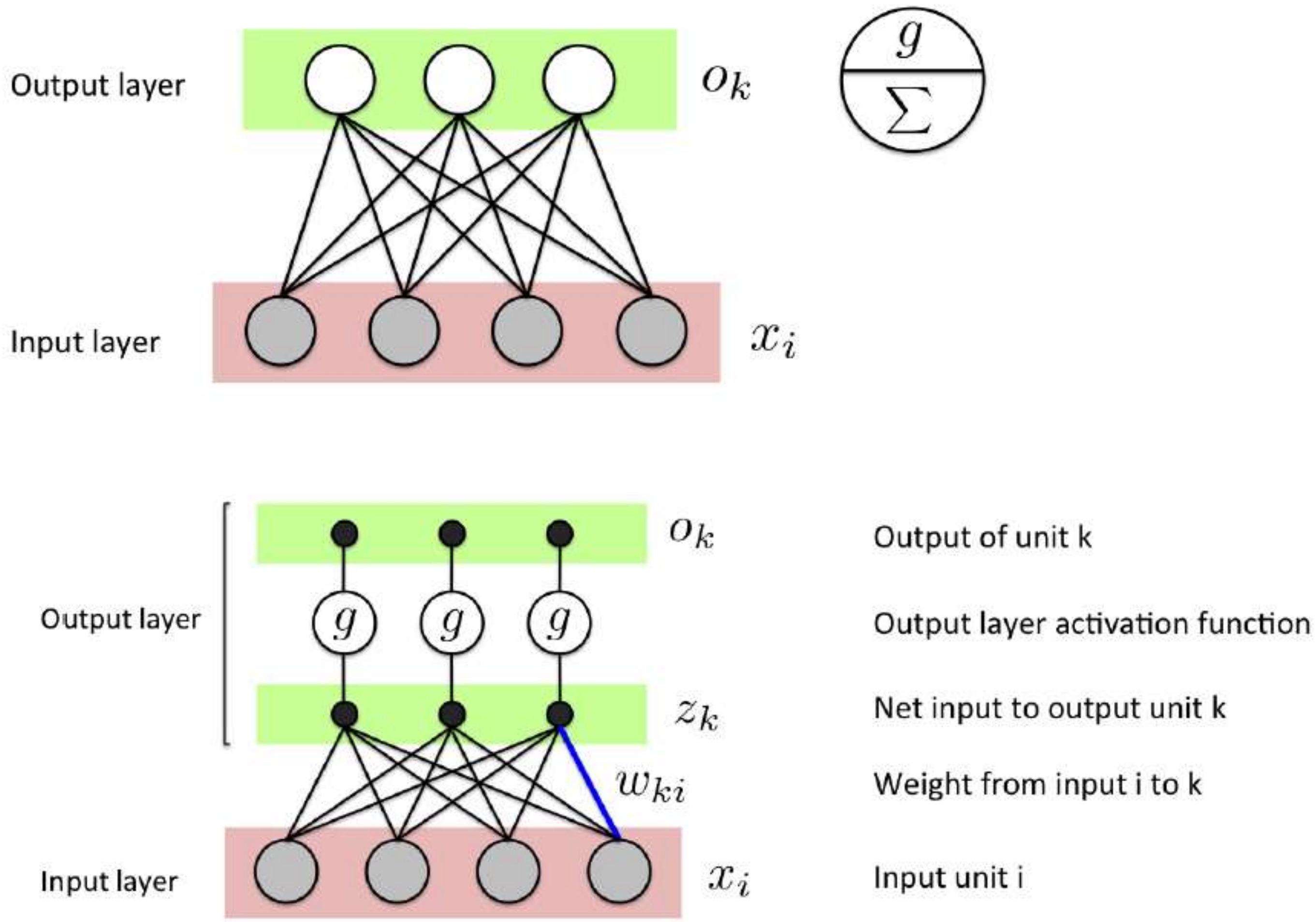
Example: Single Layer Network

- Let's take a single layer network

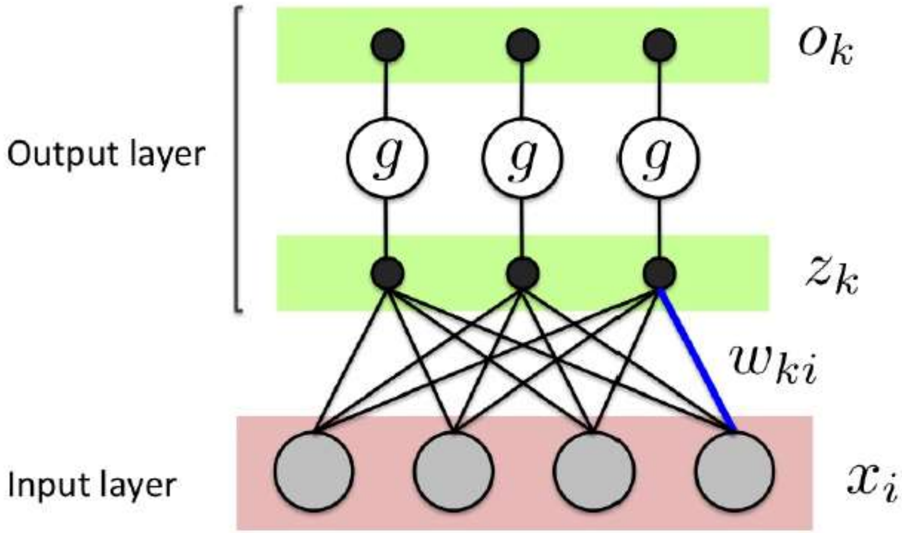


Example: Single Layer Network

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



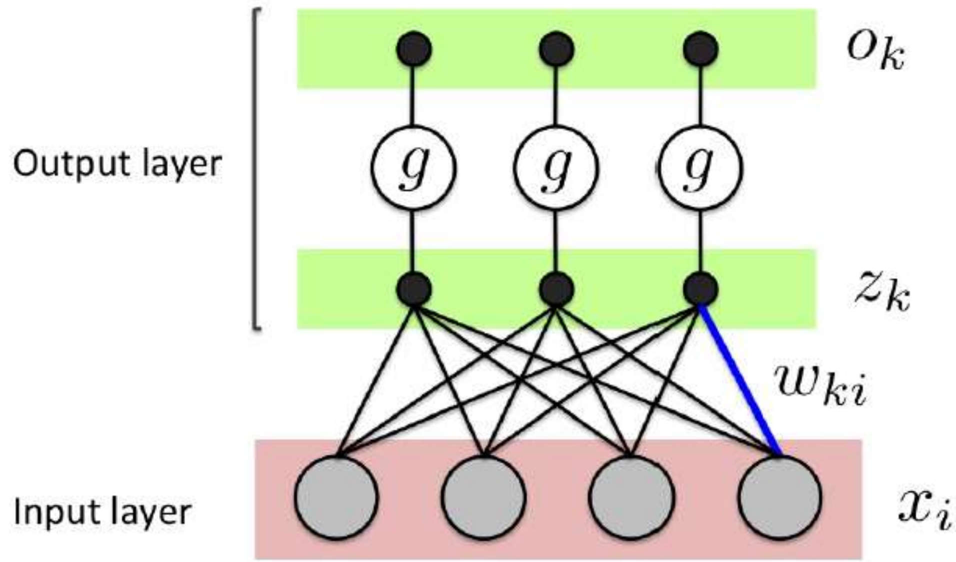
Example: Single Layer Network



- Error gradients for single layer network:

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

Example: Single Layer Network

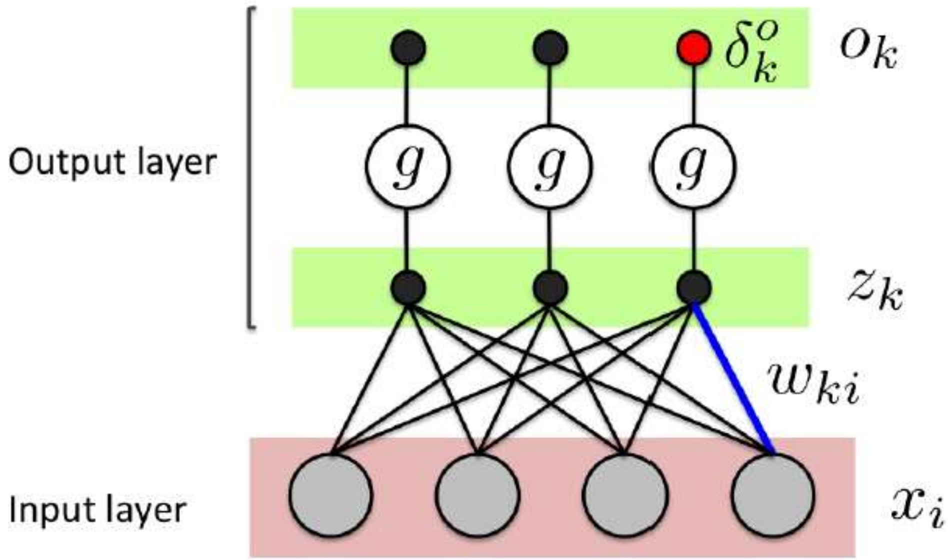


- Error gradients for single layer network:

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

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

Example: Single Layer Network

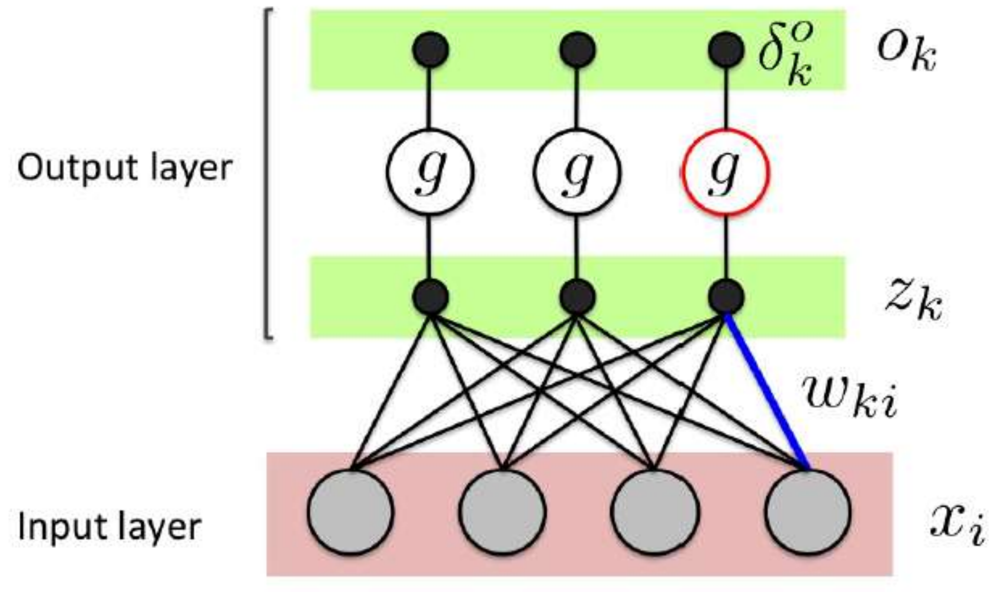


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- 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

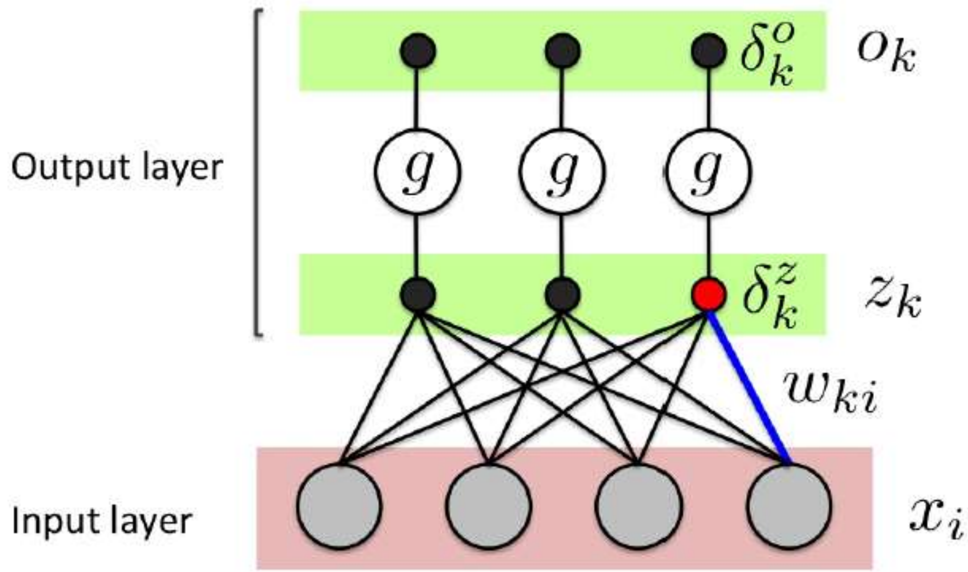


$$\frac{\partial o_k}{\partial z_k}$$

- 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

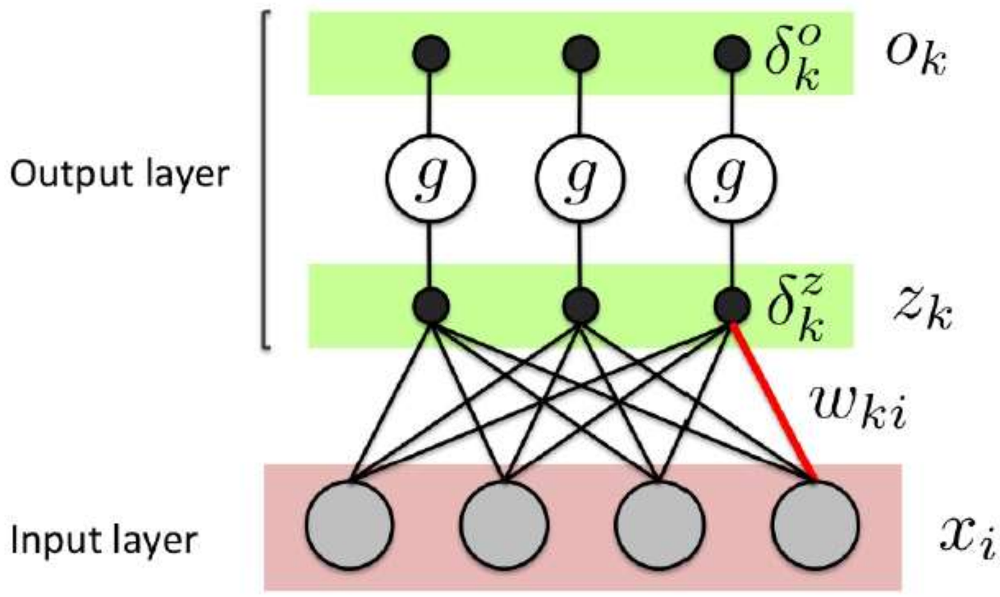


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- 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:

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

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

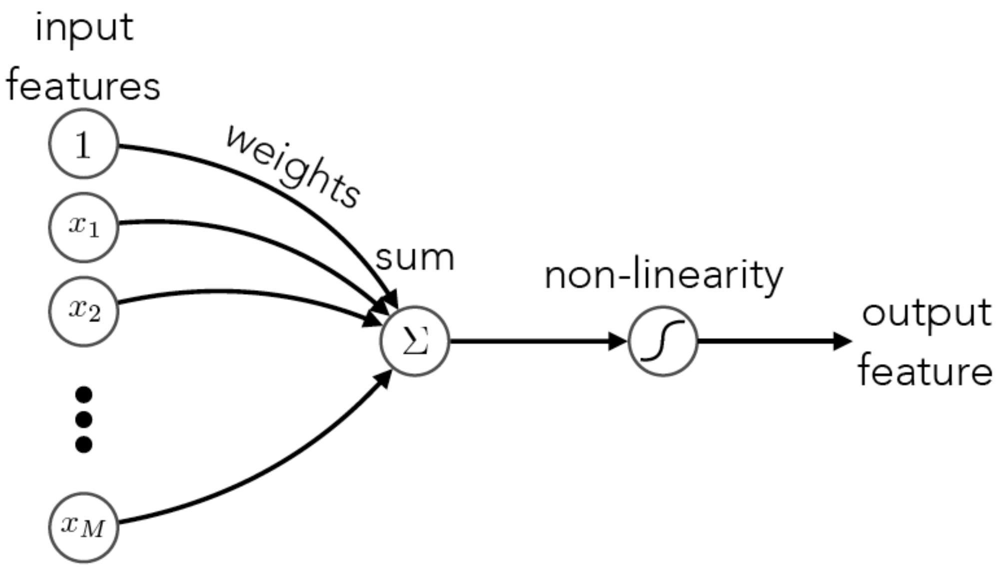
An implementation perspective

- 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:

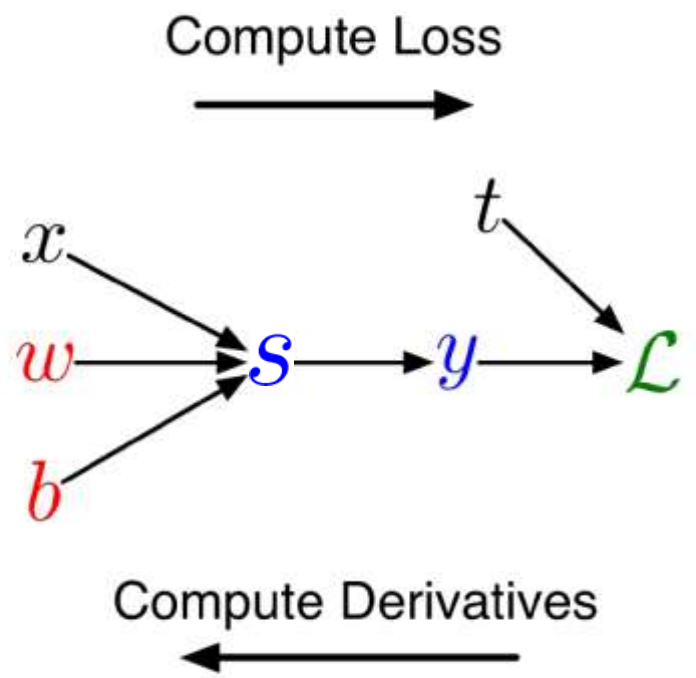
$$s = wx + b$$
$$y = \sigma(s)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

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

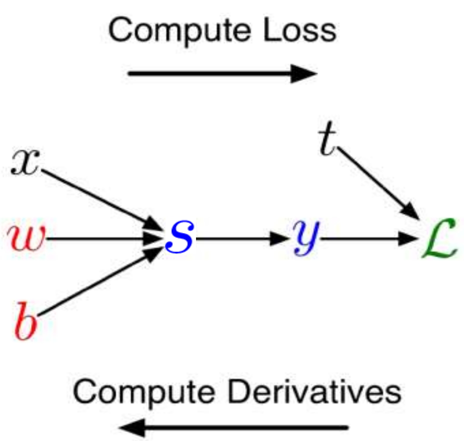


Univariate chain rule

- A shorthand notation
  - 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:

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



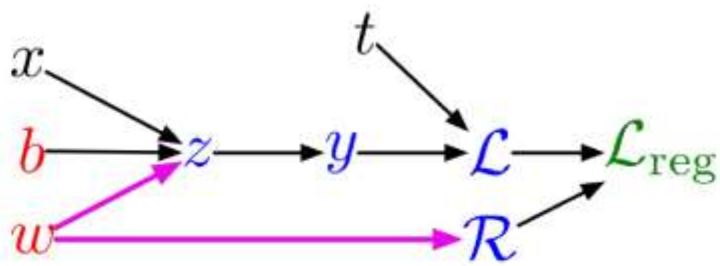
Computing the derivatives:

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

Multivariate chain rule

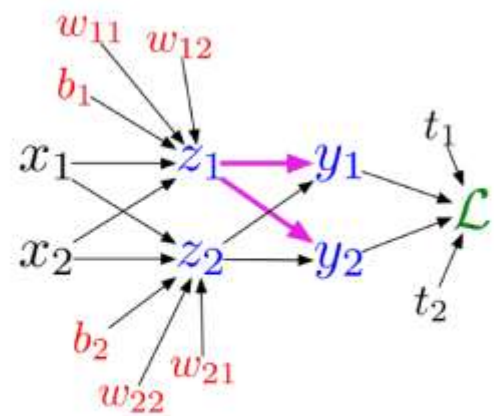
- The computation graph has fan-out > 1

L_2-Regularized regression



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

Multiclass logistic regression



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

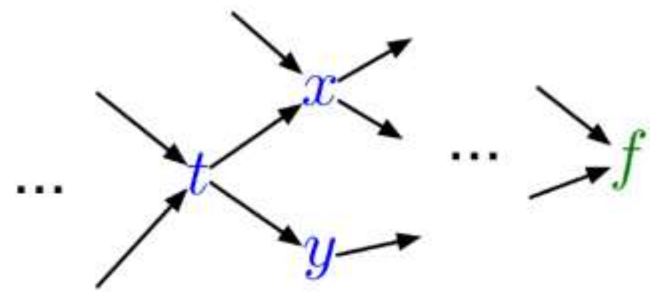
Multivariable chain rule

- 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

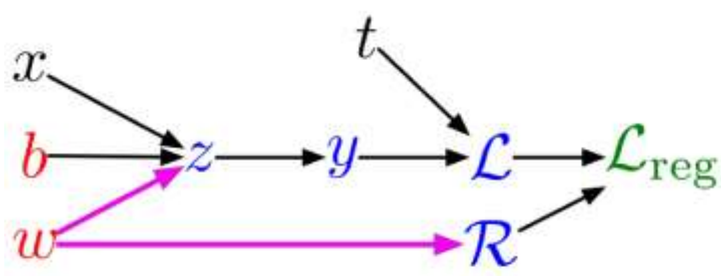
$\delta_{v_N} = 1$

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:

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

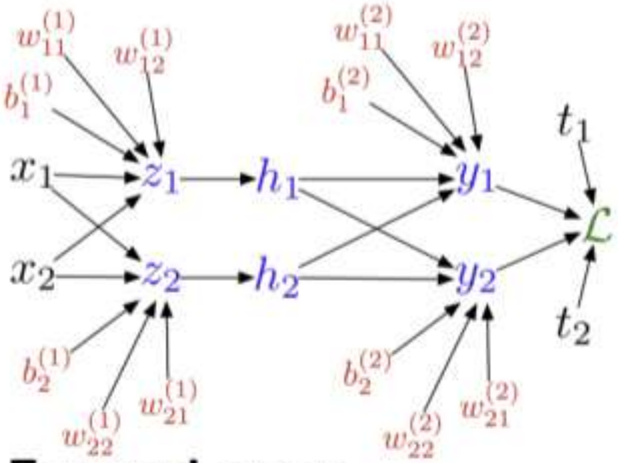
Backward pass:

$$\delta_{\mathcal{L}_{\mathrm{reg}}} =$$
$$\delta_{\mathcal{R}} =$$
$$=$$
$$\delta_{\mathcal{L}} =$$
$$=$$
$$\delta y =$$
$$=$$

$$\delta_z =$$
$$=$$
$$\delta_w =$$
$$=$$
$$\delta_b =$$
$$=$$

General Backpropagation

- Example: Multilayer Perceptron (multiple outputs)



Forward pass:

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i = \sigma(z_i)$$

$$y_k = \sum_i w_{ki}^{(2)} h_i + b_k^{(2)}$$

$$\mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y_k} = \overline{\mathcal{L}}\,(y_k - t_k)$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$

$$\overline{b_k^{(2)}} = \overline{y_k}$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

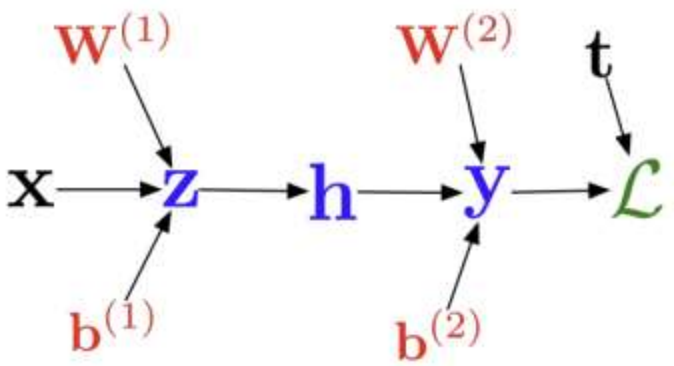
$$\overline{z_i} = \overline{h_i}\, \sigma'(z_i)$$

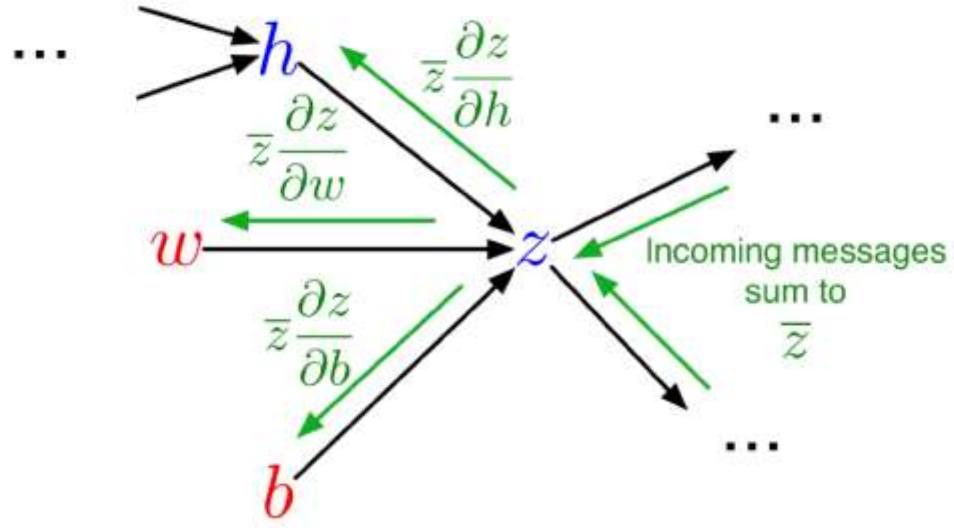
$$\overline{w_{ij}^{(1)}} = \overline{z_i}\, x_j$$

$$\overline{b_i^{(1)}} = \overline{z_i}$$

General Backpropagation

- Backprop as message passing:

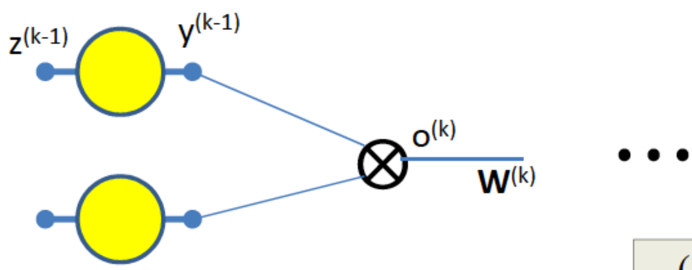




  - Each node receives a set of messages from its children, which are aggregated into its error signal, then it passes messages to its parents
  - Modularity: each node only has to know how to compute derivatives w.r.t. its arguments – local computation in the graph

Patterns in backward flow

- Multiplicative node

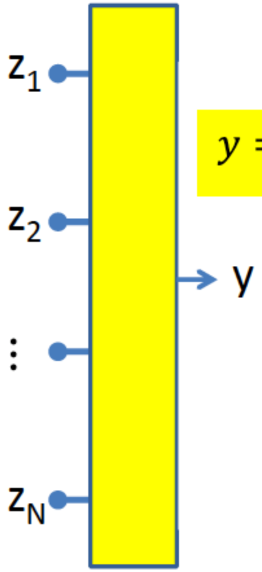


Forward: $\boxed{o_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)}}$

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

Patterns in backward flow

- Max node



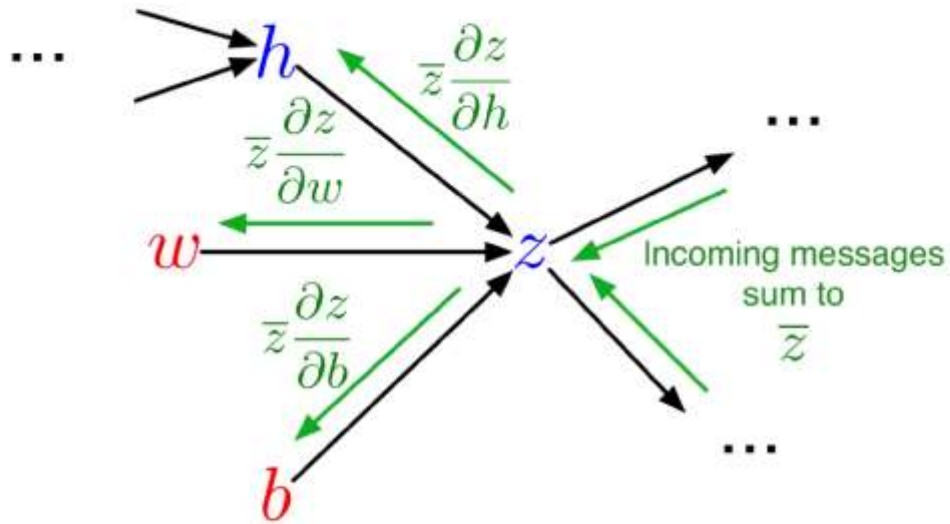
$$y = \max_j z_j$$

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

- 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



$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$\overline{w_{ki}^{(2)}} = \overline{y_k}\, h_i$$

$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

- 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-house-price.html

Summary

- Artificial neurons, Single-layer network
- Multi-layer neural networks
- Inference and learning
  - Forward and Backpropagation
- Next time …
  - Modern topics about MLP, CNN

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