



M2177.003100

Deep Learning

[3: Deep Feedforward Networks]

Electrical and Computer Engineering
Seoul National University

© 2018 Sungroh Yoon. this material is for educational uses only. some contents are based on the material provided by other paper/book authors and may be copyrighted by them.

(last compiled at 10:44:00 on 2018/09/12)

Outline

Introduction

Feedforward Networks

Deep Feedforward Networks

Summary

References

- *Deep Learning* by Goodfellow, Bengio and Courville [▶ Link](#)
 - ▶ Chapter 6
- online resources:
 - ▶ *Deep Learning Specialization (coursera)* [▶ Link](#)
 - ▶ *Stanford CS231n: CNN for Visual Recognition* [▶ Link](#)
 - ▶ *Machine Learning Yearning* [▶ Link](#)

Outline

Introduction

Feedforward Networks

Deep Feedforward Networks

Summary

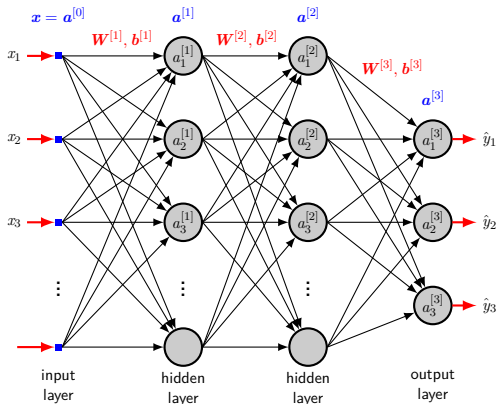
Deep feedforward net

- quintessential deep learning model
 - ▶ aka **feedforward neural net**, **multilayer perceptron** (MLP)
- goal: approximate some function f^*
 - e.g. a classifier: $y = f^*(x)$ maps input x to category y
- how it works: _____ + learn
 - ▶ define a mapping $y = f(x; \theta)$ and
 - ▶ learn parameters θ that give the best approximation
- extremely important
 - ▶ basis of many important commercial applications
 - e.g. convolutional nets, recurrent nets

Architecture

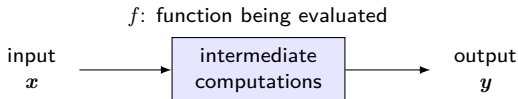
e.g. a feedforward neural net with two hidden layers

- ▶ parameters θ : weight W and bias b



- called _____ because

▶ information flows $x \rightarrow \boxed{f} \rightarrow y$



- no **feedback** connections

▶ no output is fed back into the model

c.f. recurrent neural nets (ch 10)

- called **networks** because

▶ represented by composing many different functions

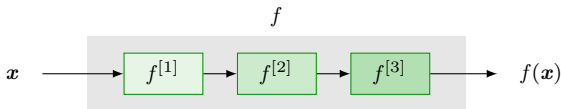
- associated with a directed acyclic graph
↑

describes how component functions are composed together

e.g. functions $f^{[1]}$, $f^{[2]}$, and $f^{[3]}$ connected in a chain:

$$f(\mathbf{x}) = f^{[3]}(f^{[2]}(f^{[1]}(\mathbf{x})))$$

- ▶ $f^{[l]}$: called **l -th layer**
- ▶ final layer: called **output layer**
- ▶ chain length \Rightarrow model _____ (“deep learning”)

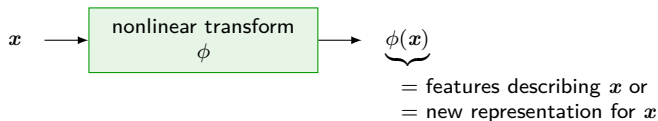


Training neural nets

- training: we drive $f(x)$ to match $f^*(x)$
- training data: noisy/approximate examples of $f^*(x)$
 - ▶ each example: $(\underbrace{x}_{\text{input}}, \underbrace{y}_{\text{label}})$ with $y \approx f^*(x)$
 - \Rightarrow directly specifies what **output layer** must do at each x
- behavior of the other layers: not directly specified by training data
 - ▶ these layers: called **layers**
 - ▶ features are distributed over hidden layers
- instead, the learning algorithm must decide
 - ▶ how to use hidden layers to best approximate f^*

Understanding feedforward nets

- begin with linear models and
 - ▶ consider how to overcome their limitations
- linear models:
 - 😊 efficient/reliable (closed form or convex)
 - 😞 cannot understand interaction between any two input variables
- to extend linear models to represent _____ functions of x
 - ▶ apply the linear model not to x itself but to transformed input $\phi(x)$



- ▶ equivalently: kernel trick (sec 5.7.2)

- how to choose ϕ ?

1. use a very **generic** ϕ (e.g. ∞ -dim ϕ as in RBF kernel)
 - ▷ “generic” but often poor “generalization”
2. **manually** engineer ϕ (e.g. traditional ML)
 - ▷ specialized but laborious
3. **learn** ϕ (e.g. deep learning)
 - ▷ we have a model

$$y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})^\top \mathbf{w}$$

- ▷ parameters $\boldsymbol{\theta}$: used to learn ϕ from a broad class of functions
 - ▷ parameters \mathbf{w} : map from $\phi(\mathbf{x})$ to desired output
- (deep) feedforward nets:
 - ▶ learn deterministic mappings from \mathbf{x} to \mathbf{y} (no feedback connections)
 - ▶ ϕ defines a _____ layer

To deploy a DNN

should make design decisions:

- just as linear model
 - ▶ choose optimizer/cost function/output units
 - ▶ gradient-based learning (sec 6.2)
- unique to feedforward nets
 - ▶ hidden layers \Rightarrow choosing activation functions (sec 6.3)
 - ▶ network architecture (sec 6.4)
 - ▷ how many layers
 - ▷ how to connect these layers
 - ▷ how many units in each layer
- training:
 - ▶ back-propagation and its modern generalizations (sec 6.5)

Outline

Introduction

Feedforward Networks

- Gradient-Based Learning
- Architecture
- Vectorized Representation

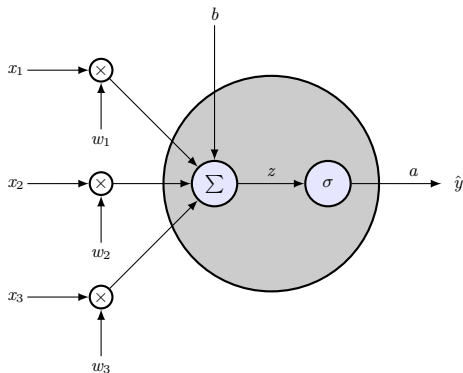
Hidden Units

Forward/Backward Functions

Deep Feedforward Networks

Summary

Recall: logistic regression as a neuron model

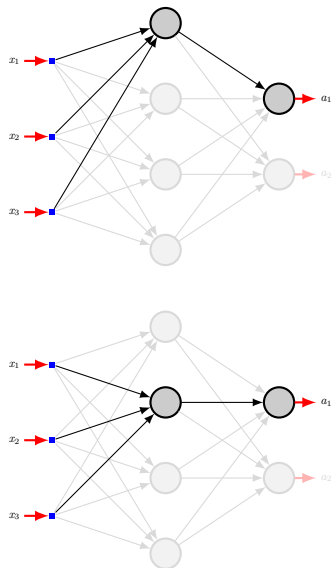
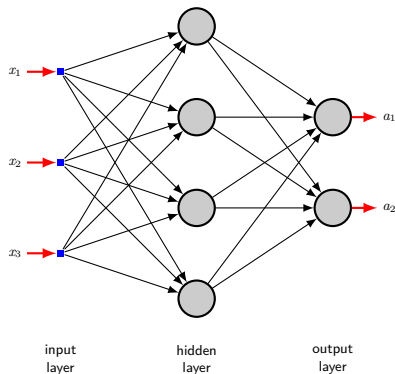


signal: $z = \mathbf{w}^\top \mathbf{x} + b$

activation: $a = \sigma(z)$

Feedforward neural net

- composed of logistic regression units

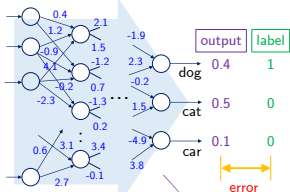


Concept of training & testing a neural net

TRAINING

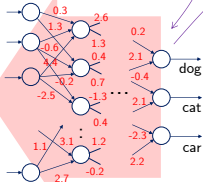


training image

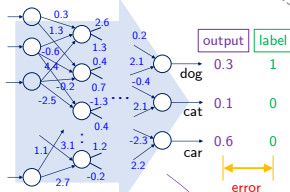


FORWARD propagation

BACKWARD propagation



training image



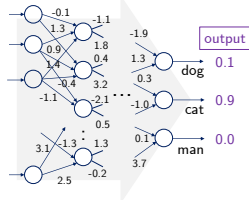
FORWARD propagation

(repeat)

TESTING



new image



Modern neural nets

- core ideas: no change since 80s
 - ▶ the same backprop/gradient descent: still in use
- recent improvement due to:
 - ▶ larger **data** sets \Rightarrow better generalization
 - ▶ larger **neural nets** \leftarrow better hw/sw infrastructure
 - ▶ better **algorithms**, in particular:

1. MSE \longrightarrow loss

2. sigmoid \longrightarrow

Outline

Introduction

Feedforward Networks

Gradient-Based Learning

Architecture

Vectorized Representation

Hidden Units

Forward/Backward Functions

Deep Feedforward Networks

Summary

Training a neural network

- nonlinearity of a neural net \Rightarrow non-convex loss function
 - ▶ largest difference from linear models
- neural nets: thus usually trained by
 - ▶ iterative, gradient-based optimizers (ch 8)
- sgd applied to non-convex loss functions
 - ▶ no convergence guarantee
 - ▶ sensitive to initial parameters
- feedforward neural nets
 - ▶ often initialize all weights to _____ values (sec 8.4)

Gradient-based learning

- gradient descent can train learning models
 - e.g.* linear regression and SVM
- computing gradient for a neural net: slightly more complicated
 - ▶ but can still be done efficiently by back-prop (sec 6.5)
- for gradient-based learning we must choose:
 1. ____ function
 2. model **output** representation

Cost function for neural nets

- total cost function
 - ▶ primary cost function + regularization term (ch 7)
- most modern neural nets: trained using maximum likelihood
 - i.e.* cost function = negative log-likelihood (NLL)
 - = cross-entropy between **training data** and **model distribution**

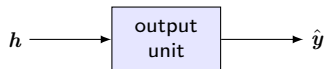
$$\begin{aligned} J(\theta) &= -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} | \mathbf{x}) \\ &= \frac{1}{m} \sum_{i=1}^m L(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) \\ (\text{for binary output}) &= -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right] \end{aligned}$$

- a recurring theme: gradient of cost function must be ____/predictable
 - ▶ NLL: more popular than MSE in this sense (see textbook)¹

¹ *e.g.* using \log undoes \exp of sigmoid/softmax

Output units

- suppose: a feedforward net provides hidden features $h = f(x; \theta)$
- output layer:
 - ▶ provides additional transformation from features to output



- ▶ most common: linear/sigmoid/_____ output units
- softmax² units: represent probability distribution over K classes
 - ▶ $\underbrace{\text{bernoulli}}_{\text{distribution}} : \underbrace{\text{sigmoid}}_{\text{function}} = \underbrace{\text{multinoulli}}_{\text{distribution}} : \text{softmax}$

²better name: "softargmax"

Multinoulli (or categorical) distribution

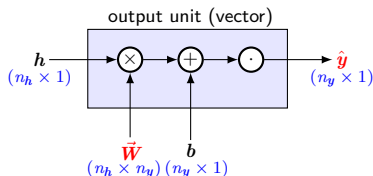
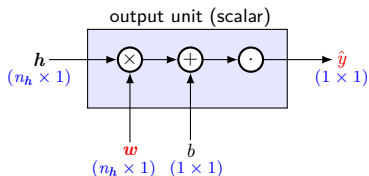
- a distribution over a single discrete variable with k finite states
 - ▶ parameterized by vector $\mathbf{p} \in [0, 1]^{k-1}$ (p_i : probability of i -th state)
 - ▶ $1 - \mathbf{1}^\top \mathbf{p}$: the final, k -th state's probability ($\mathbf{1}^\top \mathbf{p} \leq 1$)
- “_____”: recently coined term³
 - ▶ as a special case (*i.e.* **single** trial) of **multinomial** distribution
 - ▶ **multinomial** distribution: a distribution over vectors in $\{0, \dots, n\}^k$
 - ▷ represents how many times each of k categories is visited when n samples are drawn from a **multinoulli** distribution

distribution	# classes	# trials (samples)
Bernoulli	2	1
multinoulli	k	1
binomial	2	n
multinomial	k	n

³many texts use “multinomial” to refer to multinoulli without clarifying they refer only to $n = 1$ case

Types of output units

type	output	formula	output distribution
linear	vector	$\hat{\mathbf{y}} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$	Gaussian
sigmoid	scalar	$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h} + b)$	Bernoulli
softmax	vector	$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}^\top \mathbf{h} + \mathbf{b})$	multinoulli



$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Outline

Introduction

Feedforward Networks

Gradient-Based Learning

Architecture

Vectorized Representation

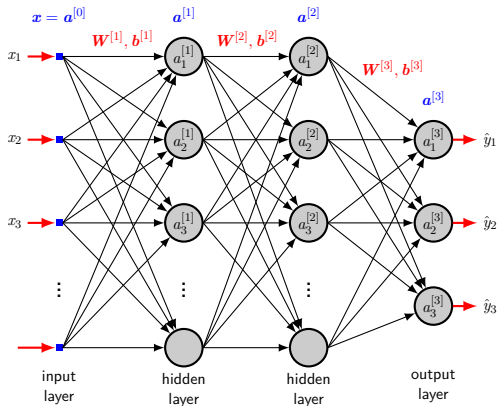
Hidden Units

Forward/Backward Functions

Deep Feedforward Networks

Summary

Notation



- notes:

- ▶ J : cost function
- ▶ \star and $d\star = \frac{\partial J}{\partial \star}$ have the ____ size

- layer/node indices

$a_j^{[l]}$	\leftarrow layer
	\leftarrow node index

- parameters

- ▶ weight: $W^{[l]}$
- ▶ bias: $b^{[l]}$

- gradient:

$d\star \triangleq \frac{\partial J}{\partial \star}$

e.g.

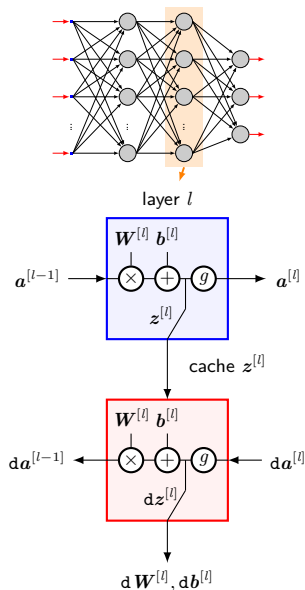
$$dz = \frac{\partial J}{\partial z}$$

$$da = \frac{\partial J}{\partial a}$$

$$dW = \frac{\partial J}{\partial W}$$

$$db = \frac{\partial J}{\partial b}$$

Operations for each layer



layer l

- ▶ parameters: $W^{[l]}, b^{[l]}$
- ▶ _____ function: $g^{[l]}$

• forward function

- ▶ input: $a^{[l-1]}$
- ▶ output: $a^{[l]} = g^{[l]}(z^{[l]})$
- ▶ cache: $z^{[l]}$

• backward function

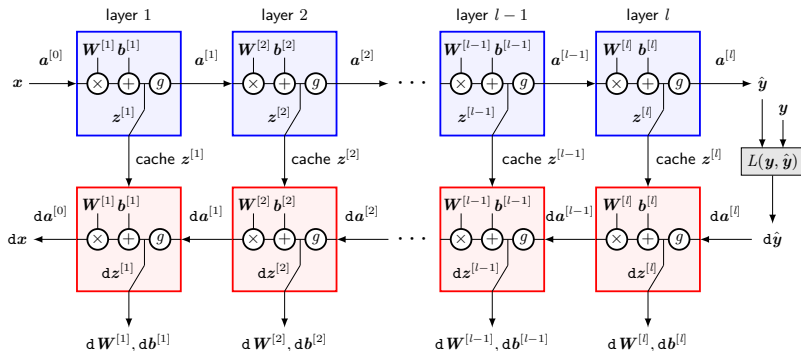
- ▶ input: $da^{[l]}$, cached $z^{[l]}$
- ▶ output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

• parameter update

$$W^{[l]} \leftarrow W^{[l]} - \epsilon da^{[l-1]} a^{[l-1]T}$$

$$b^{[l]} \leftarrow b^{[l]} - \epsilon da^{[l]}$$

Overall architecture



parameter update (ϵ : learning rate)

$$W^{[l]} \leftarrow W^{[l]} - \epsilon \frac{\partial L}{\partial W^{[l]}}$$

$$b^{[l]} \leftarrow b^{[l]} - \epsilon \frac{\partial L}{\partial b^{[l]}}$$

Outline

Introduction

Feedforward Networks

Gradient-Based Learning

Architecture

Vectorized Representation

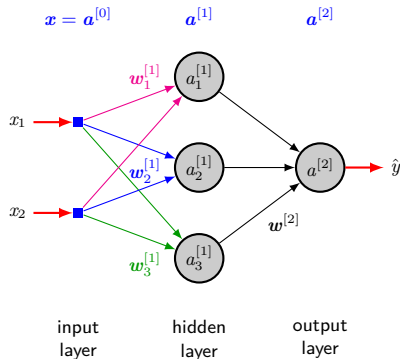
Hidden Units

Forward/Backward Functions

Deep Feedforward Networks

Summary

A running example



- hidden layer

$$a_1^{[1]} = g(\mathbf{w}_1^{[1]\top} \mathbf{x} + b_1^{[1]})$$

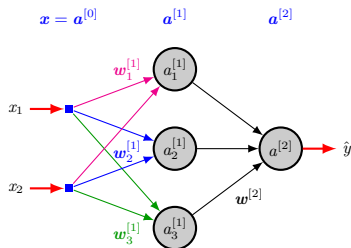
$$a_2^{[1]} = g(\mathbf{w}_2^{[1]\top} \mathbf{x} + b_2^{[1]})$$

$$a_3^{[1]} = g(\mathbf{w}_3^{[1]\top} \mathbf{x} + b_3^{[1]})$$

- output layer

$$a^{[2]} = g(\mathbf{w}^{[2]\top} \mathbf{a}^{[1]} + b^{[2]})$$

Vectorized representation



- separate equations

$$a_1^{[1]} = g(\mathbf{w}_1^{[1]\top} \mathbf{x} + b_1^{[1]}) = g(z_1^{[1]})$$

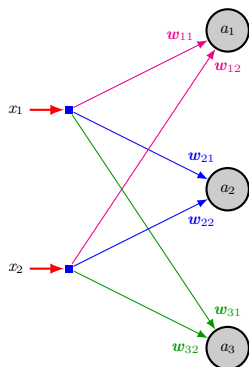
$$a_2^{[1]} = g(\mathbf{w}_2^{[1]\top} \mathbf{x} + b_2^{[1]}) = g(z_2^{[1]})$$

$$a_3^{[1]} = g(\mathbf{w}_3^{[1]\top} \mathbf{x} + b_3^{[1]}) = g(z_3^{[1]})$$

- vectorized equations

$$\mathbf{z}^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \underbrace{\begin{bmatrix} \text{---} & \mathbf{w}_1^{[1]\top} & \text{---} \\ \text{---} & \mathbf{w}_2^{[1]\top} & \text{---} \\ \text{---} & \mathbf{w}_3^{[1]\top} & \text{---} \end{bmatrix}}_{\text{matrix? TWO choices}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}, \quad \mathbf{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} = g(\mathbf{z}^{[1]})$$

Weight matrix conventions



- RL (right-left) convention

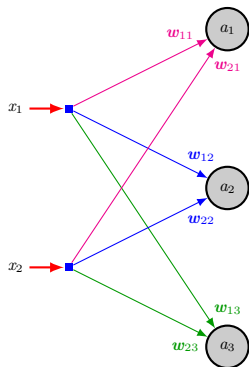
► weight for $i \rightarrow j$: w_{ji}

$$\tilde{W} = \begin{bmatrix} \textcolor{pink}{w}_{11} & \textcolor{pink}{w}_{12} \\ \textcolor{blue}{w}_{21} & \textcolor{blue}{w}_{22} \\ \textcolor{green}{w}_{31} & \textcolor{green}{w}_{32} \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$

- then

$$\begin{bmatrix} \textcolor{pink}{w}_1^\top \\ \textcolor{blue}{w}_2^\top \\ \textcolor{green}{w}_3^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \textcolor{pink}{w}_{11} & \textcolor{pink}{w}_{12} \\ \textcolor{blue}{w}_{21} & \textcolor{blue}{w}_{22} \\ \textcolor{green}{w}_{31} & \textcolor{green}{w}_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

=



- LR (left-right) convention

- ▶ weight for $i \rightarrow j : w_{ij}$

$$\vec{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \in \mathbb{R}^{2 \times 3}$$

- then

$$\begin{bmatrix} \text{---} & w_1^\top & \text{---} \\ \text{---} & w_2^\top & \text{---} \\ \text{---} & w_3^\top & \text{---} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

=

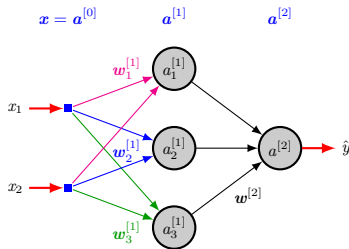
Vectorized representation

- two flavors

$$\mathbf{z}^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \begin{bmatrix} \text{---} & \textcolor{red}{w}_1^{[1]\top} & \text{---} \\ \text{---} & \textcolor{blue}{w}_2^{[1]\top} & \text{---} \\ \text{---} & \textcolor{green}{w}_3^{[1]\top} & \text{---} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}$$

$$= \begin{cases} \vec{W}^\top \mathbf{x} + \mathbf{b} & \boxed{\text{LR convention}} \\ \tilde{W} \mathbf{x} + \mathbf{b} & \boxed{\text{RL convention}} \end{cases}$$

$$\mathbf{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \end{bmatrix} = g\left(\mathbf{z}^{[1]}\right)$$



Outline

Introduction

Feedforward Networks

- Gradient-Based Learning
- Architecture
- Vectorized Representation

Hidden Units

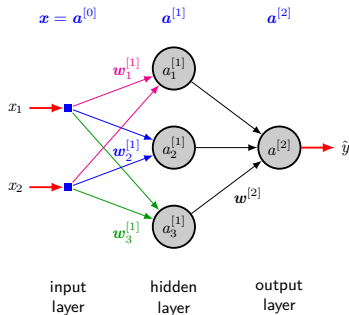
Forward/Backward Functions

Deep Feedforward Networks

Summary

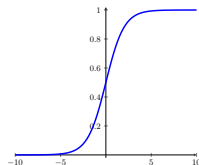
Hidden units

- what they do:
 1. accept a vector of inputs x
 2. compute an affine transformation $z = \mathbf{W}^\top x + b$
 3. apply an element-wise nonlinear function g to z
 4. return _____ $a = g(z)$



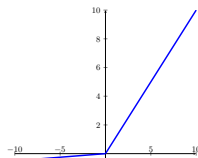
- hidden units differ only in activation function $g(\mathbf{z})$
- _____ linear units (ReLU): excellent default choice
 - ▶ non-differentiability: can be disregarded in practice
 - ▶ many other types also available
- hidden unit design remains an active area of research
 - e.g. $g(\mathbf{z}) = \cos(\mathbf{z})$ gives $< 1\%$ error on MNIST
 - ▶ new types: published only if clearly show significant improvement
- notation
 - ▶ $g^{[l]}$: activation function for layer l
 - ▶ mixing activation function types in a layer: uncommon

Activation functions

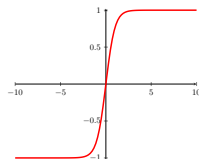


- sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

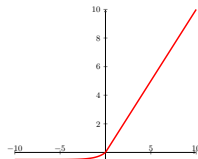


- $\max\{0.01z, z\}$

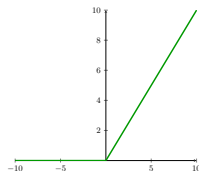


- tanh

$$\tanh(z)$$

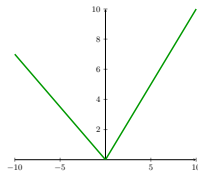


- ELU
- $$\begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$$



- ReLU

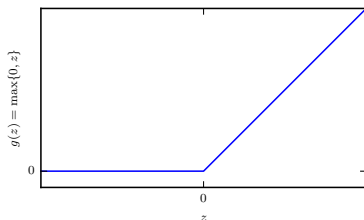
$$\max\{0, z\}$$



- $\max\{z_1, z_2\}$

Rectified linear units (ReLU)

- activation function: $g(z) = \max\{0, z\}$
- pros
 - ▶ no saturation in (+) region
 - ▶ computationally very efficient
 - ▶ converges faster than sigmoid
 - ▶ biologically more plausible than sigmoid
- cons
 - ▶ not zero-centered output
 - ▶ ____ gradient in (−) region



ReLU Initialization

- ReLU:

- ▶ typically used on top of an affine transformation:

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) \quad (1)$$

- good practice:

- ▶ set all elements of \mathbf{b} to a _____ number (*e.g.* 0.1 or 0.01)
- ⇒ ReLU initially active for most inputs in training set
- ⇒ derivatives can pass through

ReLU optimization

- easy to optimize (\because so similar to linear units)
 - ▶ half zero, half linear
 - derivatives through ReLU
 - ▶ remain large whenever the unit is active
 - ▶ not only large but also _____
 - ▷ derivative: 1 everywhere unit is active
 - ▷ second derivative: 0 almost everywhere
- \Rightarrow gradient direction is far more useful for learning
- ▶ than activation functions with second-order effects

ReLU Generalization

- overcome ReLU limitation (zero gradient in $(-)$ region)

- ▶ guaranteed to receive gradient everywhere

1. absolute value rectification: $g(z) = |z|$

2. leaky ReLU: $g(z) = \max\{\alpha z, z\}$ (fixed α)

3. parametric ReLU: $g(z) = \max\{\alpha z, z\}$ (_____ α)

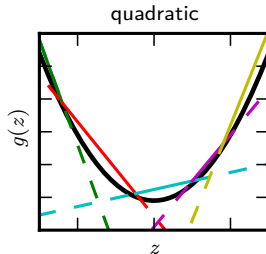
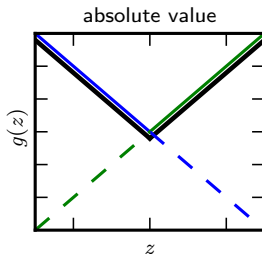
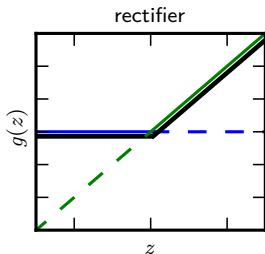
4. exponential ReLU:

$$g(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$$

- ▶ closer to zero mean outputs
- ▶ more robust to noise than leaky ReLU

Further generalization: maxout units

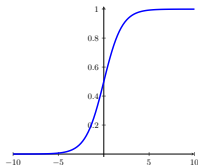
- *learn the activation function itself*
 - ▶ learn a _____ linear, convex function with up to k pieces
 - ▶ approximate any convex function with arbitrary fidelity (with large k)
- cons: more parameters/neurons required



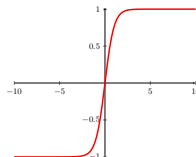
Prior to ReLU

- popular: sigmoid activations

- ▶ logistic sigmoid: $g(z) = \sigma(z)$



- ▶ tanh: $g(z) = \tanh(z)$

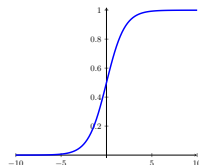


- ▶ closely related: $\tanh(z) = 2\sigma(2z) - 1$

- tanh: typically performs better than logistic sigmoid
 - ▶ zero centered (but still kills gradient when saturated)
 - ▶ resembles $y = x$ more closely at (near) zero \Rightarrow easier training
 - ▶ use ____ when a sigmoidal activation function must be used

Logistic sigmoid activation

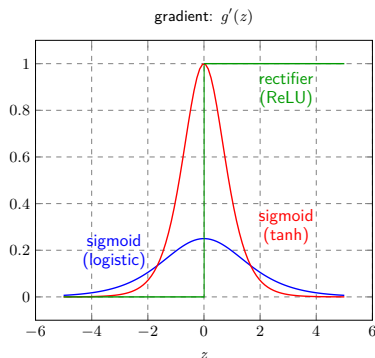
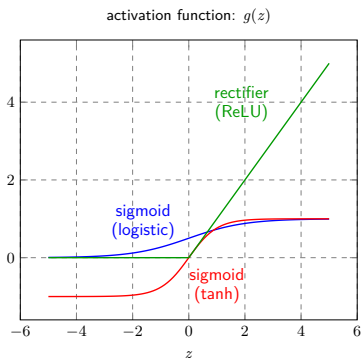
- historically popular
 - ▶ outputs to range $[0, 1]$
 - ▶ nice interpretation
 - ▷ saturating “firing rate” of a neuron



- cons
 - ▶ saturation \Rightarrow killed _____
 - ▶ not zero-centered output
 - ▶ $\exp(\cdot)$ computation
- use logistic sigmoid as **hidden units** in feedforward nets: now **discouraged**
 - ▶ use as **output unit**: acceptable (e.g. probability estimation)

Saturation kills gradient

- widespread _____ of sigmoidal units
⇒ can make gradient-based learning very difficult

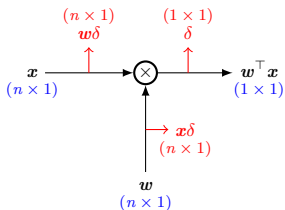


- _____ gradient problem: errors 'vanish' with backprop



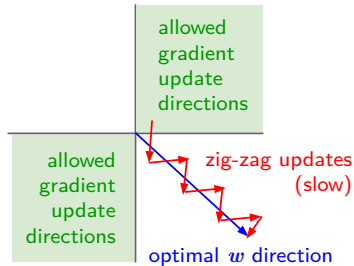
One-sided input slows down training

- recall: $\frac{\partial}{\partial w}(wx) = x$



- gradient of cost function wrt w
 - ▶ directly depends on x

- all positive/negative inputs
 - ▶ cause _____ updates
 - ⇒ slow convergence



(source: cs231n)

- normalization matters!

Practical advice

- feedforward nets
 - ▶ use ReLU (carefully tune learning rates)
 - ▶ try out leaky ReLU, maxout, ELU
 - ▶ try out tanh (but don't expect too much)
 - ▶ do not use sigmoid
 - other than feedforward nets
 - ▶ _____ activations: more common
- e.g. RNN/probabilistic models/some autoencoders

Exploiting linearity

- principle of ReLU (and its generalizations):
 - ▶ models are easier to optimize if their behavior is closer to **linear**
- this principle also applies to recurrent networks
 - ▶ training becomes much easier when some linear computations are involved
 - e.g.* LSTM propagates information through time via *summation*
- linear boundary: sometimes susceptible to _____ examples

Outline

Introduction

Feedforward Networks

- Gradient-Based Learning
- Architecture
- Vectorized Representation

Hidden Units

Forward/Backward Functions

Deep Feedforward Networks

Summary

Forward function

- interface

- ▶ input: $\mathbf{a}^{[l-1]}$
- ▶ output: $\mathbf{a}^{[l]}$
- ▶ cache: $\mathbf{z}^{[l]}$

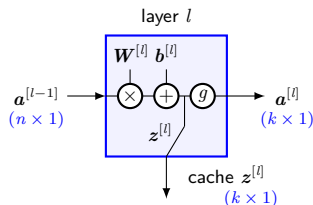
- assumptions

- ▶ $\mathbf{W} = \tilde{\mathbf{W}} \in \mathbb{R}^{(k \times n)}$
- ▶ column-arranged minibatch

- action: 1 example

$$\underbrace{\mathbf{z}^{[l]}}_{(k \times 1)} = \underbrace{\mathbf{W}^{[l]}}_{(k \times n)} \underbrace{\mathbf{a}^{[l-1]}}_{(n \times 1)} + \mathbf{b}^{[l]}$$

$$\underbrace{\mathbf{a}^{[l]}}_{(k \times 1)} = g^{[l]}(\mathbf{z}^{[l]})$$



- action: minibatch (size m)

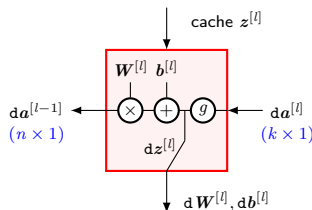
$$\underbrace{\mathbf{Z}^{[l]}}_{(k \times m)} = \underbrace{\mathbf{W}^{[l]}}_{(k \times n)} \underbrace{\mathbf{A}^{[l-1]}}_{(n \times m)} + \mathbf{b}^{[l]}$$

$$\underbrace{\mathbf{A}^{[l]}}_{(k \times m)} = g^{[l]}(\mathbf{Z}^{[l]})$$

Backward function

- interface

- ▶ input: $\mathbf{da}^{[l]}$, cached $\mathbf{z}^{[l]}$
- ▶ output: $\mathbf{da}^{[l-1]}$, $\mathbf{dW}^{[l]}$, $\mathbf{db}^{[l]}$



- action: 1 example

$$\begin{aligned}\underbrace{\mathbf{dz}^{[l]}}_{(k \times 1)} &= \underbrace{\mathbf{da}^{[l]}}_{(k \times 1)} \odot \underbrace{(\mathbf{z}^{[l]})}_{(k \times 1)} \\ \underbrace{\mathbf{dW}^{[l]}}_{(k \times n)} &= \underbrace{\mathbf{dz}^{[l]}}_{(k \times 1)} \underbrace{\mathbf{a}^{[l-1]\top}}_{(1 \times n)} \\ \underbrace{\mathbf{db}^{[l]}}_{(k \times 1)} &= \underbrace{\mathbf{dz}^{[l]}}_{(k \times 1)} \\ \underbrace{\mathbf{da}^{[l-1]}}_{(n \times 1)} &= \underbrace{\mathbf{W}^{[l]\top}}_{(n \times k)} \underbrace{\mathbf{dz}^{[l]}}_{(k \times 1)}\end{aligned}$$

- action: minibatch (size m)

$$\begin{aligned}\underbrace{\mathbf{dZ}^{[l]}}_{(k \times m)} &= \underbrace{\mathbf{dA}^{[l]}}_{(k \times m)} \odot \underbrace{g^{[l]'}(\mathbf{Z}^{[l]})}_{(k \times m)} \\ \underbrace{\mathbf{dW}^{[l]}}_{(k \times n)} &= \frac{1}{m} \underbrace{\mathbf{dZ}^{[l]}}_{(k \times m)} \underbrace{\mathbf{A}^{[l-1]\top}}_{(m \times n)} \\ \underbrace{\mathbf{db}^{[l]}}_{(k \times 1)} &= \frac{1}{m} \underbrace{\mathbf{dZ}^{[l]}}_{(k \times m)} \underbrace{\mathbf{1}}_{(m \times 1)} \\ \underbrace{\mathbf{dA}^{[l-1]}}_{(n \times m)} &= \underbrace{\mathbf{W}^{[l]\top}}_{(n \times k)} \underbrace{\mathbf{dZ}^{[l]}}_{(k \times m)}\end{aligned}$$

- post-action (update): $\mathbf{W}^{[l]} \leftarrow \mathbf{W}^{[l]} - \epsilon \mathbf{dW}^{[l]}$

$$\mathbf{b}^{[l]} \leftarrow \mathbf{b}^{[l]} - \epsilon \mathbf{db}^{[l]}$$

Exhaustive summary

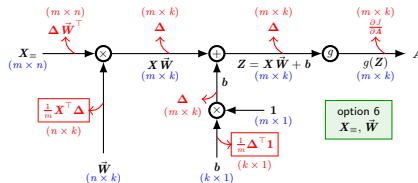
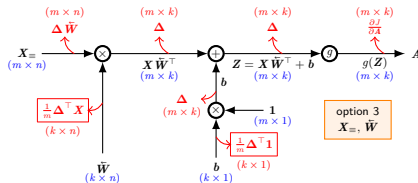
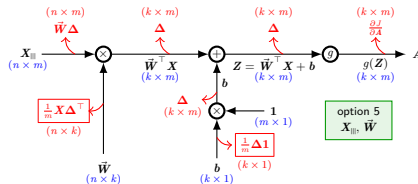
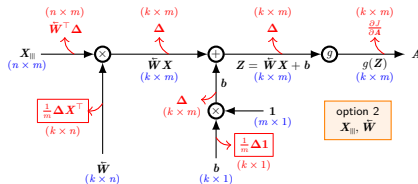
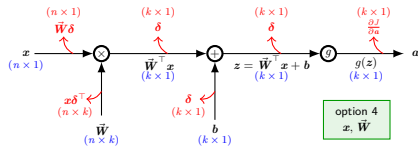
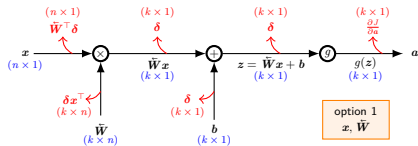
(notation: $d\star \triangleq \frac{\partial J}{\partial \star}$)

- RL-convention weight matrix: \vec{W} ($k \times n$)

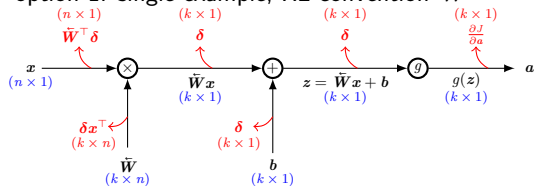
input	signal (z, Z)	output	δ -error (dz, dZ)	dW	db	dx, dX	opt
x ($n \times 1$)	$Wx + b$ ($k \times 1$)	$a = g(z)$ ($k \times 1$)	$da \odot g'(z) \triangleq \delta$ ($k \times 1$)	δx^\top ($k \times n$)	δ ($k \times 1$)	$W^\top \delta$ ($n \times 1$)	1
X_{\parallel} ($n \times m$)	$WX + b$ ($k \times m$)	$A = g(Z)$ ($k \times m$)	$dA \odot g'(Z) \triangleq \Delta$ ($k \times m$)	$\frac{1}{m} \Delta X^\top$ ($k \times n$)	$\frac{1}{m} \Delta \mathbf{1}_{m \times 1}$ ($k \times 1$)	$W^\top \Delta$ ($n \times m$)	2
X_{\equiv} ($m \times n$)	$XW^\top + b$ ($m \times k$)	A ($m \times k$)	Δ ($m \times k$)	$\frac{1}{m} \Delta^\top X$ ($k \times n$)	$\frac{1}{m} \Delta^\top \mathbf{1}_{m \times 1}$ ($k \times 1$)	ΔW ($m \times n$)	3

- LR-convention weight matrix: \vec{W} ($n \times k$)

input	signal (z, Z)	output	δ -error (dz, dZ)	dW	db	dx, dX	opt
x ($n \times 1$)	$W^\top x + b$ ($k \times 1$)	a ($k \times 1$)	δ ($k \times 1$)	$x \delta^\top$ ($n \times k$)	δ ($k \times 1$)	$W \delta$ ($n \times 1$)	4
X_{\parallel} ($n \times m$)	$W^\top X + b$ ($k \times m$)	A ($k \times m$)	Δ ($k \times m$)	$\frac{1}{m} X \Delta^\top$ ($n \times k$)	$\frac{1}{m} \Delta \mathbf{1}_{m \times 1}$ ($k \times 1$)	$W \Delta$ ($n \times m$)	5
X_{\equiv} ($m \times n$)	$XW + b$ ($m \times k$)	A ($m \times k$)	Δ ($m \times k$)	$\frac{1}{m} X^\top \Delta$ ($n \times k$)	$\frac{1}{m} \Delta^\top \mathbf{1}_{m \times 1}$ ($k \times 1$)	ΔW^\top ($m \times n$)	6



- option 1: single example, RL-convention \vec{W}



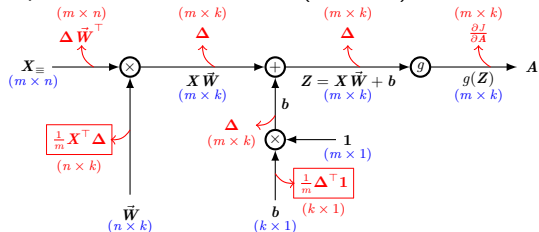
★ use in textbook

► algorithm 6.2

► algorithm 6.3

★ coursera⁴

- option 6: size- m minibatch (row-wise), LR-convention \vec{W}

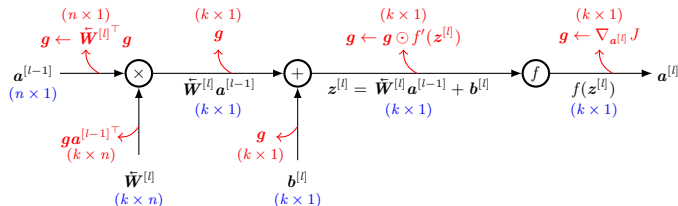


★ use in textbook

► sec 6.5.7

⁴uses option 1 for single example and option 3 for minibatch

- algorithms using textbook notation (option 1, single example)



algorithm 1 forward computation

```

1:  $a^{[0]} = x$ 
2: for  $l = 1, \dots, L$  do
3:    $z^{[l]} = \tilde{W}^{[l]} a^{[l-1]} + b^{[l]}$ 
4:    $a^{[l]} = f(z^{[l]})$ 
5: end for
6:  $\hat{y} = a^{[L]}$ 
7:  $J = L(y, \hat{y}) + \lambda \Omega(\theta)$ 

```

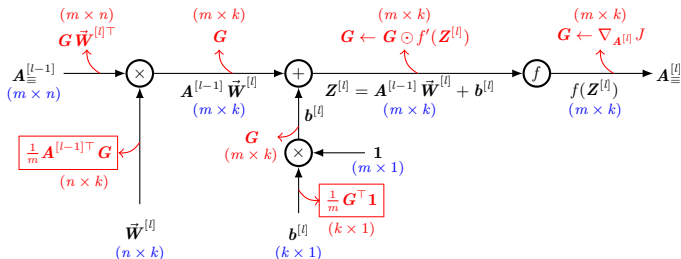
algorithm 2 backward computation

```

1:  $g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$ 
2: for  $l = L, L-1, \dots, 1$  do
3:    $g \leftarrow \nabla_{z^{[l]}} J = g \odot f'(z^{[l]})$ 
4:    $\nabla_{b^{[l]}} J = g + \lambda \nabla_{b^{[l]}} \Omega(\theta)$ 
5:    $\nabla_{\tilde{W}^{[l]}} J = g a^{[l-1]T} + \lambda \nabla_{\tilde{W}^{[l]}} \Omega(\theta)$ 
6:    $g \leftarrow \nabla_{a^{[l-1]}} J = \tilde{W}^{[l]T} g$ 
7: end for

```

- algorithms using textbook notation (option 6, minibatch)



algorithm 3 forward computation

- 1: $A^{[0]} = X_{\equiv}$
- 2: **for** $l = 1, \dots, L$ **do**
- 3: $Z^{[l]} = A^{[l-1]} \tilde{W}^{[l]} + b^{[l]}$
- 4: $A^{[l]} = f(Z^{[l]})$
- 5: **end for**
- 6: $\hat{Y} = A^{[L]}$
- 7: $J = L(Y, \hat{Y}) + \lambda \Omega(\theta)$

algorithm 4 backward computation

- 1: $G \leftarrow \nabla_{\hat{Y}} J = \nabla_{\hat{Y}} L(\hat{Y}, Y)$
- 2: **for** $l = L, L-1, \dots, 1$ **do**
- 3: $G \leftarrow \nabla_{Z^{[l]}} J = G \odot f'(Z^{[l]})$
- 4: $\nabla_{b^{[l]}} J = \frac{1}{m} G^T \mathbf{1} + \lambda \nabla_{b^{[l]}} \Omega(\theta)$
- 5: $\nabla_{\tilde{W}^{[l]}} J = \frac{1}{m} G + \lambda \nabla_{\tilde{W}^{[l]}} \Omega(\theta)$
- 6: $G \leftarrow \nabla_{A^{[l-1]}} J = G \tilde{W}^{[l]T}$
- 7: **end for**

algorithm 5 back propagation (minibatch of size m ; learning rate ϵ)

1: initialize all parameters W, b

2: **repeat**

3: pick a minibatch \mathbb{X}_m from \mathbb{X}

4: **forward:** compute all activations A

5: compute cost $J = \frac{1}{m} \sum L(Y^{(i)}, \hat{Y}^{(i)}) + \lambda \Omega(\theta)$

6: **backward:** compute all gradients

7: **update parameters:**

$$W \leftarrow W - \epsilon dW \quad (\text{weights})$$

$$b \leftarrow b - \epsilon db \quad (\text{bias})$$

8: **until** it is time to stop

9: return final parameters

$$W^*, b^*$$

Remarks

- complications of backprop in practice
 - ▶ multi-output operation
 - ▶ memory considerations
 - ▶ supporting diverse data types
 - ▶ handling undefined gradients
- field of automatic differentiation:
 - ▶ concerned with how to compute derivatives algorithmically
 - ▶ backprop: a special case of *reverse mode accumulation*
- c.f. real-time recurrent learning (RTRL): *forward mode accumulation*
- implementations such as theano and TensorFlow
 - ▶ use heuristics to iteratively simplify backprop graph for efficiency

Outline

Introduction

Feedforward Networks

Deep Feedforward Networks

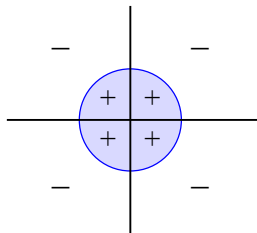
Summary

Architecture exploration

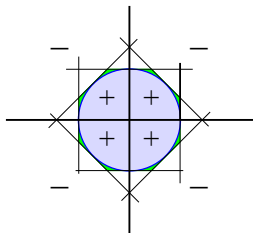
- main architectural considerations in chain-based architectures
 - ▶ network **depth** and layer **width**
- a feedforward net with a **single layer**
 - ▶ sufficient to represent any function
 - ▶ but may have infeasibly large layer and
 - ▶ may fail to learn and generalize correctly
- **deeper** networks
 - ▶ use far fewer units per layer and far fewer parameters
 - ▶ _____ better to test set
 - ▶ but harder to optimize (*e.g.* vanishing/exploding gradient)
- ideal architecture for a task
 - ▶ must be found via experimentation (guided by validation error)

Universal approximation theorem (Hornik *et al.*, '89; Cybenko, '89)

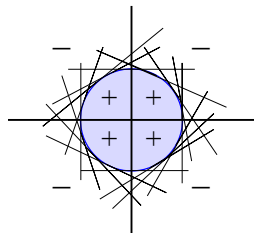
- a feedforward net with linear output layer + hidden layer(s)
 - ▶ can **approximate** any⁵ function (given enough _____ units)



target



8 perceptrons



16 perceptrons

(source: Abu-Mostafa)

- but the ability to **learn** that function: not guaranteed

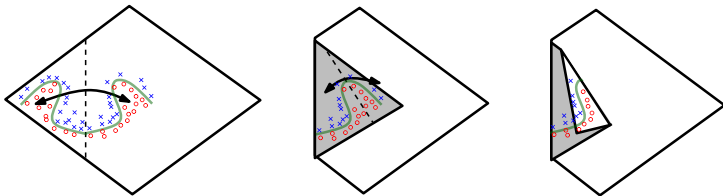
⁵should be Borel measurable: see textbook

Network size

- universal approximation theorem
 - ▶ says there exists a network large enough to achieve any accuracy
 - ▶ but does not say how large this network will be
 - unfortunately
 - ▶ an _____ number of hidden units may be required in the worse case
- e.g. binary case
- 2^{2^n} : the number of possible binary functions on vectors $v \in \{0, 1\}^n$
- 2^n bits required to select one such function
- ⇒ which will in general require $O(2^n)$ degrees of freedom

Exponential advantage of deeper networks

- some families of functions
 - ▶ can be approximated efficiently with depth $> d$
 - ▶ but require a much larger model if depth is restricted to $\leq d$
 - ▷ such _____ model requires exponential # of hidden units
 - Montufar *et al.* (2014) showed: piecewise linear networks
 - ▶ can represent functions with a number of regions exponential in net depth
- e.g. two hidden units \Rightarrow four regions



Statistical interpretation

- choosing a **specific ML algorithm** = encoding our prior beliefs



about what **kind of function** the algorithm should learn

- choosing a **deep** model = encoding a very general belief



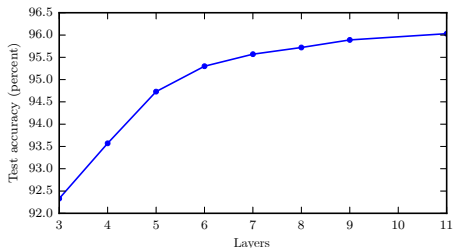
the function to learn should involve **composition of simpler functions**

- empirically:

greater	⇒	better generalization
---------	---	-----------------------

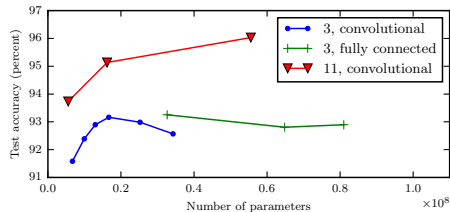
- ▶ two examples on next page

- test set accuracy consistently increases with increasing depth



- other increases to model size
 - ▶ do not yield the same effect
- task: from photos of addresses
 - ▶ transcribe multi-digit numbers

- increasing # of _____ without increasing depth: not effective



- shallow models overfit
 - ▶ at ~ 20 million parameters
- deep ones can benefit
 - ▶ from having over 60 million

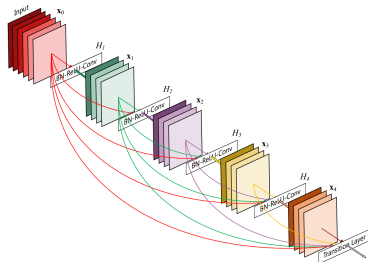
Other architectural considerations

- so far: neural nets as simple chains of layers
 - ▶ main considerations: _____ of network and _____ of each layer
- in practice: neural nets show considerably more diversity
 - ▶ many net architectures: task specific
 - ▷ CNNs: computer vision (ch 9)
 - ▷ RNNs: sequence processing (ch 10)
 - ▶ these have their own architectural considerations

Chaining

- layers need not be connected in a chain
 - ▶ even though this is the most common practice
- many architectures build a main chain
 - ▶ but then add extra architectural features to it

- e.g. _____ connections:
 - ▶ go from layer i to layer $i + 2$ or higher
 - ▶ make **gradient flow more easily** from output layers to layers nearer input
 - ▶ ResNet, HighwayNet, DenseNet (shown) [▶ Link](#)



Layer-wise connection

- another key consideration of architecture design:
 - ▶ how to connect a pair of layers to each other
 - in default neural net layer
 - (described by linear transformation via matrix W)
 - ▶ every input unit: connected to every output unit
 - many specialized networks have fewer connections
 - ▶ each input unit: connected to only a small _____ of output units
 - these strategies for reducing # of connections
 - ▶ reduce # of parameters/amount of computation to evaluate the net
 - ▶ but are highly problem-dependent → see later chapters
- e.g. CNNs (ch 9): sparse connection patterns effective for vision problems

Outline

Introduction

Feedforward Networks

Deep Feedforward Networks

Summary

Summary

- deep feed forward net: quintessential deep model
 - ▶ universal function approximator parameterized by $\theta = (W, b)$
 - ▶ learn θ by gradient-based backprop algorithm
- building blocks of deep feedforward nets
 - ▶ neuron: modeled by logistic regression
 - ▶ forward function: propagates x to output, giving loss $L(y, \hat{y})$
 - ▶ backward function: propagates da to input, giving dW, db
 - ▶ update: $dW \leftarrow dW - \epsilon da$, $db \leftarrow db - \epsilon db$
 - ▶ activation function: ReLU/variants are popular for deep feedforward nets
 - ▶ output units: linear, sigmoid, softmax units
- deep feedforward neural nets
 - ▶ more depth gives better generalization, but training is challenging

⇒ architectural modifications in convolutional nets/recurrent nets