

Machine Learning

(機器學習)

Lecture 14: From Neural Net. to Deep Learning

Hsuan-Tien Lin (林軒田)

`htlin@csie.ntu.edu.tw`

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)



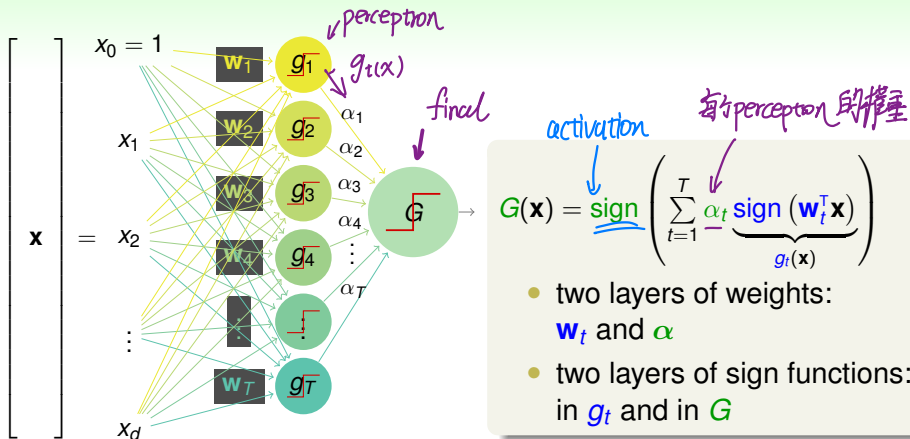
Roadmap

- 1 When Can Machines Learn?
- 2 Why Can Machines Learn?
- 3 How Can Machines Learn?
- 4 How Can Machines Learn Better?
- 5 Embedding Numerous Features: Kernel Models
- 6 Combining Predictive Features: Aggregation Models
- 7 Distilling Implicit Features: Extraction Models

Lecture 14: From Neural Net. to Deep Learning

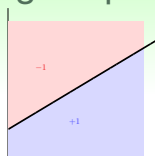
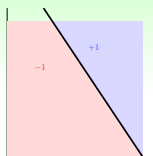
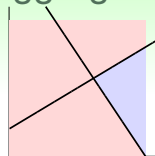
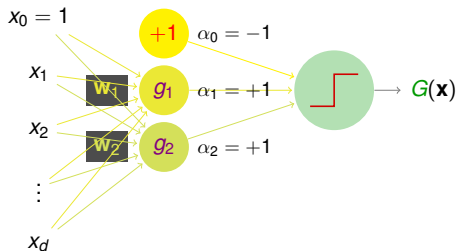
- Motivation
- Neural Network Hypothesis
- Neural Network Learning
- Deep Neural Network
- Vanishing Gradient Issue
- Modern Activation Functions

Linear Aggregation of Perceptrons: Pictorial View



what boundary can G implement?

Logic Operations with Aggregation

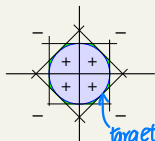

 g_1

 g_2

 $\text{AND}(g_1, g_2)$


$$G(\mathbf{x}) = \text{sign}(-1 + g_1(\mathbf{x}) + g_2(\mathbf{x}))$$

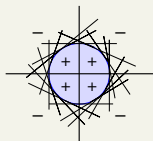
- $g_1(\mathbf{x}) = g_2(\mathbf{x}) = +1$ (TRUE):
 $G(\mathbf{x}) = +1$ (TRUE)
- otherwise:
 $G(\mathbf{x}) = -1$ (FALSE)
- $G \equiv \text{AND}(g_1, g_2)$

OR, NOT can be **similarly implemented**

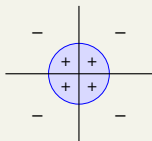
Powerfulness and Limitation



8 perceptrons

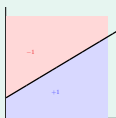


16 perceptrons

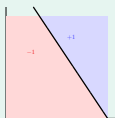


target boundary

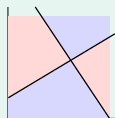
- 'convex set' hypotheses implemented: $d_{VC} \rightarrow \infty$, remember? :-)
- powerfulness: enough perceptrons \approx smooth boundary



g_1



g_2



$\text{XOR}(g_1, g_2)$

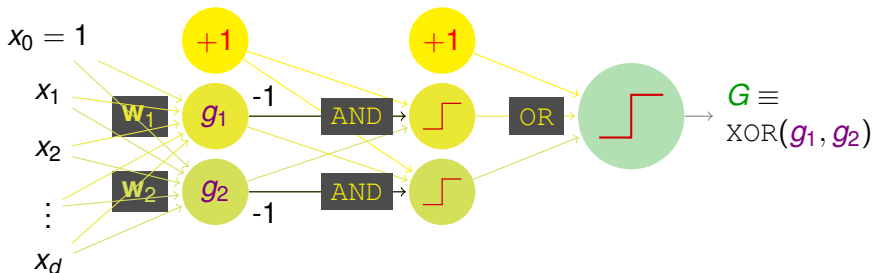
- limitation: XOR not 'linear separable' under $\phi(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}))$

how to implement $\text{XOR}(g_1, g_2)$?

Multi-Layer Perceptrons: Basic Neural Network

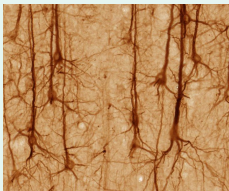
- non-separable data: can use more **transform**
- how about one more layer of AND transform?

$$\text{XOR}(g_1, g_2) = \text{OR}(\text{AND}(-g_1, g_2), \text{AND}(g_1, -g_2))$$



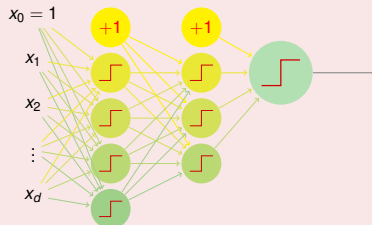
perceptron (simple)
 \Rightarrow aggregation of perceptrons (powerful)
 \Rightarrow multi-layer perceptrons (more powerful)

Connection to Biological Neurons



by UC Regents Davis campus-brainmaps.org.

Licensed under CC BY 3.0 via Wikimedia Commons



by Lauris Rubenis.
Licensed under CC BY
2.0 via
<https://flic.kr/p/fkVuZX>

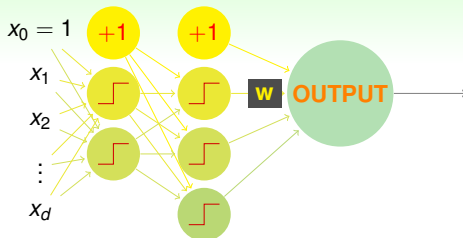


by Pedro Ribeiro
Simões. Licensed
under CC BY 2.0 via
<https://flic.kr/p/adiv7b>

neural network: **bio-inspired** model

Questions?

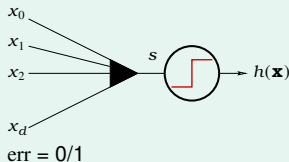
Neural Network Hypothesis: Output



- **OUTPUT**: simply a **linear model** with $\mathbf{s} = \mathbf{w}^T \phi^{(2)}(\phi^{(1)}(\mathbf{x}))$
- any linear model can be used—**remember? :-)**

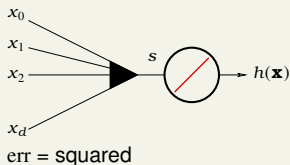
linear classification

$$h(\mathbf{x}) = \text{sign}(\mathbf{s})$$



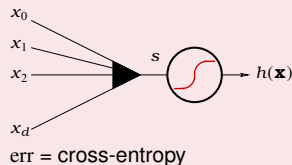
linear regression

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



logistic regression

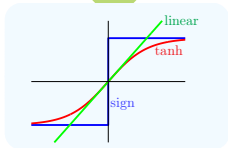
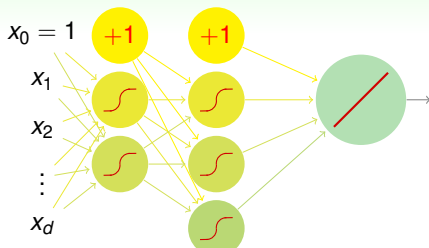
$$h(\mathbf{x}) = \theta(\mathbf{s})$$



will discuss **'regression' with squared error**

Neural Network Hypothesis: Transformation

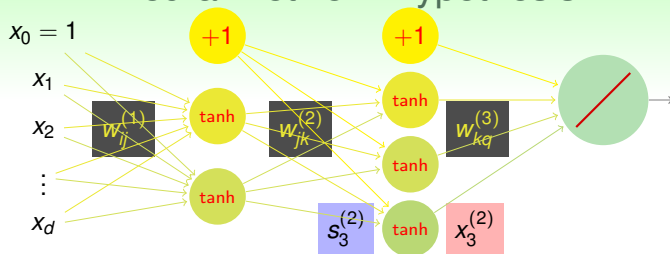
- \lceil : **transformation** function of score (signal) s
- **any transformation?**
 - \diagup : whole network linear & thus **less useful**
 - \lceil : discrete & thus **hard to optimize** for \mathbf{w}
- traditional choice of **transformation**: $\frown = \tanh(s)$
 - ‘analog’ approximation of \lceil : **easier to optimize**
 - somewhat **closer to biological** neuron
 - **not that new!** :-)



$$\begin{aligned}\tanh(s) &= \frac{\exp(s) - \exp(-s)}{\exp(s) + \exp(-s)} \\ &= 2\theta(2s) - 1\end{aligned}$$

will discuss with **tanh** first

Neural Network Hypothesis



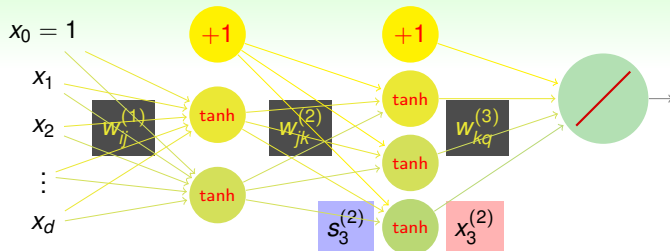
$d^{(0)}-d^{(1)}-d^{(2)}-\dots-d^{(L)}$ Neural Network (NNet)

$$w_{ij}^{(\ell)} : \begin{cases} 1 \leq \ell \leq L & \text{layers} \\ 0 \leq i \leq d^{(\ell-1)} & \text{inputs} \\ 1 \leq j \leq d^{(\ell)} & \text{outputs} \end{cases}, \text{ score } s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)},$$

$$\text{transformed } x_j^{(\ell)} = \begin{cases} \tanh(s_j^{(\ell)}) & \text{if } \ell < L \\ s_j^{(\ell)} & \text{if } \ell = L \end{cases}$$

apply \mathbf{x} as **input layer** $\mathbf{x}^{(0)}$, go through **hidden layers** to get $\mathbf{x}^{(\ell)}$, predict at **output layer** $x_1^{(L)}$

Physical Interpretation



- each layer: **transformation** to be **learned** from data

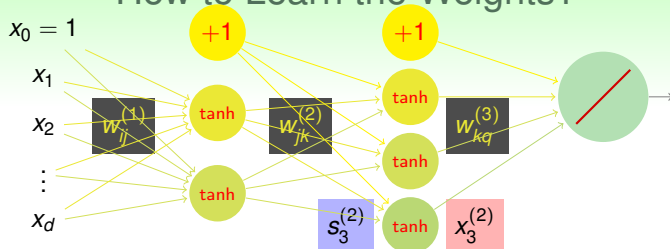
- $\phi^{(\ell)}(\mathbf{x}) = \tanh \left(\begin{bmatrix} \sum_{i=0}^{d^{(\ell-1)}} w_{i1}^{(\ell)} x_i^{(\ell-1)} \\ \vdots \end{bmatrix} \right)$

—whether \mathbf{x} ‘matches’ weight vectors in pattern

NNet: **pattern extraction** with
layers of connection weights

Questions?

How to Learn the Weights?



- goal: learning all $\{w_{ij}^{(\ell)}\}$ to **minimize** $E_{\text{in}}(\{w_{ij}^{(\ell)}\})$
- one hidden layer: simply **aggregation of perceptrons**
—**gradient boosting** to determine hidden neuron one by one
- multiple hidden layers? **not easy**
- let $e_n = (y_n - \text{NNet}(\mathbf{x}_n))^2$:
can apply **(stochastic) GD** after computing $\frac{\partial e_n}{\partial w_{ij}^{(\ell)}}$!

next: efficient computation of $\frac{\partial e_n}{\partial w_{ij}^{(\ell)}}$

Computing $\frac{\partial e_n}{\partial w_{i1}^{(L)}}$ (Output Layer)

$$e_n = (y_n - \text{NNet}(\mathbf{x}_n))^2 = (y_n - s_1^{(L)})^2 = \left(y_n - \sum_{i=0}^{d^{(L-1)}} w_{i1}^{(L)} x_i^{(L-1)} \right)^2$$

specially (output layer)
 $(0 \leq i \leq d^{(L-1)})$

$$\begin{aligned} & \frac{\partial e_n}{\partial w_{i1}^{(L)}} \\ &= \frac{\partial e_n}{\partial s_1^{(L)}} \cdot \frac{\partial s_1^{(L)}}{\partial w_{i1}^{(L)}} \\ &= -2 (y_n - s_1^{(L)}) \cdot (x_i^{(L-1)}) \end{aligned}$$

generally $(1 \leq \ell < L)$
 $(0 \leq i \leq d^{(\ell-1)}; 1 \leq j \leq d^{(\ell)})$

$$\begin{aligned} & \frac{\partial e_n}{\partial w_{ij}^{(\ell)}} \\ &= \frac{\partial e_n}{\partial s_j^{(\ell)}} \cdot \frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \\ &= \delta_j^{(\ell)} \cdot (x_i^{(\ell-1)}) \end{aligned}$$

$$\delta_1^{(L)} = -2 (y_n - s_1^{(L)}) , \text{ how about others?}$$

Computing $\delta_j^{(\ell)} = \frac{\partial e_n}{\partial s_j^{(\ell)}}$

$$s_j^{(\ell)} \xRightarrow{\tanh} x_j^{(\ell)} \xRightarrow{w_{jk}^{(\ell+1)}} \begin{bmatrix} s_1^{(\ell+1)} \\ \vdots \\ s_k^{(\ell+1)} \\ \vdots \end{bmatrix} \Rightarrow \dots \Rightarrow e_n$$

$$\begin{aligned} \delta_j^{(\ell)} = \frac{\partial e_n}{\partial s_j^{(\ell)}} &= \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial e_n}{\partial s_k^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} \\ &= \sum_k \left(\delta_k^{(\ell+1)} \right) \left(w_{jk}^{(\ell+1)} \right) \left(\tanh' \left(s_j^{(\ell)} \right) \right) \end{aligned}$$

$\delta_j^{(\ell)}$ can be computed **backwards** from $\delta_k^{(\ell+1)}$

Backpropagation (Backprop) Algorithm

Backprop on NNet

initialize all weights $w_{ij}^{(\ell)}$

for $t = 0, 1, \dots, T$

- 1 stochastic: randomly pick $n \in \{1, 2, \dots, N\}$
- 2 forward: compute all $x_i^{(\ell)}$ with $\mathbf{x}^{(0)} = \mathbf{x}_n$
- 3 backward: compute all $\delta_j^{(\ell)}$ subject to $\mathbf{x}^{(0)} = \mathbf{x}_n$
- 4 gradient descent: $w_{ij}^{(\ell)} \leftarrow w_{ij}^{(\ell)} - \eta x_i^{(\ell-1)} \delta_j^{(\ell)}$

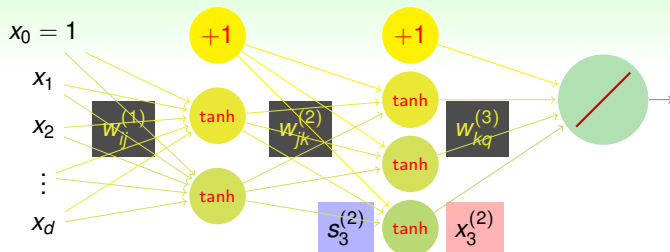
return $g_{\text{NNET}}(\mathbf{x}) = \left(\dots \tanh \left(\sum_j w_{jk}^{(2)} \cdot \tanh \left(\sum_i w_{ij}^{(1)} x_i \right) \right) \right)$

sometimes (1) to (3) is (parallelly) done many times and average($x_i^{(\ell-1)} \delta_j^{(\ell)}$) taken for update in (4), called **mini-batch**

basic NNet algorithm: backprop to compute the gradient **efficiently**

Questions?

Physical Interpretation of NNet Revisited



- each layer: **pattern feature extracted** from data, **remember? :-)**
- how many neurons? how many layers?
—more generally, **what structure?**
 - subjectively, **your design!**
 - objectively, **validation, maybe?**

structural decisions:
key issue for applying NNet

Shallow versus Deep Neural Networks

shallow: few (hidden) layers; deep: many layers

Shallow NNet

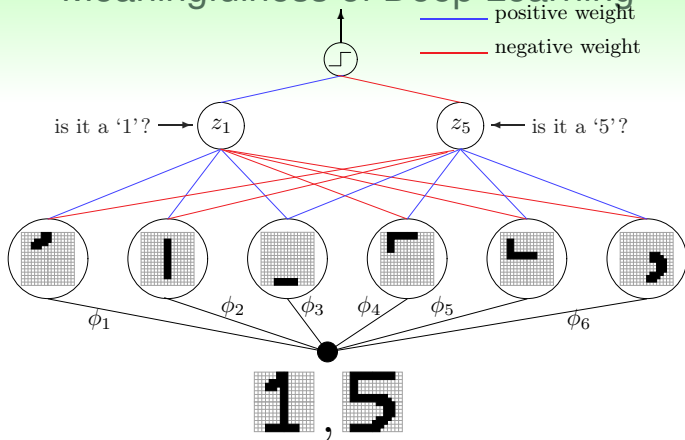
- more **efficient** to train (○)
- **simpler** structural decisions (○)
- theoretically **powerful enough** (○)

Deep NNet

- **challenging** to train (×)
- **sophisticated** structural decisions (×)
- **'arbitrarily' powerful** (○)
- more **'meaningful'?** (see next slide)

deep NNet (**deep learning**)
gaining attention in recent years

Meaningfulness of Deep Learning



⚙️ **'less burden'** for each layer: **simple** to **complex** features

- natural for **difficult** learning task with **raw features**, like **vision**

deep NNet: currently popular in
vision/speech/...

Challenges and Key Techniques for Deep Learning

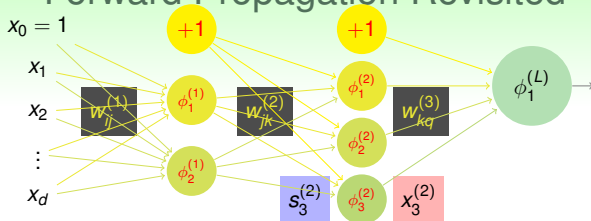
- difficult **structural decisions**:
 - subjective with **domain knowledge**: like **convNet** for images
- high **model complexity**:
 - no big worries if **big enough data**
 - **regularization** towards noise-tolerant: like
 - **dropout** (tolerant when network corrupted)
 - **denoising** (tolerant when input corrupted)
- hard **optimization problem**:
 - **change optimization landscape**
 - **careful initialization**
 - **higher-order optimization** beyond SGD/backprop
- huge **computational complexity** (worsen with **big data**):
 - novel hardware/architecture: like **mini-batch with GPU**

In My Humble opinion.

IMHO, careful **regularization** and **optimization** are key techniques

Questions?

Forward Propagation Revisited

 $d^{(0)}-d^{(1)}-d^{(2)}-\dots-d^{(L)}$ Neural Network (NNet)

$w_{ij}^{(\ell)}$: $(1 + d^{(\ell-1)}) \times (d^{(\ell)})$ matrix before layer ℓ

$$\text{score } s_j^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} \cdot x_i^{(\ell-1)}, \text{ transformed } x_j^{(\ell)} = \phi_j^{(\ell)} \left(s_j^{(\ell)} \right)$$

$\phi_j^{(\ell)}$: transformation (activation) function, e.g.

$\phi_1^{(L)} = \text{linear}$ for regression;

$\phi_j^{(\ell)} = \tanh$ for traditional NNet

Backpropagation Revisited

$$\begin{aligned}
 \delta_j^{(\ell)} &= \frac{\partial e_n}{\partial s_j^{(\ell)}} \\
 &= \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial e_n}{\partial s_k^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} \\
 &= \sum_k \left(\delta_k^{(\ell+1)} \right) \left(w_{jk}^{(\ell+1)} \right) \left(\phi' \left(s_j^{(\ell)} \right) \right) \\
 &= \sum_k \left(\sum_m \left(\delta_m^{(\ell+2)} \right) \left(w_{km}^{(\ell+2)} \right) \phi' \left(s_k^{(\ell+1)} \right) \right) \left(w_{jk}^{(\ell+1)} \right) \left(\phi' \left(s_j^{(\ell)} \right) \right)
 \end{aligned}$$

$\delta_i^{(1)}$: $\delta_1^{(L)}$ multiplied by many $w_{??}^{(\ell+1)}$ and $\phi'(s_?^{(\ell)})$
 for $\ell = 1, 2, \dots, L-1$

The Vanishing Gradient Issue

$$\text{gradient } \nabla_{ij}^{(\ell)} = x_i^{(\ell-1)} \cdot \delta_j^{(\ell)} = x_i^{(\ell-1)} \sum \sum \sum \delta_1^{(L)} w w w \phi' \phi' \phi'$$

when $\phi = \tanh \Rightarrow x_i^{(\ell-1)} \in (-1, 1)$

- $\phi(s) \rightarrow \pm 1$ when $s \rightarrow \pm\infty$: saturation
- $\phi'(s) = 1 - \tanh^2(s)$
 - $\rightarrow 0$ when $s \rightarrow \pm\infty$
- vanishing gradient: $w_{??}^{(\ell)}$ too big $\Rightarrow |s|$ too big
 $\Rightarrow \phi'(s)$ too small $\Rightarrow \delta_{?}^{(1)}$ too small $\Rightarrow \nabla_{??}^{(1)}$ too small

vanishing gradient: early weights not updated
 \Rightarrow cannot train 'deep' network

Possible Cures for Vanishing Gradient

- better-behaved network
 - skip connection (escape some $w\phi'$)
- better-behaved weights
 - small-random initialization (to be discussed)
- better-behaved network + better-behaved weights
 - layer-wise pre-training (see MLTech Lecture 213)
- better-behaved (hidden) inputs
 - internal normalization (scale $x_j^{(\ell)}$)
- better-behaved gradient
 - gradient normalization (to be discussed)
- better-behaved activation functions (to be discussed)

vanishing/varying gradients: difficulty of deep learning optimization

Questions?

Rectified Linear Unit

$$\phi(s) = \max(s, 0)$$

- **Rectified Linear Unit (ReLU):**
 $\phi(s) = s$ for $s > 0$, 0 for $s = 0$, 0 for $s < 0$
—continuous
- $\phi'(s) = 1$ for $s > 0$, 0 for $s < 0$
—**less gradient vanishing** (\approx ‘half’ of the time)
- $\phi'(s) = \text{undefined}$ for $s = 0$
—floating point 0.0 hardly encountered, replace by ‘sub-gradient’ usually okay
- **sparse** network per example
- **fast** arithmetic computation

ReLU (with or without some tanh): arguably the most widely-used for deep learning

Dead Neuron Issue

$$\phi(s) = \max(s, 0)$$

- $s < 0$: $\phi(s) = 0$ and $\phi'(s) = 0$ per example
- $s < 0$ for every example (dead neuron) if
 - very negative $w_{0?}^{(\ell)}$ (e.g. update from a very big gradient step)
 - all positive $x_i^{(0)}$ (e.g. images without shifting) + negative weights $w_{ij}^{(1)}$

dead neurons worrisome (but not always serious)

Leaky Rectified Linear Unit

$$\phi(s) = \max(s, 0.01s) = \max(s, 0) + 0.01 \min(s, 0)$$

- $s > 0$: $\phi(s) = s$ and $\phi'(s) = 1$ per example
- $s < 0$: $\phi(s) = 0.01s$ and $\phi'(s) = 0.01$ per example

less likely to have dead neurons, but why 0.01?

Parametric Rectified Linear Unit

$$\phi(\alpha, \mathbf{s}) = \max(\mathbf{s}, \alpha \cdot \mathbf{s})$$

- ReLU: $\alpha = 0$, Leaky ReLU: fixed α
- optimizable α :

$$\begin{aligned} \frac{\partial \mathbf{e}_n}{\partial \alpha_j^{(\ell)}} &= \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial \mathbf{e}_n}{\partial \mathbf{s}_k^{(\ell+1)}} \frac{\partial \mathbf{s}_k^{(\ell+1)}}{\partial \mathbf{x}_j^{(\ell)}} \frac{\partial \mathbf{x}_j^{(\ell)}}{\partial \alpha_j^{(\ell)}} \\ &= \sum_k \left(\delta_k^{(\ell+1)} \right) \left(\mathbf{w}_{jk}^{(\ell+1)} \right) \left(\frac{\partial \phi(\alpha_j^{(\ell)}, \mathbf{s}_j^{(\ell)})}{\partial \alpha_j^{(\ell)}} \right) \end{aligned}$$

with $\frac{\partial \phi(\alpha, \mathbf{s})}{\partial \alpha} = \mathbf{s}$ if $\alpha \mathbf{s} > \mathbf{s}$, or 0 otherwise.

‘power’ of deep learning: anything (loosely)
differentiable is ‘learnable’

Questions?

Summary

1 Distilling Implicit Features: Extraction Models

Lecture 14: From Neural Net. to Deep Learning

- Motivation

multi-layer for power with biological inspirations

- Neural Network Hypothesis

layered pattern extraction until linear hypothesis

- Neural Network Learning

backprop to compute gradient efficiently

- Deep Neural Network

difficult hierarchical feature extraction problem

- Vanishing Gradient Issue

saturated neuron cannot backprop error signal

- Modern Activation Functions

ReLU extensions with fewer saturating ends