M2177.003100
Deep Learning

[3: Deep Feedforward Networks]

Electrical and Computer Engineering
Seoul National University

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

Outline

Introduction

Feedfoward 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

Feedfoward 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: $y = f^*(\boldsymbol{x})$ maps input $\boldsymbol{x}$ to category y

- how it works: $\underline{\text{parameter}}$ + learn
  - define a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and
  - learn parameters $\boldsymbol{\theta}$ that give the best approximation

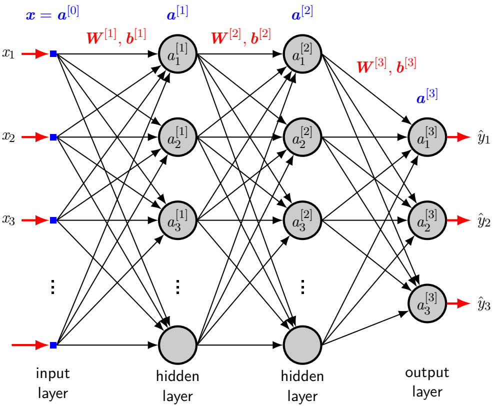
- extremely important
  - basis of many important commercial applications

  e.g. convolutional nets, recurrent nets

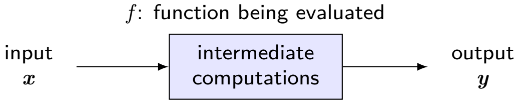
Architecture

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

- parameters $\boldsymbol{\theta}$: weight $\boldsymbol{W}$ and bias $\boldsymbol{b}$



- called feed-forward because
  - information flows $\boldsymbol{x} \rightarrow \boxed{f} \rightarrow \boldsymbol{y}$



- no **feedback** connections
  - no output is fed back into the model

  c.f. recurrent neural nets (ch 10)

  ex) RNN은 feedback이 있음

- called **networks** because
  - represented by composing many different functions

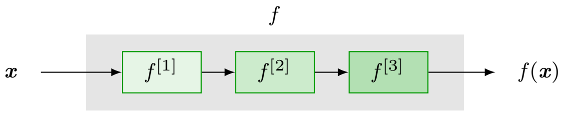
- associated with a $\underbrace{\text{directed acyclic graph}}_{\uparrow}$

  describes how component functions are composed together

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

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

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



Training neural nets

- training: we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$
- training data: noisy/approximate examples of $f^*(\boldsymbol{x})$
  - each example: $(\underbrace{\boldsymbol{x}}_{\text{input}}, \underbrace{y}_{\text{label}})$ with $y \approx f^*(\boldsymbol{x})$

$\Rightarrow$ directly specifies what output layer must do at each $\boldsymbol{x}$

- behavior of the other layers: not directly specified by training data
  - these layers: called <u>hidden</u> 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 non–linear functions of $\boldsymbol{x}$
  - apply the linear model not to $\boldsymbol{x}$ itself but to transformed input $\phi(\boldsymbol{x})$

$$\boldsymbol{x} \longrightarrow \boxed{\begin{array}{c}\text{nonlinear transform}\\ \phi\end{array}} \longrightarrow \underbrace{\phi(\boldsymbol{x})}$$

$=$ features describing $\boldsymbol{x}$ or
$=$ new representation for $\boldsymbol{x}$

  - 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

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

     - parameters $\boldsymbol{\theta}$: used to learn ϕ from a broad class of functions
     - parameters $\boldsymbol{w}$: map from $\phi(\boldsymbol{x})$ to desired output

- (deep) feedforward nets:
  - learn deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ (no feedback connections)
  - ϕ defines a hidden 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

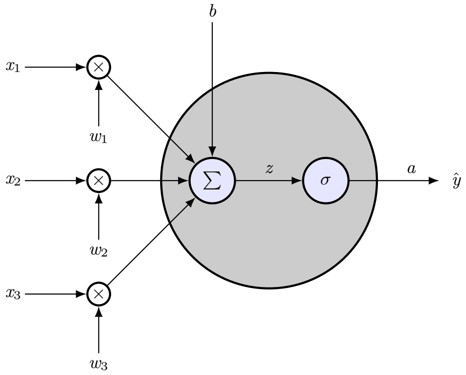
Introduction

Feedfoward Networks
- Gradient-Based Learning
- Architecture
- Vectorized Representation
- Hidden Units
- Forward/Backward Functions

Deep Feedforward Networks

Summary

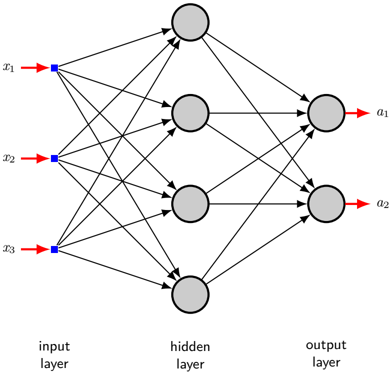
Recall: logistic regression as a neuron model

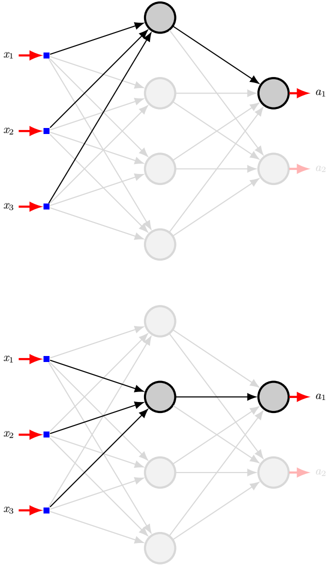


$$\begin{aligned} \text{signal:} \quad & z = \boldsymbol{w}^\top \boldsymbol{x} + b \\ \text{activation:} \quad & a = \sigma(z) \end{aligned}$$

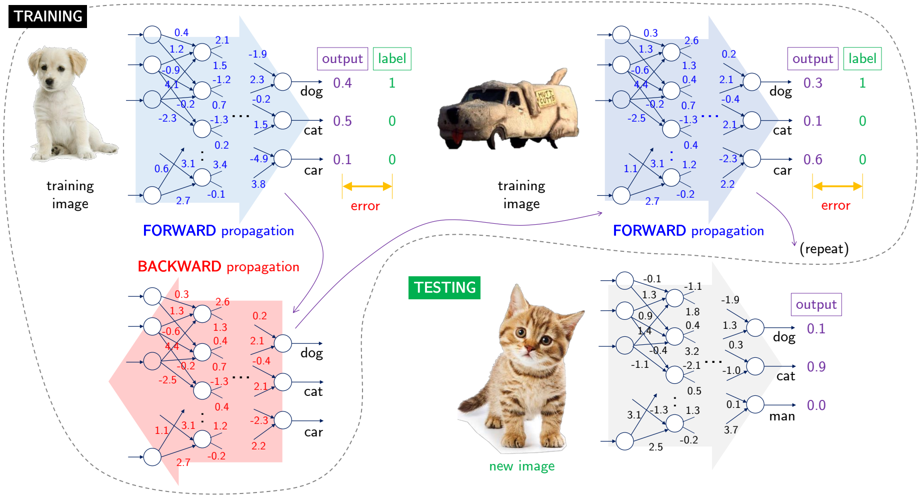
Feedforward neural net

- composed of logistic regression units





Concept of training & testing a neural net



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 $\longrightarrow$ $\boxed{\text{cross-entropy loss}}$
2. sigmoid $\longrightarrow$ $\boxed{\text{ReLU}}$

Outline

Introduction

Feedfoward 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 small random 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. $\underline{\text{cost}}$ 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

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

- a recurring theme: gradient of cost function must be large/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 $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- output layer:
  - provides additional transformation from features to output

$$\boldsymbol{h} \longrightarrow \boxed{\begin{matrix}\text{output}\\ \text{unit}\end{matrix}} \longrightarrow \hat{\boldsymbol{y}}$$

  - most common: linear/sigmoid/softmax output units

- softmax[2] units: represent probability distribution over K classes
  - $\underbrace{\text{bernoulli}}_{\text{distribution}} : \underbrace{\text{sigmoid}}_{\text{function}} = \text{multinoulli} : \text{softmax}$

[2] better name: “softargmax”

Multinoulli (or categorical) distribution

- a distribution over a single discrete variable with k finite states
  - parameterized by vector $\boldsymbol{p} \in [0,1]^{k-1}$ (p_i : probability of i-th state)
  - $1 - \mathbf{1}^\top \boldsymbol{p}$: the final, k-th state's probability ($\mathbf{1}^\top \boldsymbol{p} \leq 1$)

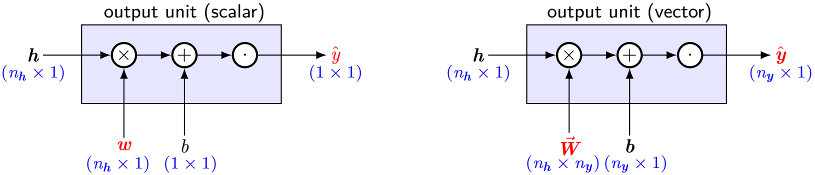
- "multinoulli": recently coined term[3]
  - as a special case (*i.e.* single trial) of multinomial distribution
  - multinomial distribution: a distribution over vectors in $\{0, \ldots, n\}^k$
    - represents how many times each of k categories is visited when n samples are drawn from a multinoulli distribution

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

[3] many 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	$\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$	Gaussian
sigmoid	scalar	$\hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{h} + b)$	Bernoulli
softmax	vector	$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b})$	multinoulli



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

Outline

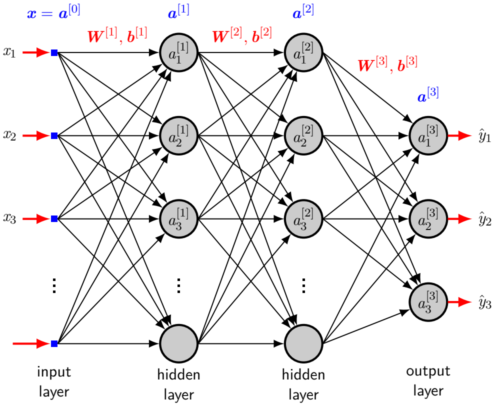
Introduction

Feedfoward Networks
- Gradient-Based Learning
- Architecture
- Vectorized Representation
- Hidden Units
- Forward/Backward Functions

Deep Feedforward Networks

Summary

Notation



- layer/node indices

$$\boxed{a_j^{[l]} \quad \begin{array}{l}\leftarrow \text{layer} \\ \leftarrow \text{node index}\end{array}}$$

- parameters
  - weight: $\boldsymbol{W}^{[l]}$
  - bias: $\boldsymbol{b}^{[l]}$
- gradient: $\boxed{\mathtt{d}\bigstar \triangleq \frac{\partial J}{\partial \bigstar}}$

e.g.

$$\mathtt{d}\boldsymbol{z} = \frac{\partial J}{\partial \boldsymbol{z}}$$

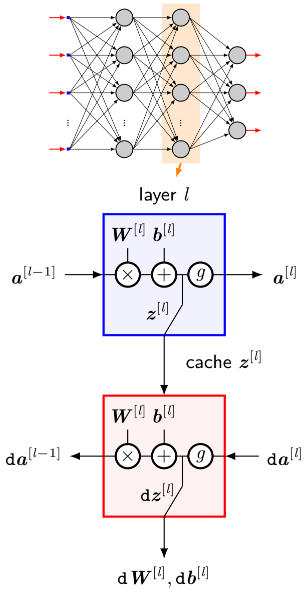
$$\mathtt{d}\boldsymbol{a} = \frac{\partial J}{\partial \boldsymbol{a}}$$

$$\mathtt{d}\boldsymbol{W} = \frac{\partial J}{\partial \boldsymbol{W}}$$

$$\mathtt{d}\boldsymbol{b} = \frac{\partial J}{\partial \boldsymbol{b}}$$

- notes:
  - J: cost function
  - $\bigstar$ and $\mathtt{d}\bigstar = \frac{\partial J}{\partial \bigstar}$ have the same size

Operations for each layer



$\boxed{\text{layer } l}$

- parameters: $\boldsymbol{W}^{[l]}, \boldsymbol{b}^{[l]}$
- activation function: $g^{[l]}$

- forward function
  - input: $\boldsymbol{a}^{[l-1]}$
  - output: $\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$
  - cache: $\boldsymbol{z}^{[l]}$

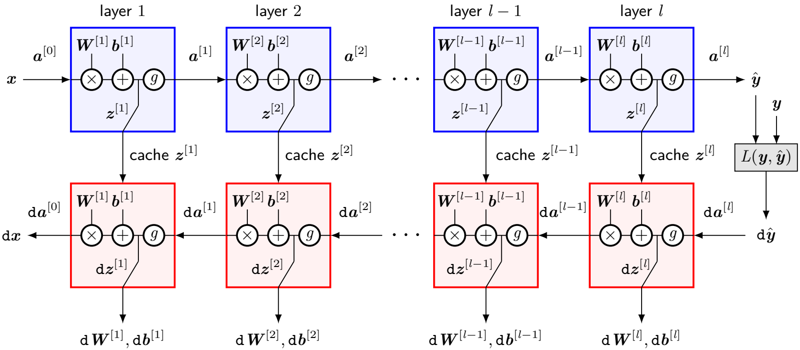
- backward function
  - input: $\mathtt{d}\boldsymbol{a}^{[l]}$, cached $\boldsymbol{z}^{[l]}$
  - output: $\mathtt{d}\boldsymbol{a}^{[l-1]}, \mathtt{d}\boldsymbol{W}^{[l]}, \mathtt{d}\boldsymbol{b}^{[l]}$

- parameter update

$$\boldsymbol{W}^{[l]} \leftarrow \boldsymbol{W}^{[l]} - \epsilon \mathtt{d}\boldsymbol{W}^{[l]}$$

$$\boldsymbol{b}^{[l]} \leftarrow \boldsymbol{b}^{[l]} - \epsilon \mathtt{d}\boldsymbol{b}^{[l]}$$

Overall architecture



parameter update (ϵ: learning rate)

$$\boldsymbol{W}^{[l]} \leftarrow \boldsymbol{W}^{[l]} - \epsilon \underline{\text{ dW^l }}$$

$$\boldsymbol{b}^{[l]} \leftarrow \boldsymbol{b}^{[l]} - \epsilon \mathrm{d}\boldsymbol{b}^{[l]}$$

Outline

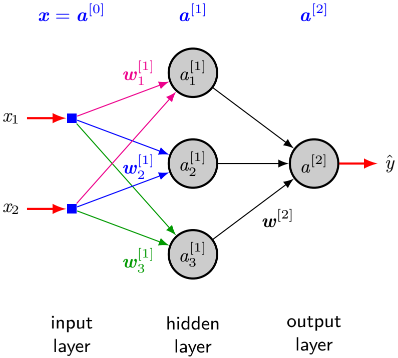
Introduction

Feedfoward Networks
- Gradient-Based Learning
- Architecture
- Vectorized Representation
- Hidden Units
- Forward/Backward Functions

Deep Feedforward Networks

Summary

A running example



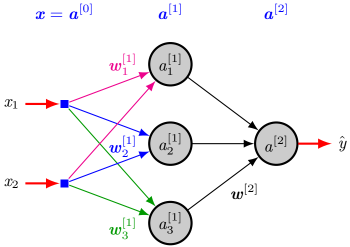
- hidden layer

$$a_1^{[1]} = g(\boldsymbol{w}_1^{[1]\top}\boldsymbol{x} + b_1^{[1]})$$
$$a_2^{[1]} = g(\boldsymbol{w}_2^{[1]\top}\boldsymbol{x} + b_2^{[1]})$$
$$a_3^{[1]} = g(\boldsymbol{w}_3^{[1]\top}\boldsymbol{x} + b_3^{[1]})$$

- output layer

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

Vectorized representation



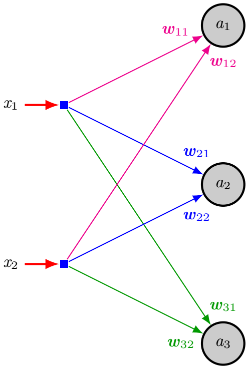
- separate equations

$$a_1^{[1]} = g(\boldsymbol{w}_1^{[1]\top}\boldsymbol{x} + b_1^{[1]}) = g(z_1^{[1]})$$
$$a_2^{[1]} = g(\boldsymbol{w}_2^{[1]\top}\boldsymbol{x} + b_2^{[1]}) = g(z_2^{[1]})$$
$$a_3^{[1]} = g(\boldsymbol{w}_3^{[1]\top}\boldsymbol{x} + b_3^{[1]}) = g(z_3^{[1]})$$

- vectorized equations

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

Weight matrix conventions



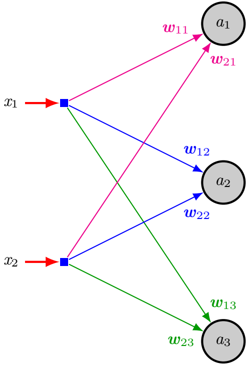
- RL (right-left) convention
  - weight for $i \to j$: w_{ji}

$$\overleftarrow{\boldsymbol{W}} = \begin{bmatrix} \boldsymbol{w}_{11} & \boldsymbol{w}_{12} \\ \boldsymbol{w}_{21} & \boldsymbol{w}_{22} \\ \boldsymbol{w}_{31} & \boldsymbol{w}_{32} \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

- then

$$\begin{bmatrix} \text{---} & \boldsymbol{w}_1^\top & \text{---} \\ \text{---} & \boldsymbol{w}_2^\top & \text{---} \\ \text{---} & \boldsymbol{w}_3^\top & \text{---} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_{11} & \boldsymbol{w}_{12} \\ \boldsymbol{w}_{21} & \boldsymbol{w}_{22} \\ \boldsymbol{w}_{31} & \boldsymbol{w}_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$= \text{Wx}$$



- LR (left-right) convention
  - weight for $i \to j$: w_{ij}

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

- then

$$\begin{bmatrix} \text{——} & \boldsymbol{w}_1^\top & \text{——} \\ \text{——} & \boldsymbol{w}_2^\top & \text{——} \\ \text{——} & \boldsymbol{w}_3^\top & \text{——} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_{11} & \boldsymbol{w}_{21} \\ \boldsymbol{w}_{12} & \boldsymbol{w}_{22} \\ \boldsymbol{w}_{13} & \boldsymbol{w}_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$=$ W^T * x

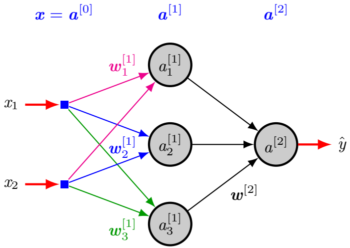
Vectorized representation

- two flavors

$$\boldsymbol{z}^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \end{bmatrix} = \begin{bmatrix} \text{——} & \boldsymbol{w}_1^{[1]\top} & \text{——} \\ \text{——} & \boldsymbol{w}_2^{[1]\top} & \text{——} \\ \text{——} & \boldsymbol{w}_3^{[1]\top} & \text{——} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \end{bmatrix}$$

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

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



Outline

Introduction

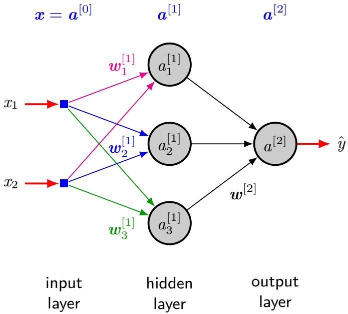
Feedfoward Networks
- Gradient-Based Learning
- Architecture
- Vectorized Representation
- Hidden Units
- Forward/Backward Functions

Deep Feedforward Networks

Summary

Hidden units

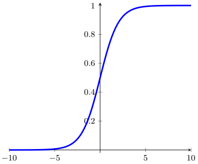
- what they do:
  1. accept a vector of inputs $\boldsymbol{x}$
  2. compute an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$
  3. apply an element-wise nonlinear function g to $\boldsymbol{z}$
  4. return activation $\boldsymbol{a} = g(\boldsymbol{z})$



- hidden units differ only in activation function $g(\boldsymbol{z})$
- Rectified 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

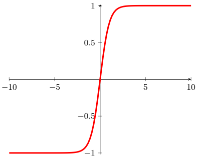
  e.g. $g(\boldsymbol{z}) = \cos(\boldsymbol{z})$ gives $< 1\%$ error on MNIST
  - new types: published only if clearly show significant improvement
- notation
  - $g^{[l]}$: activation function for layer l
  - mixing activation function types in a layer: uncommon

Activation functions



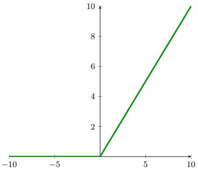
- sigmoid

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



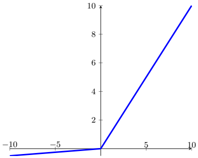
- tanh

$$\tanh(z)$$



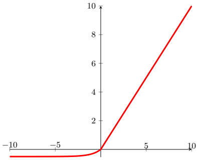
- ReLU

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



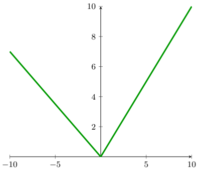
- leaky ReLU

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



- ELU

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

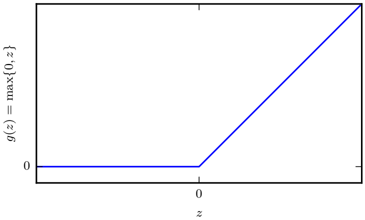


- maxout

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

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
  - zero gradient in $(-)$ region

=> gradient가 0이므로 train이 아예 안될 수 있

ReLU Initialization

- ReLU:
  - typically used on top of an affine transformation:

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

- good practice:
  - set all elements of $\boldsymbol{b}$ to a <u>small positive</u> number (*e.g.* 0.1 or 0.01)

$\Rightarrow$ ReLU initially active for most inputs in training set

$\Rightarrow$ 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 consistent
    - 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: $g(z) = \max\{\alpha z, z\}$ (learnable α)
4. exponential ReLU:

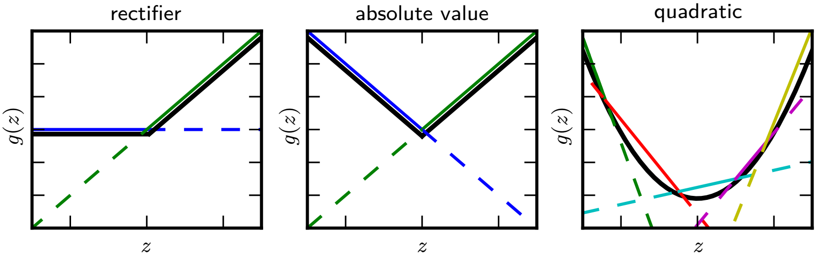
$$g(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}$$

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

Further generalization: maxout units

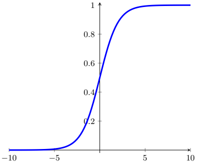
- *learn the activation function* itself
  - learn a piecewise linear, convex function with up to k pieces
  - approximate any convex function with arbitrary fidelity (with large k)

- cons: more parameters/neurons required

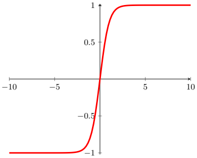


Prior to ReLU

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



  - tanh: $g(z) = \tanh(z)$

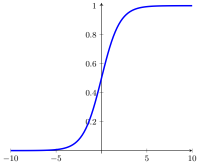


  - closely related: $\tanh(z) = 2\sigma(2z) - 1$

- 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 tanh when a sigmoidal activation function must be used

일반적으로 tanh보다 ReLU가 더 좋음

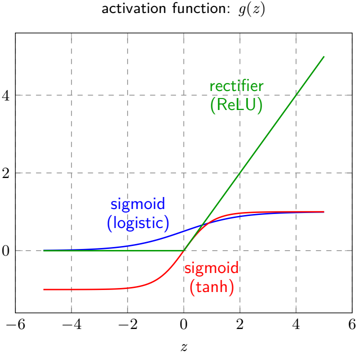
Logistic sigmoid activation

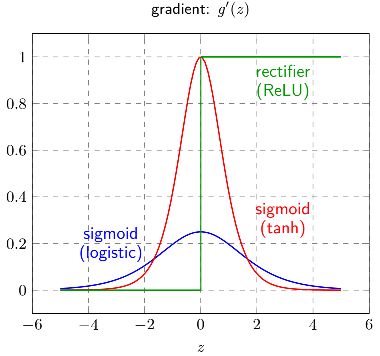


- historically popular
  - outputs to range $[0, 1]$
  - nice interpretation
    - saturating “firing rate” of a neuron
- cons
  - saturation $\Rightarrow$ killed gradient
  - not zero-centered output
  - $\exp(\cdot)$ computation
- use logistic sigmoid as hidden units in feedfoward nets: now discouraged
  - use as output unit: acceptable (*e.g.* probability estimation)

Saturation kills gradient

- widespread <u>saturation</u> of sigmoidal units
  - $\Rightarrow$ can make gradient-based learning very difficult



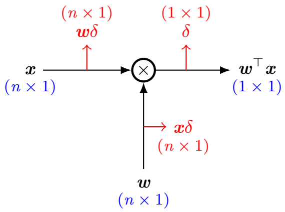


- vanishing gradient problem: errors ‘vanish’ with backprop

One-sided input slows down training

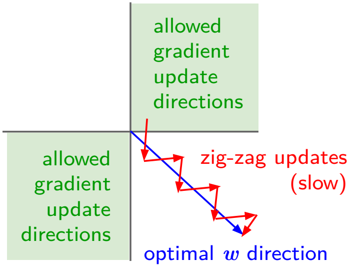
- recall: $\frac{\partial}{\partial w}(wx) = x$



- gradient of cost function wrt $\boldsymbol{w}$
  - directly depends on $\boldsymbol{x}$

- all positive/negative inputs
  - cause zig–zag updates

$\Rightarrow$ 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
  - sigmoidal 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 adversarial examples

Outline

Introduction

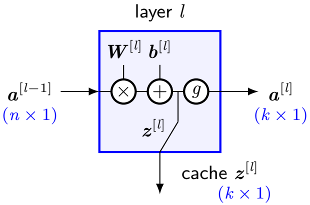
Feedfoward Networks
Gradient-Based Learning
Architecture
Vectorized Representation
Hidden Units
Forward/Backward Functions

Deep Feedforward Networks

Summary

Forward function

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



- assumptions
  - $\boldsymbol{W} = \overleftarrow{\boldsymbol{W}} \in \mathbb{R}^{(k \times n)}$
  - column-arranged minibatch

- action: 1 example

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

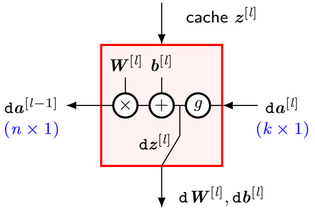
$$\underbrace{\boldsymbol{a}^{[l]}}_{(k\times 1)} = g^{[l]}(\boldsymbol{z}^{[l]})$$

- action: minibatch (size m)

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

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

Backward function



- interface
  - input: $\mathtt{d}\boldsymbol{a}^{[l]}$, cached $\boldsymbol{z}^{[l]}$
  - output: $\mathtt{d}\boldsymbol{a}^{[l-1]}, \mathtt{d}\boldsymbol{W}^{[l]}, \mathtt{d}\boldsymbol{b}^{[l]}$

- action: 1 example

$$\underbrace{\mathtt{d}\boldsymbol{z}^{[l]}}_{(k\times 1)} = \underbrace{\mathtt{d}\boldsymbol{a}^{[l]}}_{(k\times 1)} \odot \text{g\^{}[l]`} \underbrace{(\boldsymbol{z}^{[l]})}_{(k\times 1)}$$

$$\underbrace{\mathtt{d}\boldsymbol{W}^{[l]}}_{(k\times n)} = \underbrace{\mathtt{d}\boldsymbol{z}^{[l]}}_{(k\times 1)} \underbrace{\boldsymbol{a}^{[l-1]\top}}_{(1\times n)}$$

$$\underbrace{\mathtt{d}\boldsymbol{b}^{[l]}}_{(k\times 1)} = \underbrace{\mathtt{d}\boldsymbol{z}^{[l]}}_{(k\times 1)}$$

$$\underbrace{\mathtt{d}\boldsymbol{a}^{[l-1]}}_{(n\times 1)} = \underbrace{\boldsymbol{W}^{[l]\top}}_{(n\times k)} \underbrace{\mathtt{d}\boldsymbol{z}^{[l]}}_{(k\times 1)}$$

- action: minibatch (size m)

$$\underbrace{\mathtt{d}\boldsymbol{Z}^{[l]}}_{(k\times m)} = \underbrace{\mathtt{d}\boldsymbol{A}^{[l]}}_{(k\times m)} \odot \underbrace{g^{[l]\prime}(\boldsymbol{Z}^{[l]})}_{(k\times m)}$$

$$\underbrace{\mathtt{d}\boldsymbol{W}^{[l]}}_{(k\times n)} = \frac{1}{m} \underbrace{\mathtt{d}\boldsymbol{Z}^{[l]}}_{(k\times m)} \underbrace{\boldsymbol{A}^{[l-1]\top}}_{(m\times n)}$$

$$\underbrace{\mathtt{d}\boldsymbol{b}^{[l]}}_{(k\times 1)} = \frac{1}{m} \underbrace{\mathtt{d}\boldsymbol{Z}^{[l]}}_{(k\times m)} \underbrace{\mathbf{1}}_{(m\times 1)}$$

$$\underbrace{\mathtt{d}\boldsymbol{A}^{[l-1]}}_{(n\times m)} = \underbrace{\boldsymbol{W}^{[l]\top}}_{(n\times k)} \underbrace{\mathtt{d}\boldsymbol{Z}^{[l]}}_{(k\times m)}$$

- post-action (update): $\boldsymbol{W}^{[l]} \leftarrow \boldsymbol{W}^{[l]} - \epsilon \mathtt{d}\boldsymbol{W}^{[l]}$ $\quad\quad$ $\boldsymbol{b}^{[l]} \leftarrow \boldsymbol{b}^{[l]} - \epsilon \mathtt{d}\boldsymbol{b}^{[l]}$

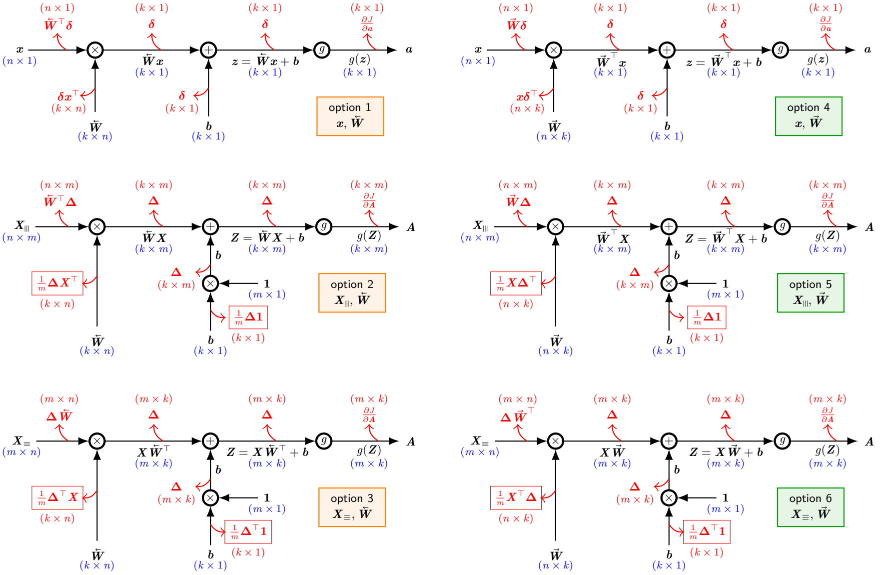
Exhaustive summary (notation: $\mathrm{d}\bigstar \triangleq \frac{\partial J}{\partial \bigstar}$)

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

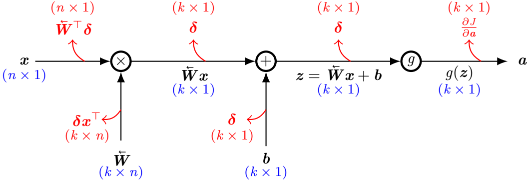
input	signal $(\boldsymbol{z}, \boldsymbol{Z})$	output	δ-error $(\mathrm{d}\boldsymbol{z}, \mathrm{d}\boldsymbol{Z})$	$\mathrm{d}\boldsymbol{W}$	$\mathrm{d}\boldsymbol{b}$	$\mathrm{d}\boldsymbol{x}, \mathrm{d}\boldsymbol{X}$	opt
$\boldsymbol{x}$ $(n \times 1)$	$\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ $(k \times 1)$	$\boldsymbol{a} = g(\boldsymbol{z})$ $(k \times 1)$	$\mathrm{d}\boldsymbol{a} \odot g'(\boldsymbol{z}) \triangleq \boldsymbol{\delta}$ $(k \times 1)$	$\boldsymbol{\delta}\boldsymbol{x}^\top$ $(k \times n)$	$\boldsymbol{\delta}$ $(k \times 1)$	$\boldsymbol{W}^\top\boldsymbol{\delta}$ $(n \times 1)$	1
$\boldsymbol{X}_{\vert\vert\vert}$ $(n \times m)$	$\boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}$ $(k \times m)$	$\boldsymbol{A} = g(\boldsymbol{Z})$ $(k \times m)$	$\mathrm{d}\boldsymbol{A} \odot g'(\boldsymbol{Z}) \triangleq \boldsymbol{\Delta}$ $(k \times m)$	$\frac{1}{m}\boldsymbol{\Delta}\boldsymbol{X}^\top$ $(k \times n)$	$\frac{1}{m}\boldsymbol{\Delta}\boldsymbol{1}_{m\times 1}$ $(k \times 1)$	$\boldsymbol{W}^\top\boldsymbol{\Delta}$ $(n \times m)$	2
$\boldsymbol{X}_{\equiv}$ $(m \times n)$	$\boldsymbol{X}\boldsymbol{W}^\top + \boldsymbol{b}$ $(m \times k)$	$\boldsymbol{A}$ $(m \times k)$	$\boldsymbol{\Delta}$ $(m \times k)$	$\frac{1}{m}\boldsymbol{\Delta}^\top\boldsymbol{X}$ $(k \times n)$	$\frac{1}{m}\boldsymbol{\Delta}^\top\boldsymbol{1}_{m\times 1}$ $(k \times 1)$	$\boldsymbol{\Delta}\boldsymbol{W}$ $(m \times n)$	3

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

input	signal $(\boldsymbol{z}, \boldsymbol{Z})$	output	δ-error $(\mathrm{d}\boldsymbol{z}, \mathrm{d}\boldsymbol{Z})$	$\mathrm{d}\boldsymbol{W}$	$\mathrm{d}\boldsymbol{b}$	$\mathrm{d}\boldsymbol{x}, \mathrm{d}\boldsymbol{X}$	opt
$\boldsymbol{x}$ $(n \times 1)$	$\boldsymbol{W}^\top\boldsymbol{x} + \boldsymbol{b}$ $(k \times 1)$	$\boldsymbol{a}$ $(k \times 1)$	$\boldsymbol{\delta}$ $(k \times 1)$	$\boldsymbol{x}\boldsymbol{\delta}^\top$ $(n \times k)$	$\boldsymbol{\delta}$ $(k \times 1)$	$\boldsymbol{W}\boldsymbol{\delta}$ $(n \times 1)$	4
$\boldsymbol{X}_{\vert\vert\vert}$ $(n \times m)$	$\boldsymbol{W}^\top\boldsymbol{X} + \boldsymbol{b}$ $(k \times m)$	$\boldsymbol{A}$ $(k \times m)$	$\boldsymbol{\Delta}$ $(k \times m)$	$\frac{1}{m}\boldsymbol{X}\boldsymbol{\Delta}^\top$ $(n \times k)$	$\frac{1}{m}\boldsymbol{\Delta}\boldsymbol{1}_{m\times 1}$ $(k \times 1)$	$\boldsymbol{W}\boldsymbol{\Delta}$ $(n \times m)$	5
$\boldsymbol{X}_{\equiv}$ $(m \times n)$	$\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}$ $(m \times k)$	$\boldsymbol{A}$ $(m \times k)$	$\boldsymbol{\Delta}$ $(m \times k)$	$\frac{1}{m}\boldsymbol{X}^\top\boldsymbol{\Delta}$ $(n \times k)$	$\frac{1}{m}\boldsymbol{\Delta}^\top\boldsymbol{1}_{m\times 1}$ $(k \times 1)$	$\boldsymbol{\Delta}\boldsymbol{W}^\top$ $(m \times n)$	6


option 1
$\boldsymbol{x}, \vec{\boldsymbol{W}}$
option 2
$\boldsymbol{X}_{|||}, \vec{\boldsymbol{W}}$
option 3
$\boldsymbol{X}_{\equiv}, \vec{\boldsymbol{W}}$
option 4
$\boldsymbol{x}, \vec{\boldsymbol{W}}$
option 5
$\boldsymbol{X}_{|||}, \vec{\boldsymbol{W}}$
option 6
$\boldsymbol{X}_{\equiv}, \vec{\boldsymbol{W}}$

- option 1: single example, RL-convention $\overleftarrow{\boldsymbol{W}}$

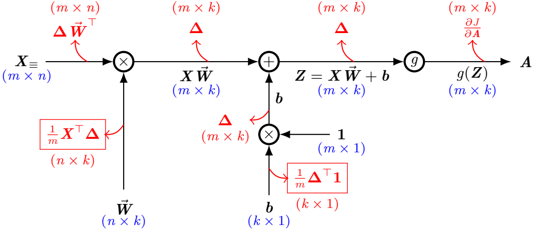


⋆ use in textbook

- algorithm 6.2
- algorithm 6.3

⋆ coursera[4]

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

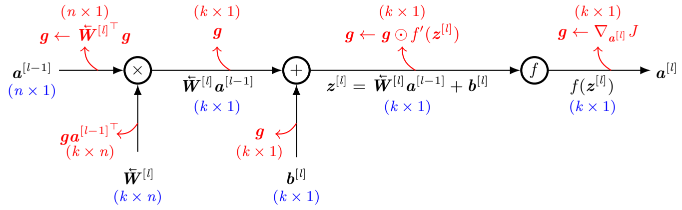


⋆ use in textbook

- sec 6.5.7

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

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



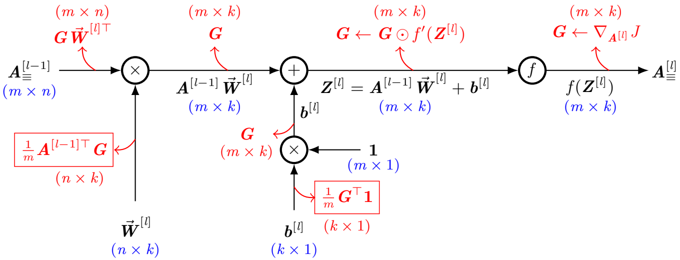
algorithm 1 forward computation

$\boldsymbol{a}^{[0]} = \boldsymbol{x}$
for $l = 1, \ldots, L$ **do**
    $\boldsymbol{z}^{[l]} = \overleftarrow{\boldsymbol{W}}^{[l]} \boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$
    $\boldsymbol{a}^{[l]} = f(\boldsymbol{z}^{[l]})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{a}^{[L]}$
$J = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\boldsymbol{\theta})$

algorithm 2 backward computation

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $l = L, L-1, \ldots, 1$ **do**
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{z}^{[l]}} J = \boldsymbol{g} \odot f'(\boldsymbol{z}^{[l]})$
    $\nabla_{\boldsymbol{b}^{[l]}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{[l]}} \Omega(\boldsymbol{\theta})$
    $\nabla_{\overleftarrow{\boldsymbol{W}}^{[l]}} J = \boldsymbol{g}\boldsymbol{a}^{[l-1]\top} + \lambda \nabla_{\overleftarrow{\boldsymbol{W}}^{[l]}} \Omega(\boldsymbol{\theta})$
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{[l-1]}} J = \overleftarrow{\boldsymbol{W}}^{[l]\top} \boldsymbol{g}$
end for

- algorithms using textbook notation (option 6, minibatch)



algorithm 3 forward computation

$\boldsymbol{A}^{[0]} = \boldsymbol{X}_{\equiv}$
for $l = 1, \ldots, L$ **do**
$\quad \boldsymbol{Z}^{[l]} = \boldsymbol{A}^{[l-1]}\,\vec{\boldsymbol{W}}^{[l]} + \boldsymbol{b}^{[l]}$
$\quad \boldsymbol{A}^{[l]} = f(\boldsymbol{Z}^{[l]})$
end for
$\hat{\boldsymbol{Y}} = \boldsymbol{A}^{[L]}$
$J = L(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) + \lambda\Omega(\boldsymbol{\theta})$

algorithm 4 backward computation

$\boldsymbol{G} \leftarrow \nabla_{\hat{\boldsymbol{Y}}} J = \nabla_{\hat{\boldsymbol{Y}}} L(\hat{\boldsymbol{Y}}, \boldsymbol{Y})$
for $l = L, L-1, \ldots, 1$ **do**
$\quad \boldsymbol{G} \leftarrow \nabla_{\boldsymbol{Z}^{[l]}} J = \boldsymbol{G} \odot f'(\boldsymbol{Z}^{[l]})$
$\quad \nabla_{\boldsymbol{b}^{[l]}} J = \frac{1}{m}\boldsymbol{G}^{\top}\boldsymbol{1} + \lambda\nabla_{\boldsymbol{b}^{[l]}}\Omega(\boldsymbol{\theta})$
$\quad \nabla_{\vec{\boldsymbol{W}}^{[l]}} J = \frac{1}{m} \qquad \boldsymbol{G} + \lambda\nabla_{\vec{\boldsymbol{W}}^{[l]}}\Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{G} \leftarrow \nabla_{\boldsymbol{A}^{[l-1]}} J = \boldsymbol{G}\,\vec{\boldsymbol{W}}^{[l]\top}$
end for

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

1: initialize all parameters $\boldsymbol{W}, \boldsymbol{b}$

2: **repeat**

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

4: forward: compute all activations $\boldsymbol{A}$

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

6: backward: compute all gradients

7: update parameters:

$$\begin{aligned} \boldsymbol{W} &\leftarrow \boldsymbol{W} - \epsilon \mathrm{d}\boldsymbol{W} && \text{(weights)} \\ \boldsymbol{b} &\leftarrow \boldsymbol{b} - \epsilon \mathrm{d}\boldsymbol{b} && \text{(bias)} \end{aligned}$$

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

9: return final parameters

$$\boldsymbol{W}^*, \boldsymbol{b}^*$$

Remarks

- complications of backprop in practice
  - multi-output operation
  - memory considerations
  - supporting diverse data types
  - handling undefined gradients
- field of <u>automatic</u> 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

Deep => hidden layer가 2개 이상인 것

Introduction

Feedfoward Networks

Deep Feedforward Networks

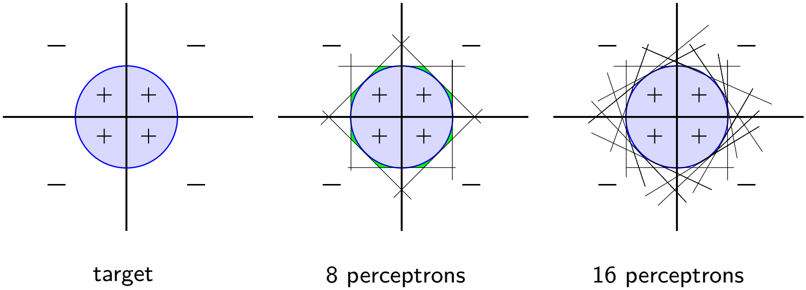
Summary

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
  - generalize 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 any[5] function (given enough hidden units)



(source: Abu-Mostafa)

- but the ability to learn that function: not guaranteed

[5] should 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 exponential 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 $\boldsymbol{v} \in \{0, 1\}^n$

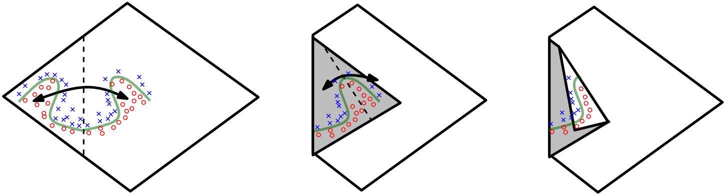
2^n bits required to select one such function

$\Rightarrow$ 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 <u>shallow</u> 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 $\underbrace{\text{our prior beliefs}}_{\uparrow}$

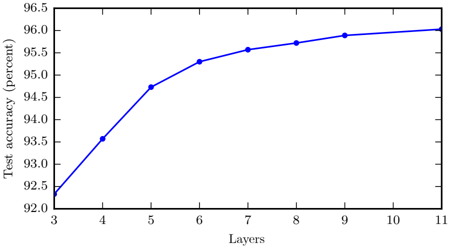
  about what kind of function the algorithm should learn

- choosing a deep model = encoding a very $\underbrace{\text{general belief}}_{\uparrow}$

  the function to learn should involve composition of simpler functions

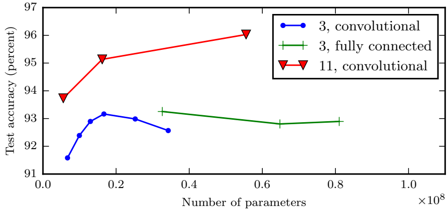
- empirically: $\boxed{\text{greater }^{\text{depth}} \Rightarrow \text{better generalization}}$
  - two examples on next page

- 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



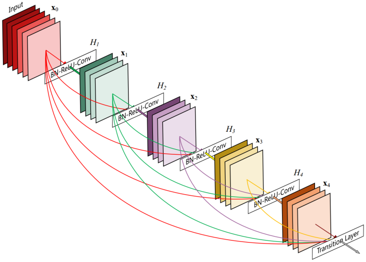
- shallow models overfit
  - at ∼20 million parameters
- deep ones can benefit
  - from having over 60 million

Other architectural considerations

- so far: neural nets as simple chains of layers
  - main considerations: depth of network and width 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.* skip 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 $\boldsymbol{W}$)
  - every input unit: connected to every output unit
- many specialized networks have fewer connections
  - each input unit: connected to only a small $\underline{\text{ subset }}$ of output units
- these strategies for reducing # of connections
  - reduce # of parameters/amount of computation to evaluate the net
  - but are highly problem-dependent $\rightarrow$ see later chapters

e.g. CNNs (ch 9): sparse connection patterns effective for vision problems

Outline

Introduction

Feedfoward Networks

Deep Feedforward Networks

Summary

Summary

- deep feed forward net: quintessential deep model
  - universal function approximator parameterized by $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b})$
  - learn $\boldsymbol{\theta}$ by gradient-based backprop algorithm
- building blocks of deep feedforward nets
  - neuron: modeled by logistic regression
  - forward function: propagates $\boldsymbol{x}$ to output, giving loss $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$
  - backward function: propagates $\mathtt{d}\boldsymbol{a}$ to input, giving $\mathtt{d}\boldsymbol{W}, \mathtt{d}\boldsymbol{b}$
  - update: $\mathtt{d}\boldsymbol{W} \leftarrow \mathtt{d}\boldsymbol{W} - \epsilon \mathtt{d}\boldsymbol{W}$, $\mathtt{d}\boldsymbol{b} \leftarrow \mathtt{d}\boldsymbol{b} - \epsilon \mathtt{d}\boldsymbol{b}$
  - 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

  $\Rightarrow$ architectural modifications in convolutional nets/recurrent nets