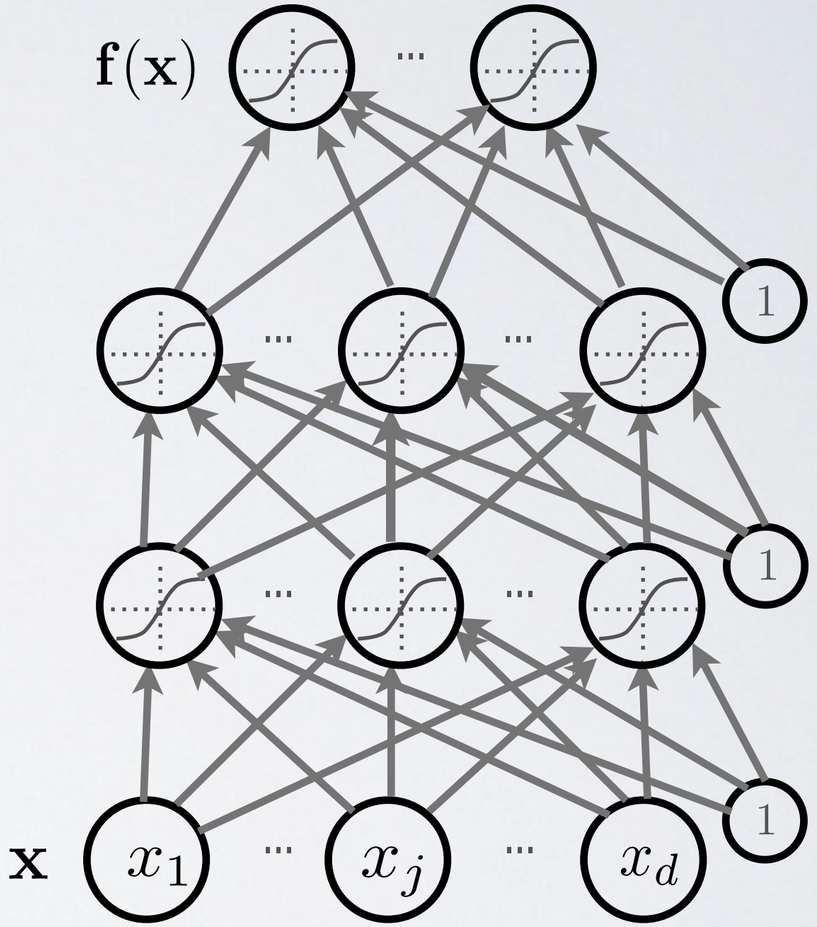
Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

FEEDFORWARD NEURAL NETWORK

- What we'll cover
  - how neural networks take input $\mathbf{x}$ and make predict $\mathbf{f}(\mathbf{x})$
    - forward propagation
    - types of units
    - capacity of neural networks
  - how to train neural nets (classifiers) on data
    - loss function
    - parameter gradient computation with backpropagation
    - gradient descent algorithms
  - deep learning
    - dropout
    - batch normalization
    - unsupervised pre-training



Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

Plan

- forward propagation (compute output)
- backpropagation (compute gradients)

lunch

- complete training algorithm
- deep learning

Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

Plan

- **forward propagation** (compute output)
- backpropagation (compute gradients)

lunch

- complete training algorithm
- deep learning

Neural networks

Feedforward neural network - artificial neuron

ARTIFICIAL NEURON

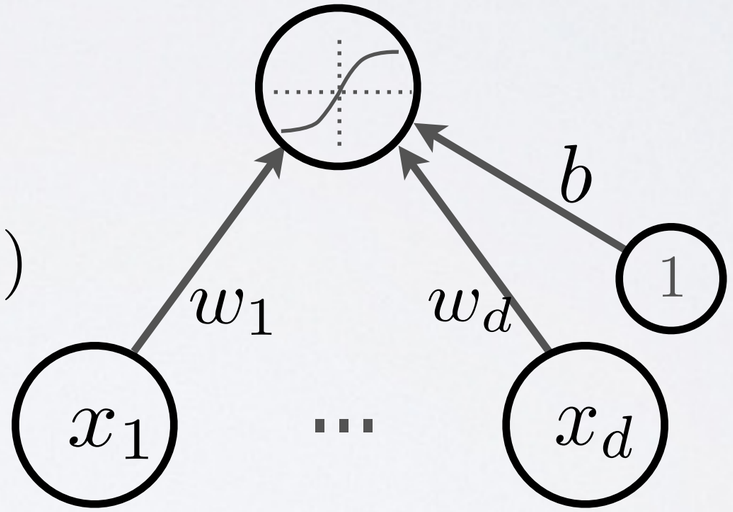
Topics: connection weights, bias, activation function

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron (output) activation

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$



$\mathbf{w}$ are the connection weights

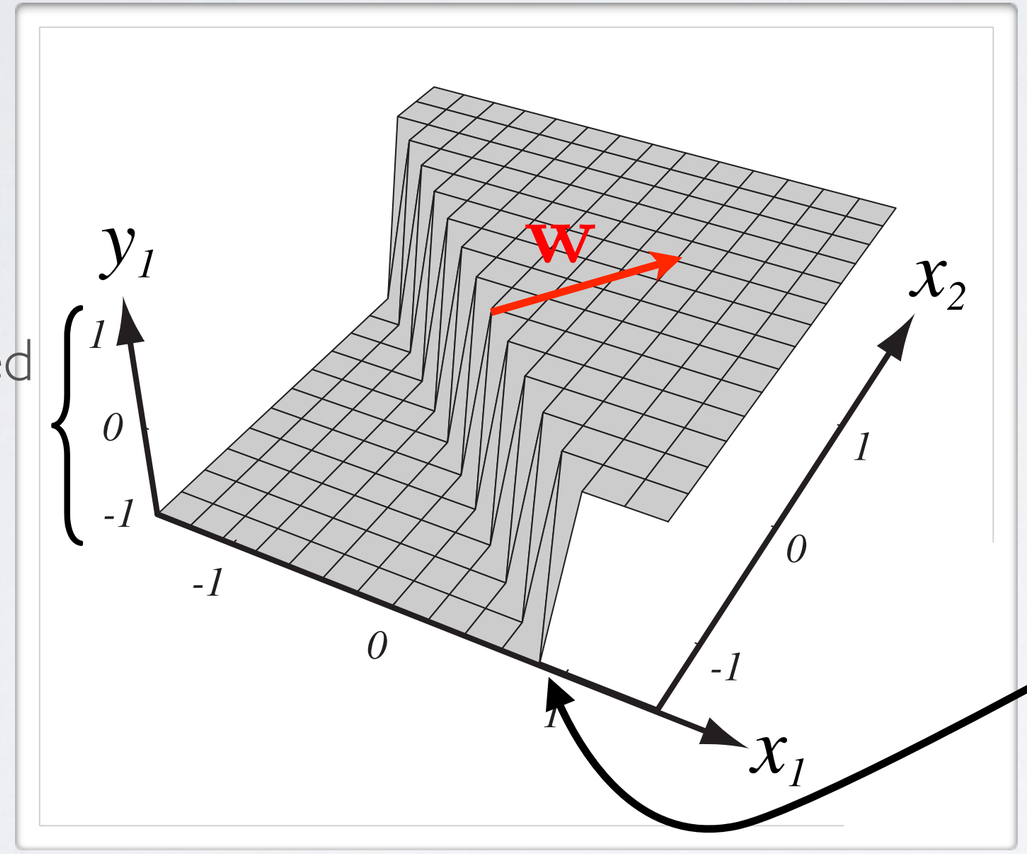
b is the neuron bias

$g(\cdot)$ is called the activation function

ARTIFICIAL NEURON

Topics: connection weights, bias, activation function

range determined by $g(\cdot)$



bias b only changes the position of the riff

(from Pascal Vincent's slides)

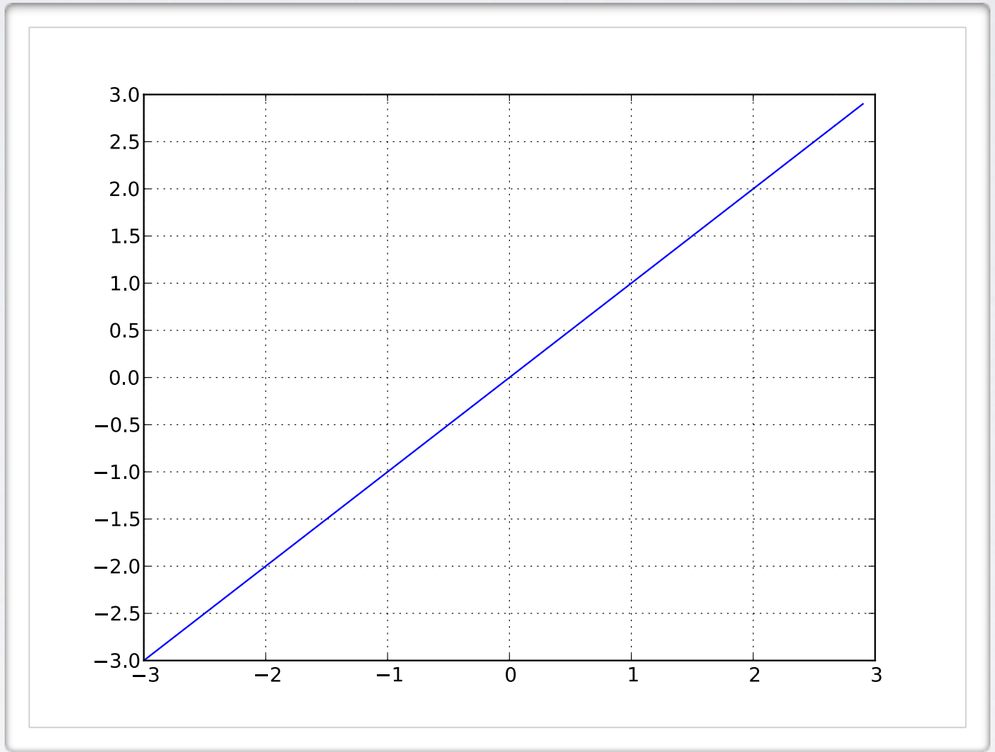
Neural networks

Feedforward neural network - activation function

ACTIVATION FUNCTION

Topics: linear activation function

- Performs no input squashing
- Not very interesting...

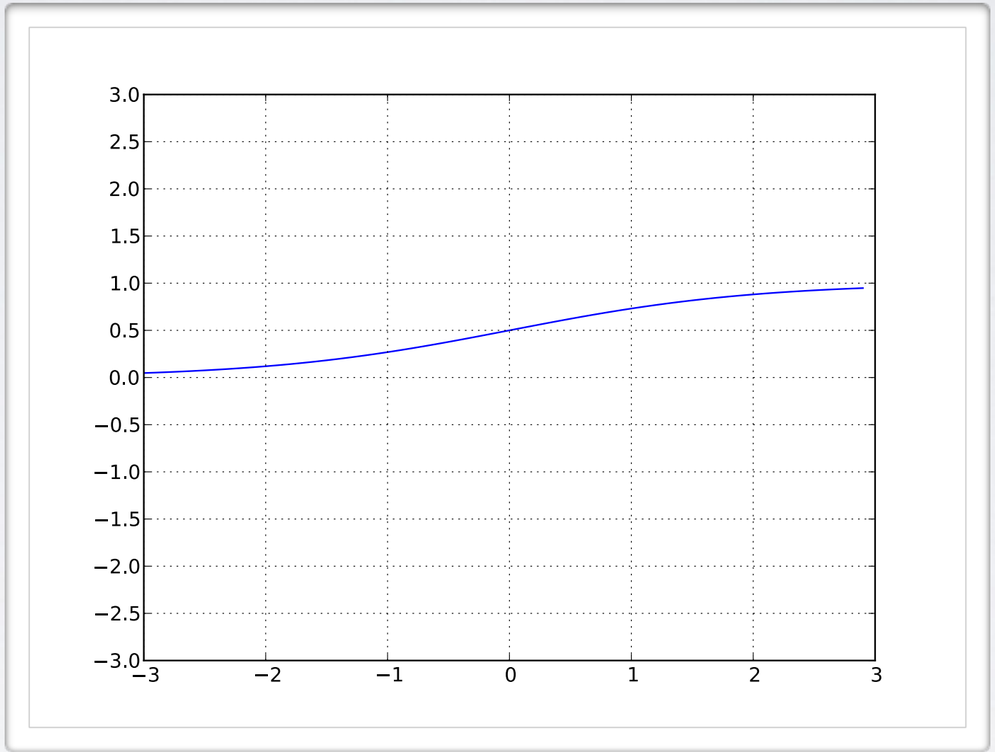


$$g(a) = a$$

ACTIVATION FUNCTION

Topics: sigmoid activation function

- Squashes the neuron's pre-activation between 0 and 1
- Always positive
- Bounded
- Strictly increasing

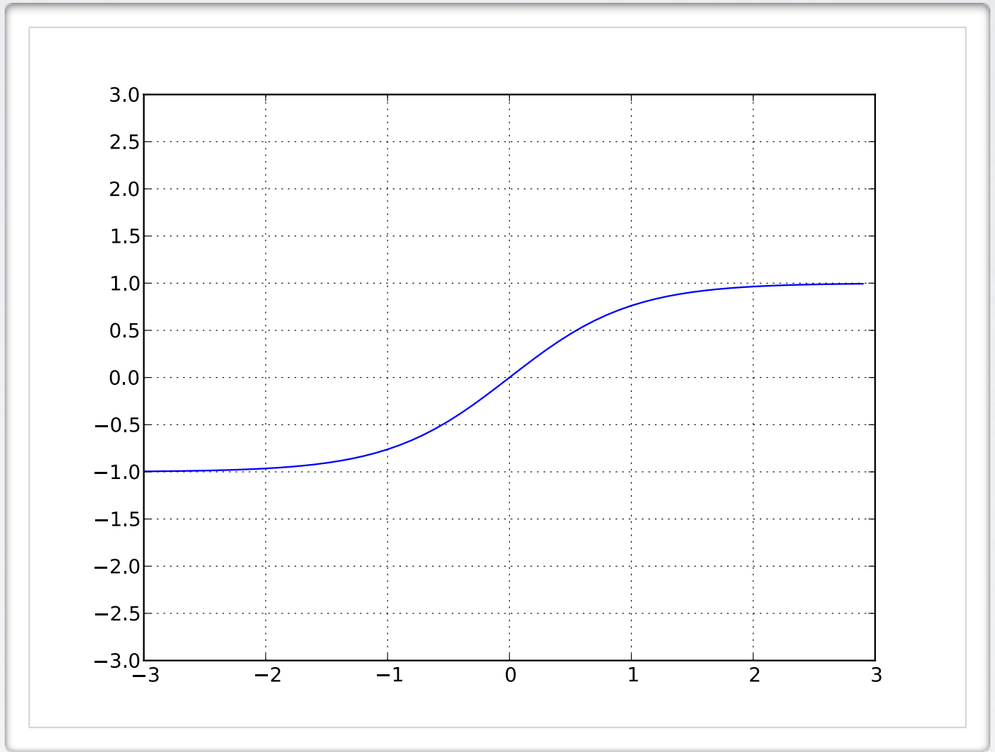


$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

ACTIVATION FUNCTION

Topics: hyperbolic tangent ("tanh") activation function

- Squashes the neuron's pre-activation between -1 and 1
- Can be positive or negative
- Bounded
- Strictly increasing

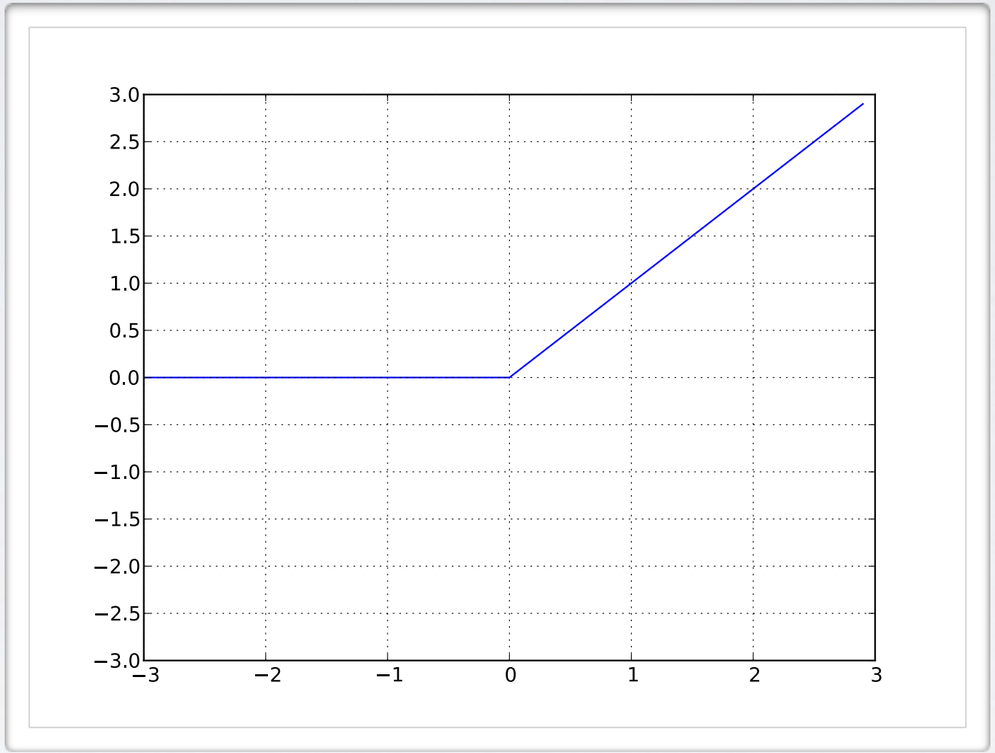


$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

ACTIVATION FUNCTION

Topics: rectified linear activation function

- Bounded below by 0 (always non-negative)
- Not upper bounded
- Strictly increasing
- Tends to give neurons with sparse activities



$$g(a) = \text{reclin}(a) = \max(0, a)$$

Neural networks

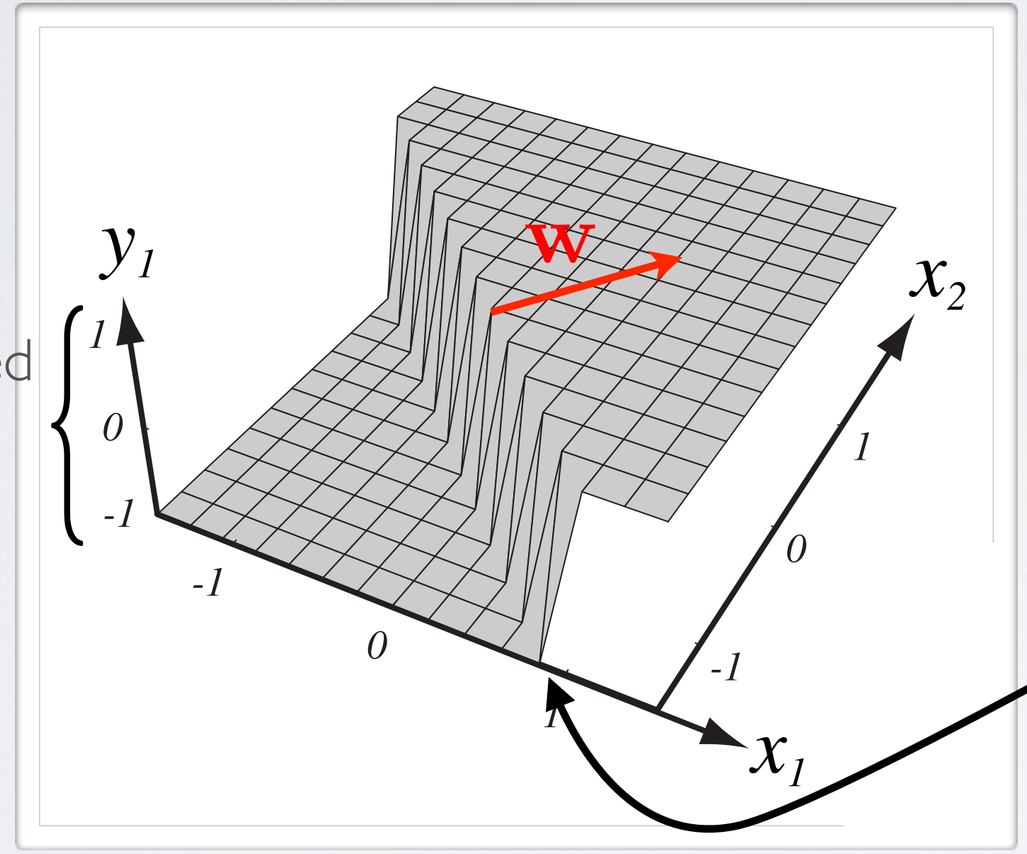
Feedforward neural network - capacity of single neuron

ARTIFICIAL NEURON

REMINDER

Topics: connection weights, bias, activation function

range determined by $g(\cdot)$



bias b only changes the position of the riff

(from Pascal Vincent's slides)

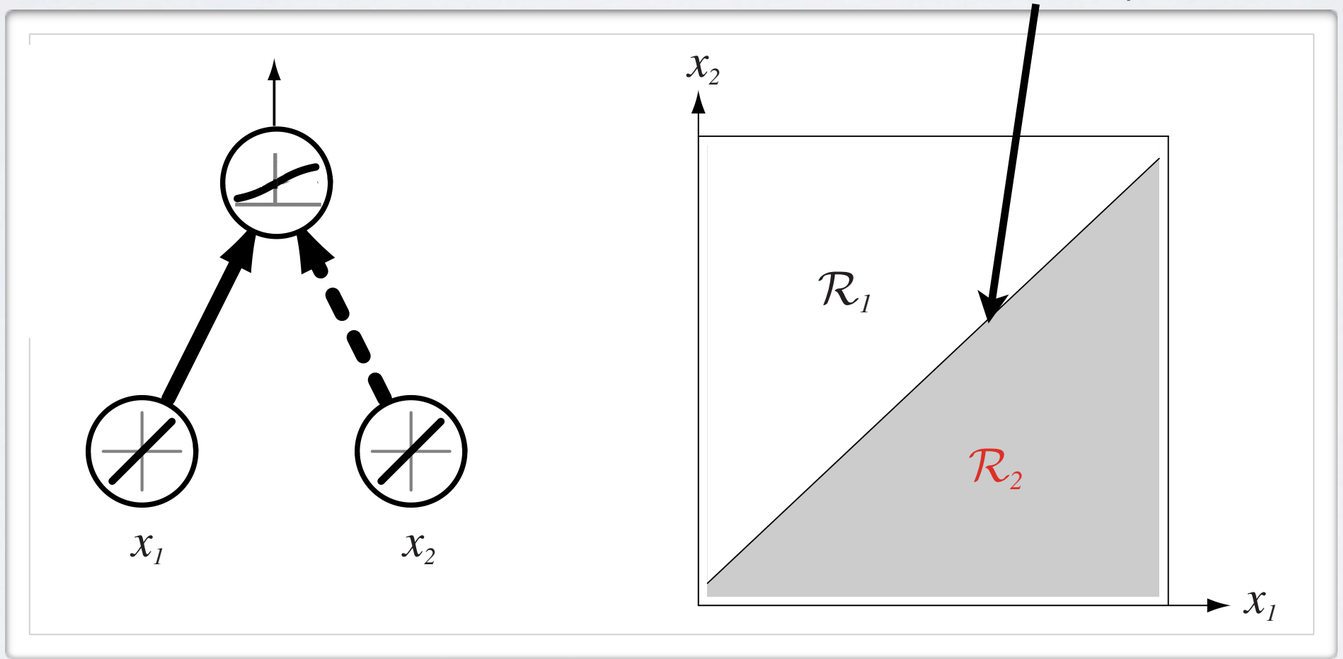
ARTIFICIAL NEURON

Topics: capacity, decision boundary of neuron

- Could do binary classification:
  - with sigmoid, can interpret neuron as estimating $p(y = 1|\mathbf{x})$
  - also known as logistic regression classifier
  - if greater than 0.5, predict class 1
  - otherwise, predict class 0

(similar idea can apply with tanh)

decision boundary is linear

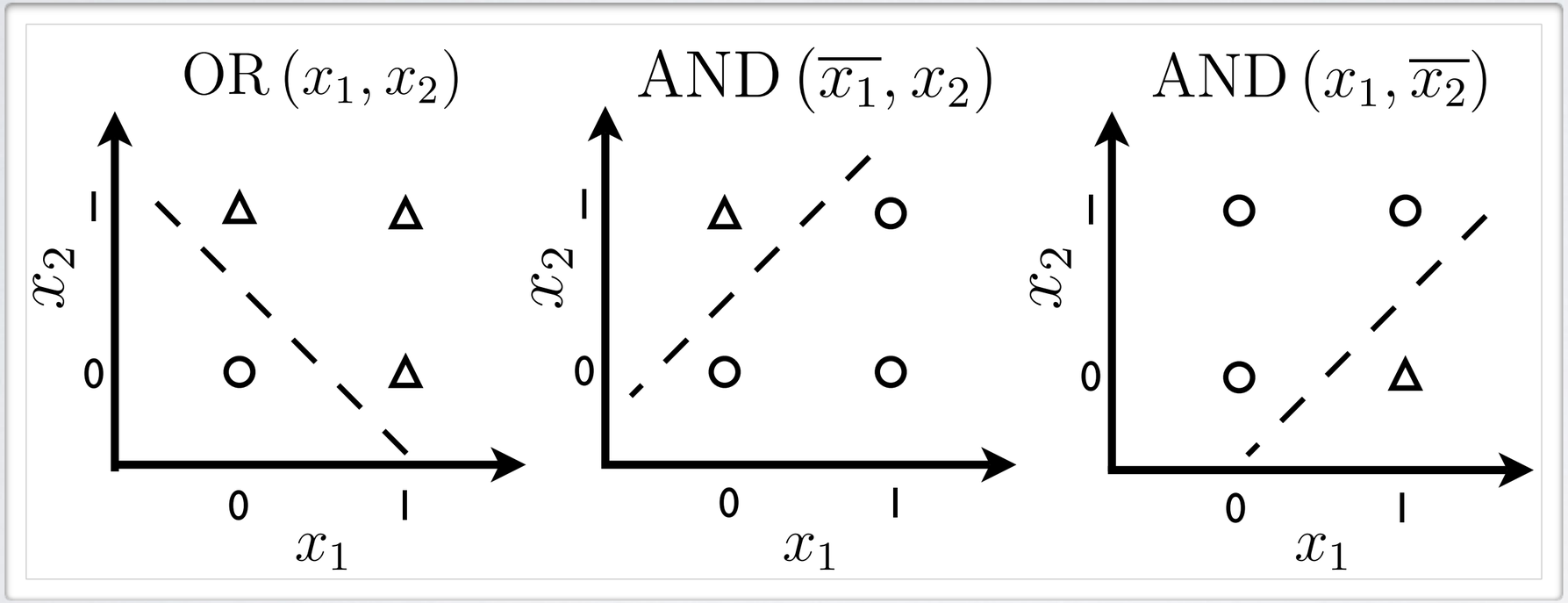


(from Pascal Vincent's slides)

ARTIFICIAL NEURON

Topics: capacity of single neuron

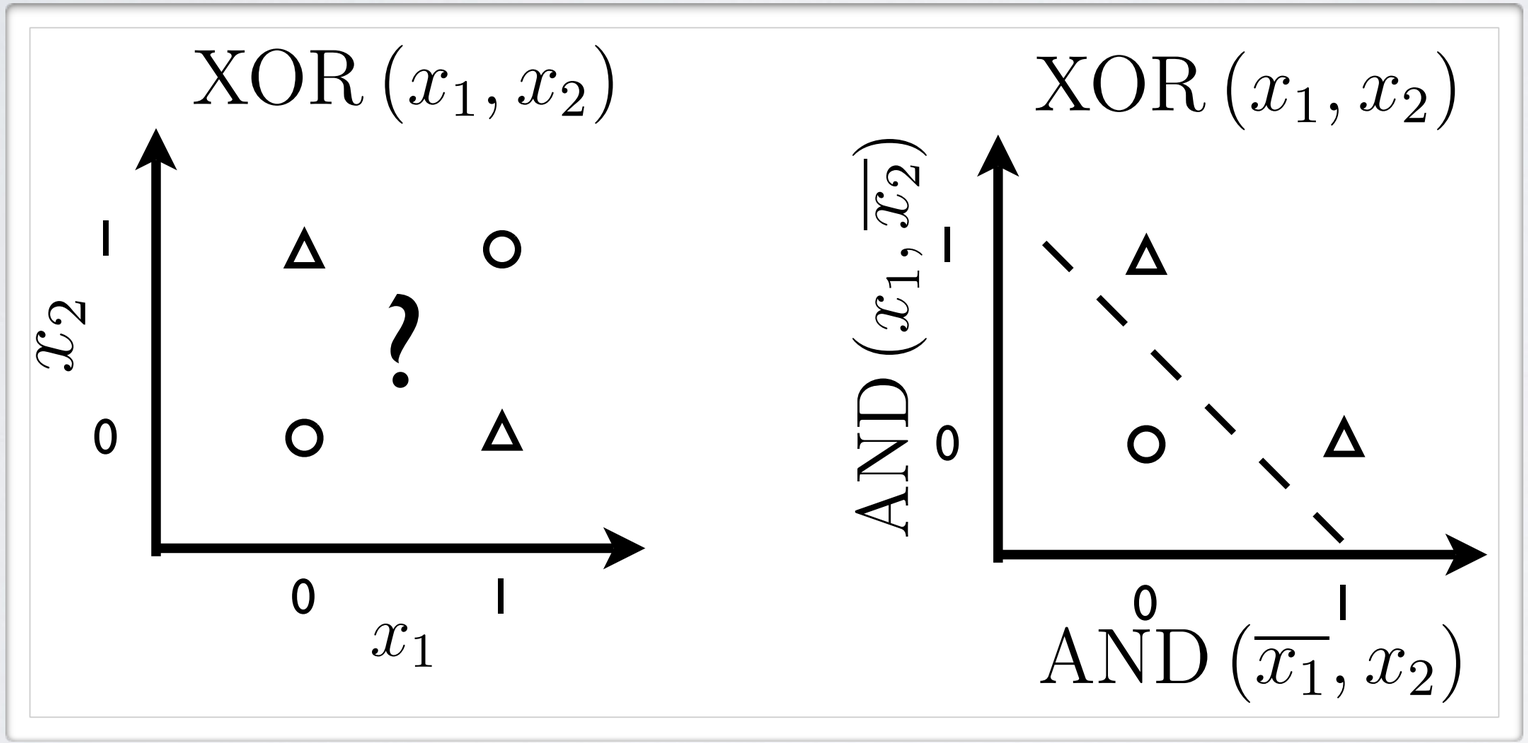
- Can solve linearly separable problems



ARTIFICIAL NEURON

Topics: capacity of single neuron

- Can't solve non linearly separable problems...



- ... unless the input is transformed in a better representation

Neural networks

Feedforward neural network - multilayer neural network

NEURAL NETWORK

Topics: single hidden layer neural network

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$\left(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j\right)$$

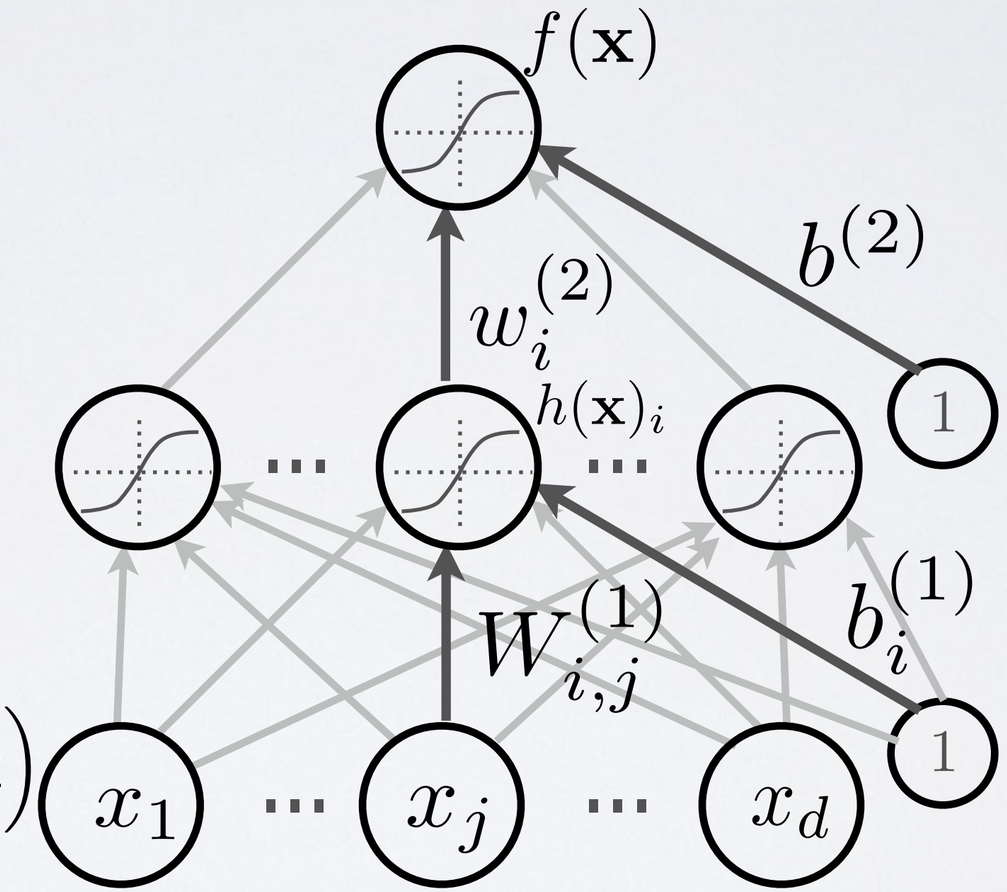
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)\top}\mathbf{h}^{(1)}\mathbf{x}\right)$$

output activation function



NEURAL NETWORK

Topics: softmax activation function

- For multi-class classification:
  - we need multiple outputs (1 output per class)
  - we would like to estimate the conditional probability $p(y = c|\mathbf{x})$

- We use the softmax activation function at the output:

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$

  - strictly positive
  - sums to one

- Predicted class is the one with highest estimated probability

NEURAL NETWORK

Topics: multilayer neural network

- Could have L hidden layers:
  - layer pre-activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

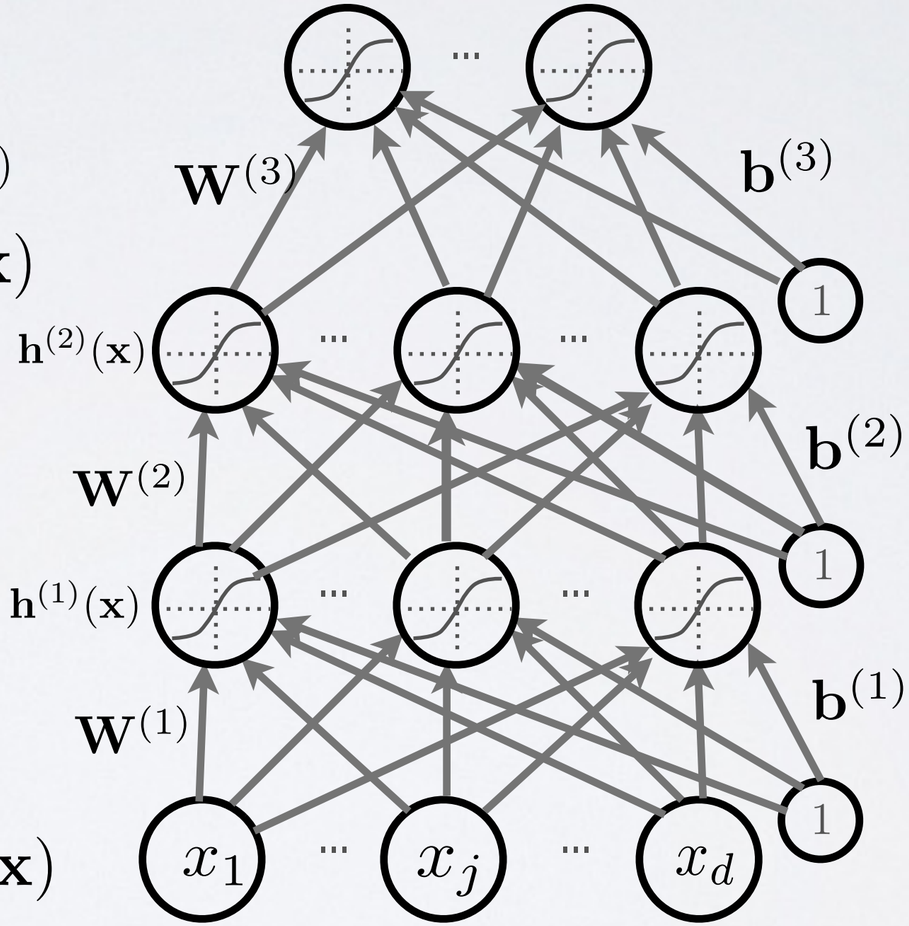
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$

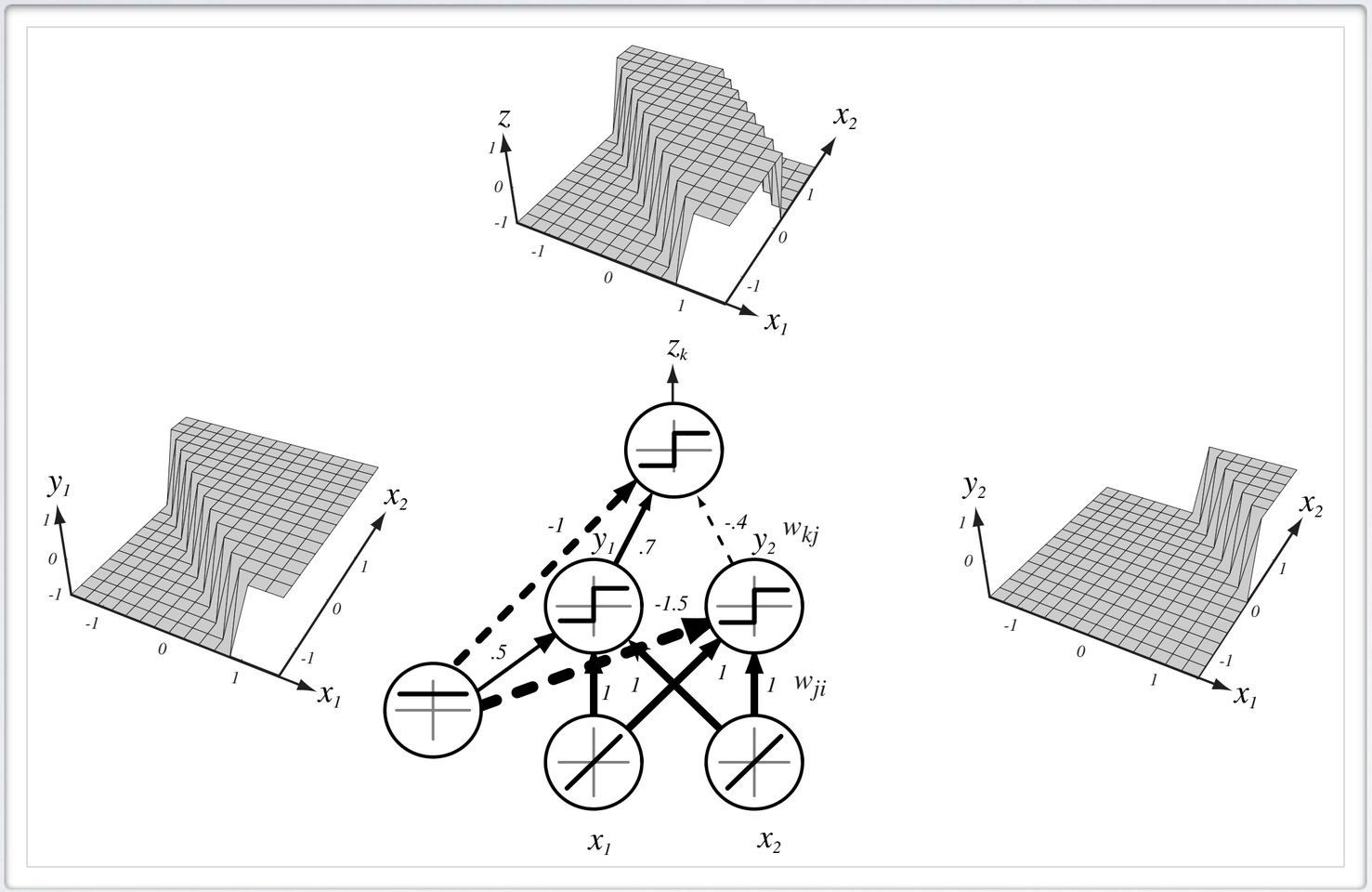


Neural networks

Feedforward neural network - capacity of neural network

CAPACITY OF NEURAL NETWORK

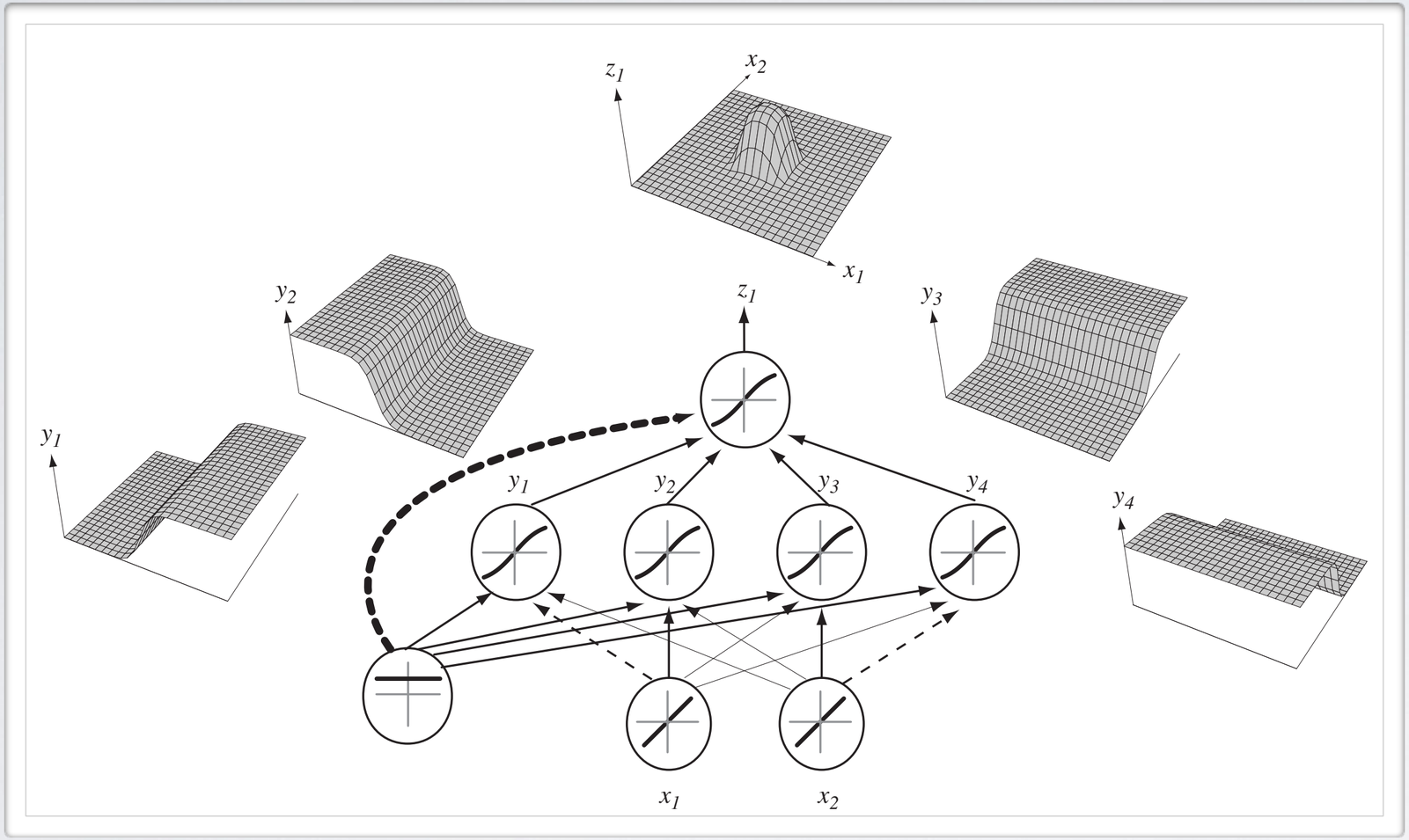
Topics: single hidden layer neural network



(from Pascal Vincent's slides)

CAPACITY OF NEURAL NETWORK

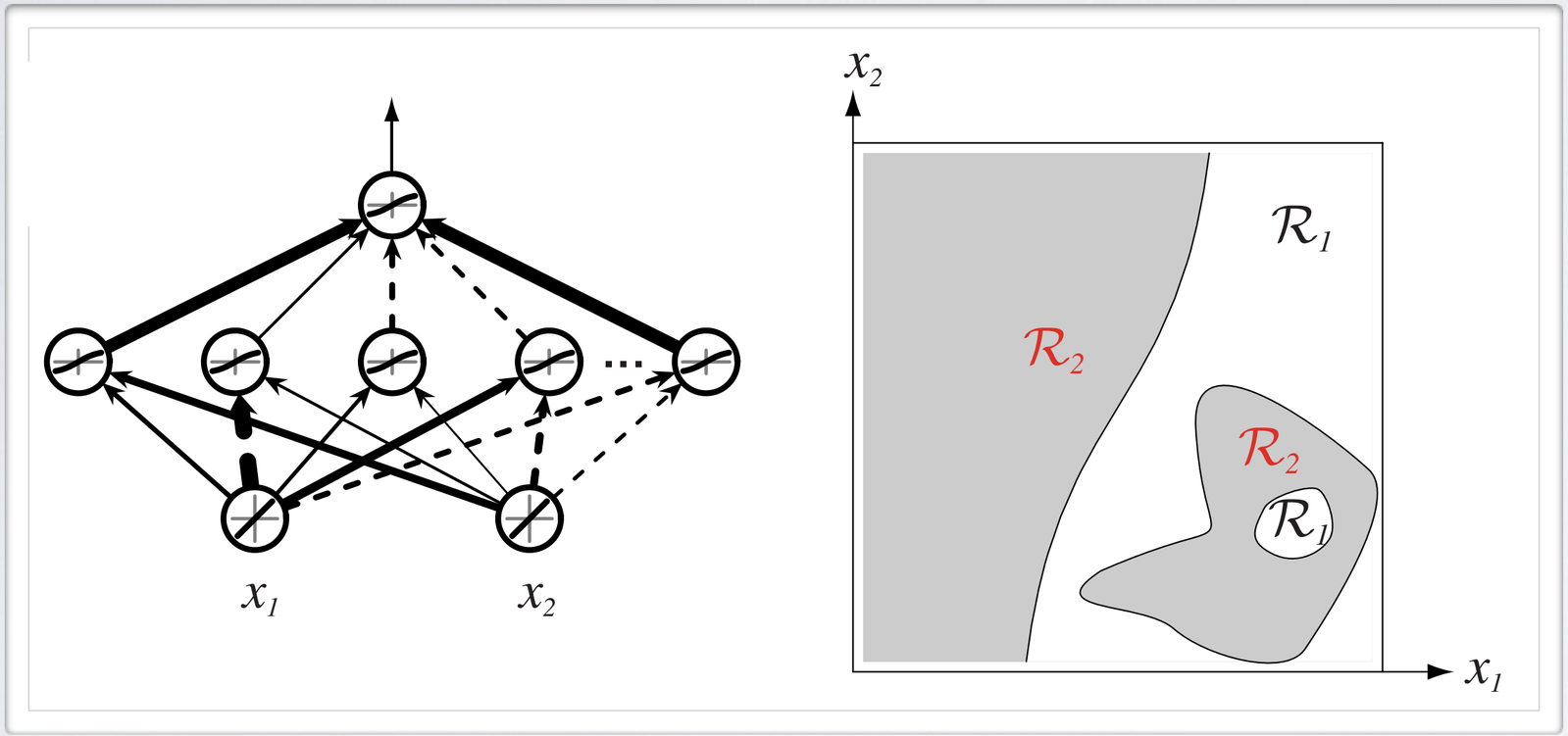
Topics: single hidden layer neural network



(from Pascal Vincent's slides)

CAPACITY OF NEURAL NETWORK

Topics: single hidden layer neural network



(from Pascal Vincent's slides)

CAPACITY OF NEURAL NETWORK

Topics: universal approximation

- Universal approximation theorem (Hornik, 1991):
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"
- The result applies for sigmoid, tanh and many other hidden layer activation functions
- This is a good result, but it doesn't mean there is a learning algorithm that can find the necessary parameter values!

Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

Plan

- forward propagation (compute output)
- **backpropagation** (compute gradients)

lunch

- complete training algorithm
- deep learning

Neural networks

Training neural networks - empirical risk minimization

NEURAL NETWORK

REMINDER

Topics: multilayer neural network

- Could have L hidden layers:

- layer input pre-activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

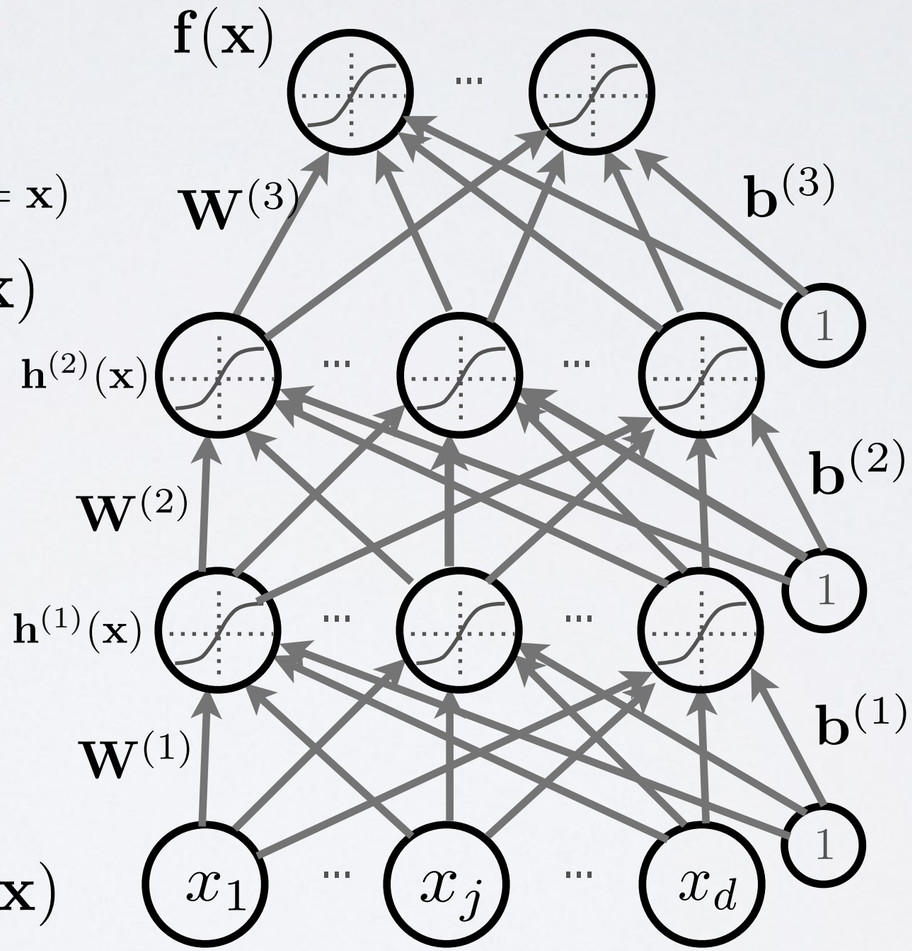
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



MACHINE LEARNING

Topics: empirical risk minimization, regularization

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function
  - $\Omega(\boldsymbol{\theta})$ is a regularizer (penalizes certain values of $\boldsymbol{\theta}$)
- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

MACHINE LEARNING

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

Neural networks

Training neural networks - loss function

LOSS FUNCTION

Topics: loss function for classification

- Neural network estimates $f(\mathbf{x})_c = p(y = c|\mathbf{x})$
  - we could maximize the probabilities of $y^{(t)}$ given $\mathbf{x}^{(t)}$ in the training set

- To frame as minimization, we minimize the negative log-likelihood

natural log (ln)

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$

  - we take the log to simplify for numerical stability and math simplicity
  - sometimes referred to as cross-entropy

Neural networks

Training neural networks - output layer gradient

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

GRADIENT COMPUTATION

Topics: loss gradient at output

- Partial derivative:

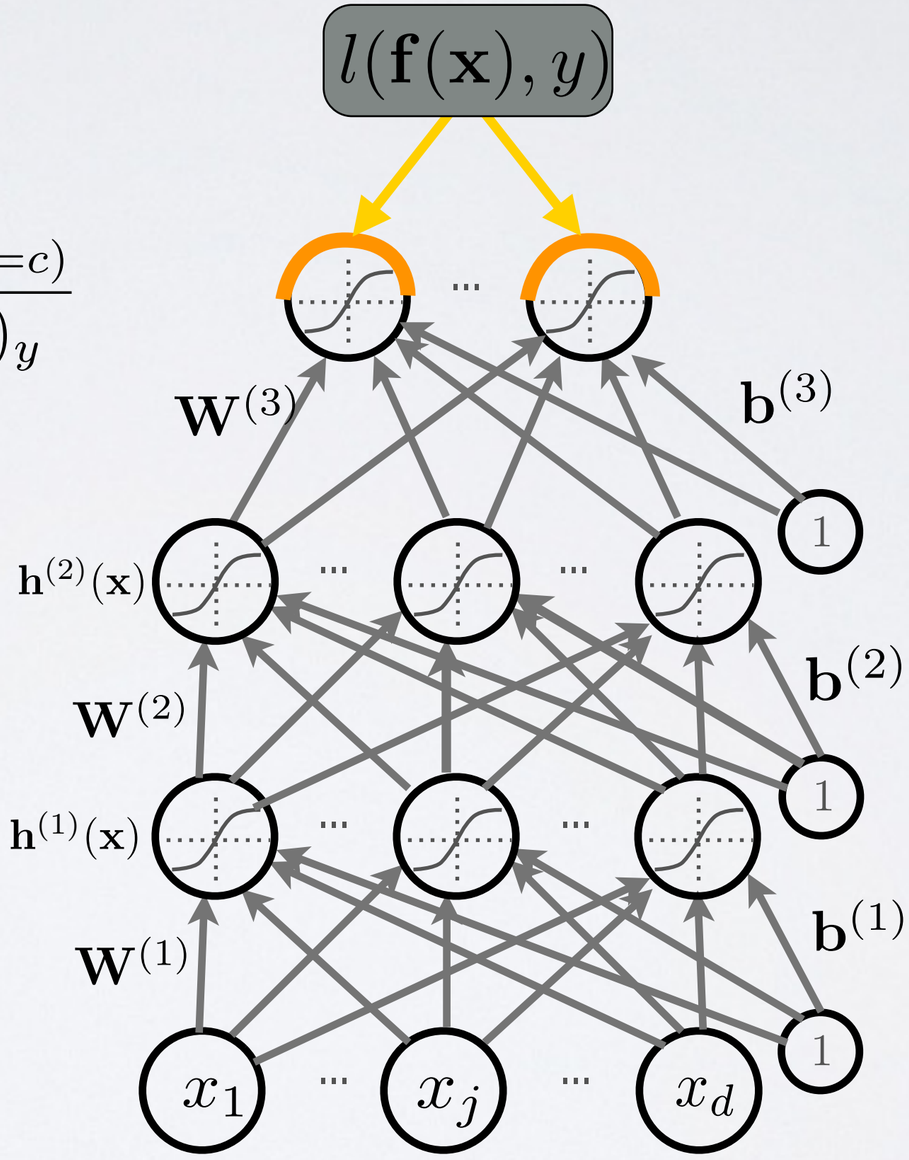
$$\frac{\partial}{\partial f(\mathbf{x})_c} - \log f(\mathbf{x})_y = \frac{-1_{(y=c)}}{f(\mathbf{x})_y}$$

- Gradient:

$$\nabla_{\mathbf{f}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \frac{-1}{f(\mathbf{x})_y} \begin{bmatrix} 1_{(y=0)} \\ \vdots \\ 1_{(y=C-1)} \end{bmatrix}$$

$$= \frac{-\mathbf{e}(y)}{f(\mathbf{x})_y}$$



GRADIENT COMPUTATION

Topics: loss gradient at output pre-activation

- Partial derivative:

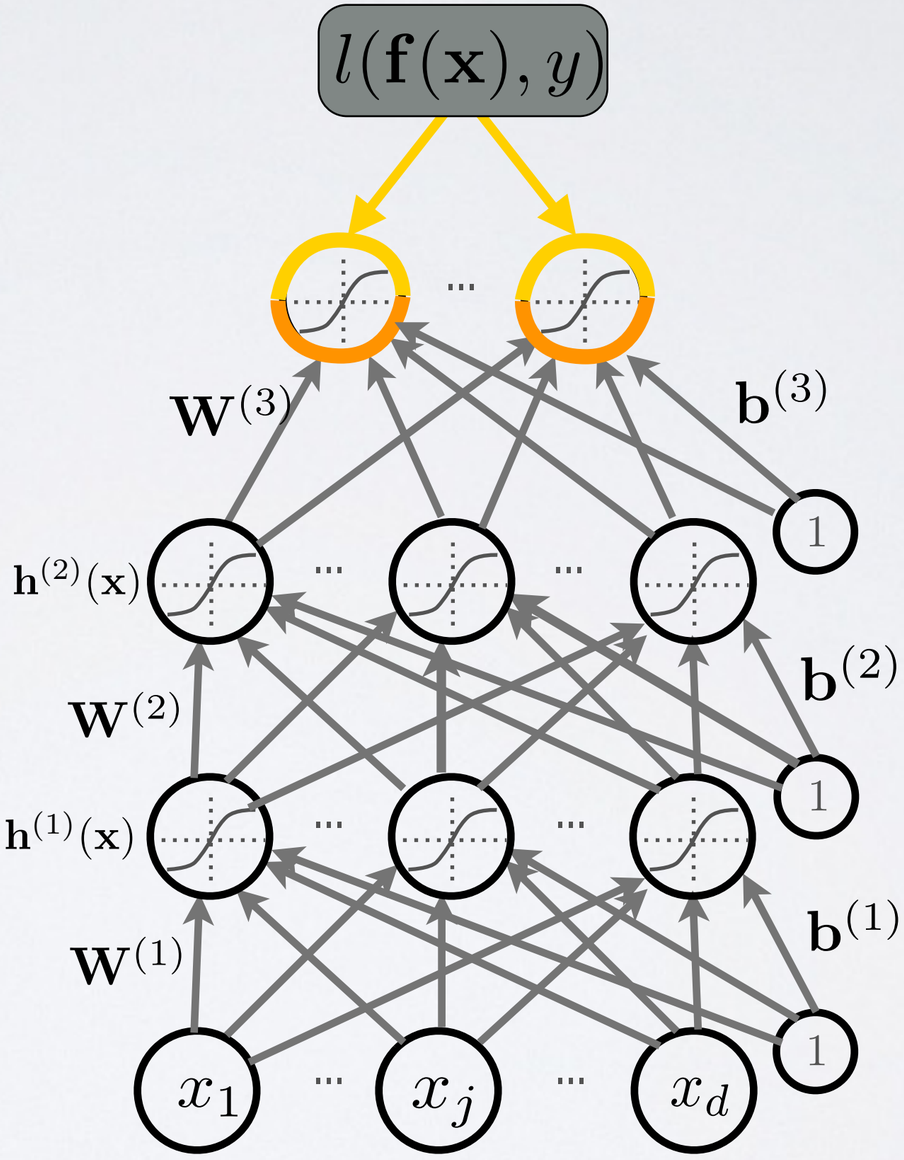
$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y$$

$$= \quad -\left(1_{(y=c)} - f(\mathbf{x})_c\right)$$

- Gradient:

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \quad -\left(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})\right)$$



$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y$$

$$\frac{\partial \frac{g(x)}{h(x)}}{\partial x} = \frac{\partial g(x)}{\partial x}\frac{1}{h(x)} - \frac{g(x)}{h(x)^2}\frac{\partial h(x)}{\partial x}$$

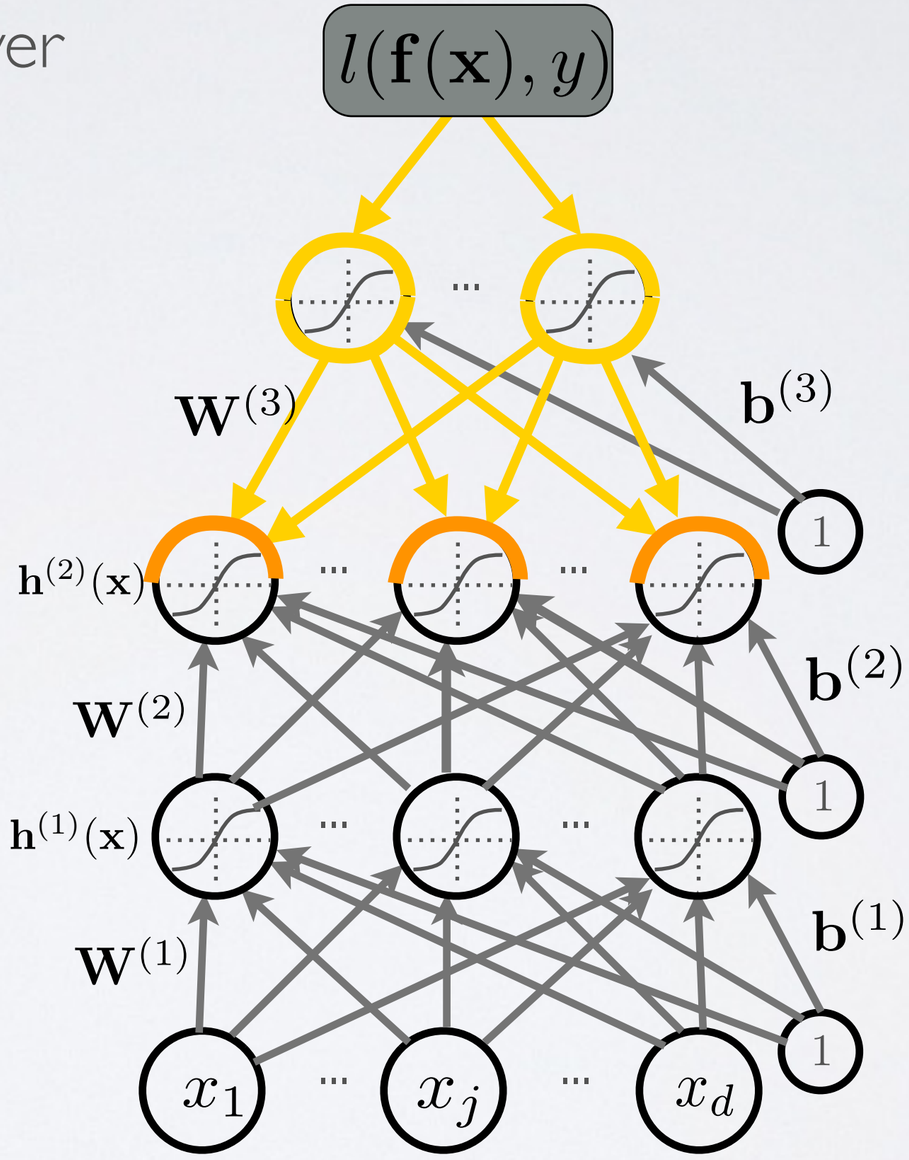
Neural networks

Training neural networks - hidden layer gradient

GRADIENT COMPUTATION

Topics: loss gradient at hidden layer

- ... this is getting complicated!!



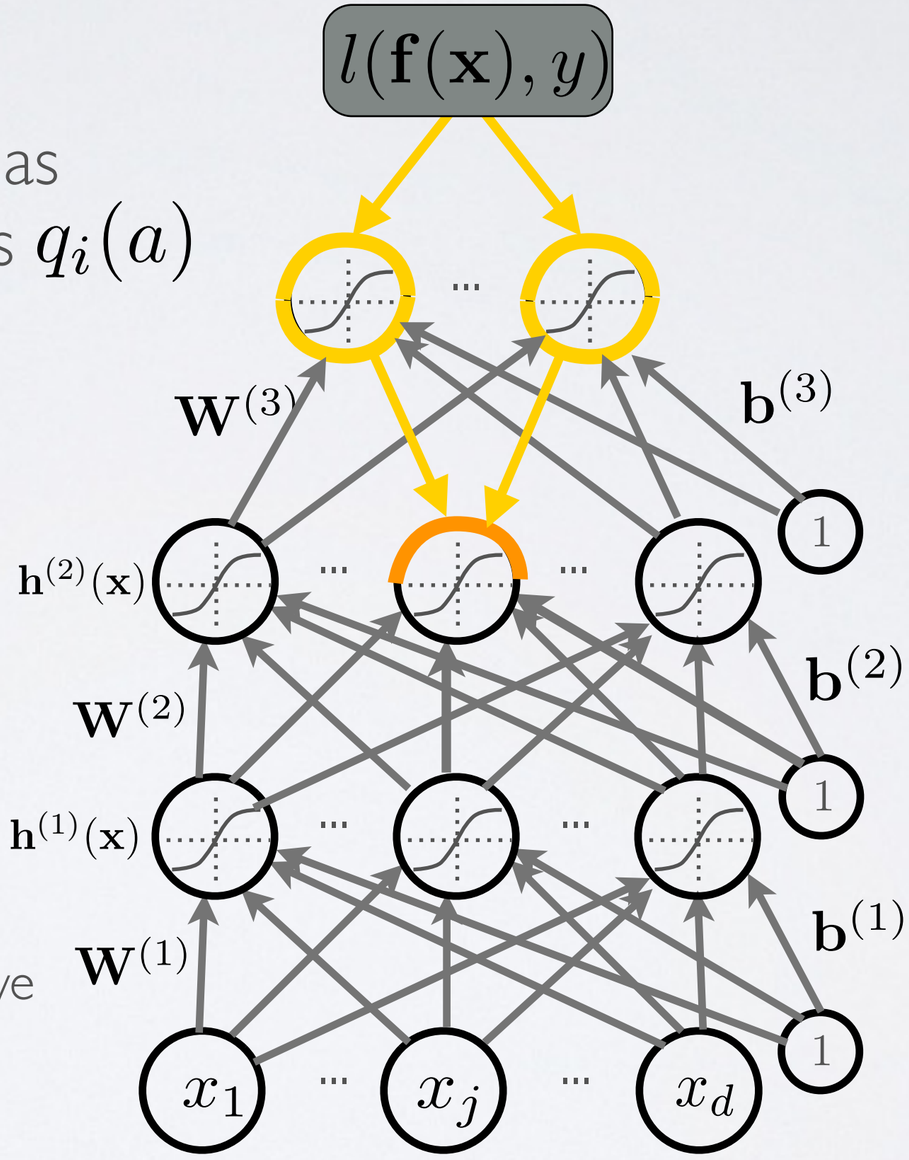
GRADIENT COMPUTATION

Topics: chain rule

- If a function $p(a)$ can be written as a function of intermediate results $q_i(a)$ then we have:

$$\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$$

- We can invoke it by setting
  - a to a unit in layer
  - $q_i(a)$ to a pre-activation in the layer above
  - $p(a)$ is the loss function



GRADIENT COMPUTATION

Topics: loss gradient at hidden layers

- Partial derivative:

$$\frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

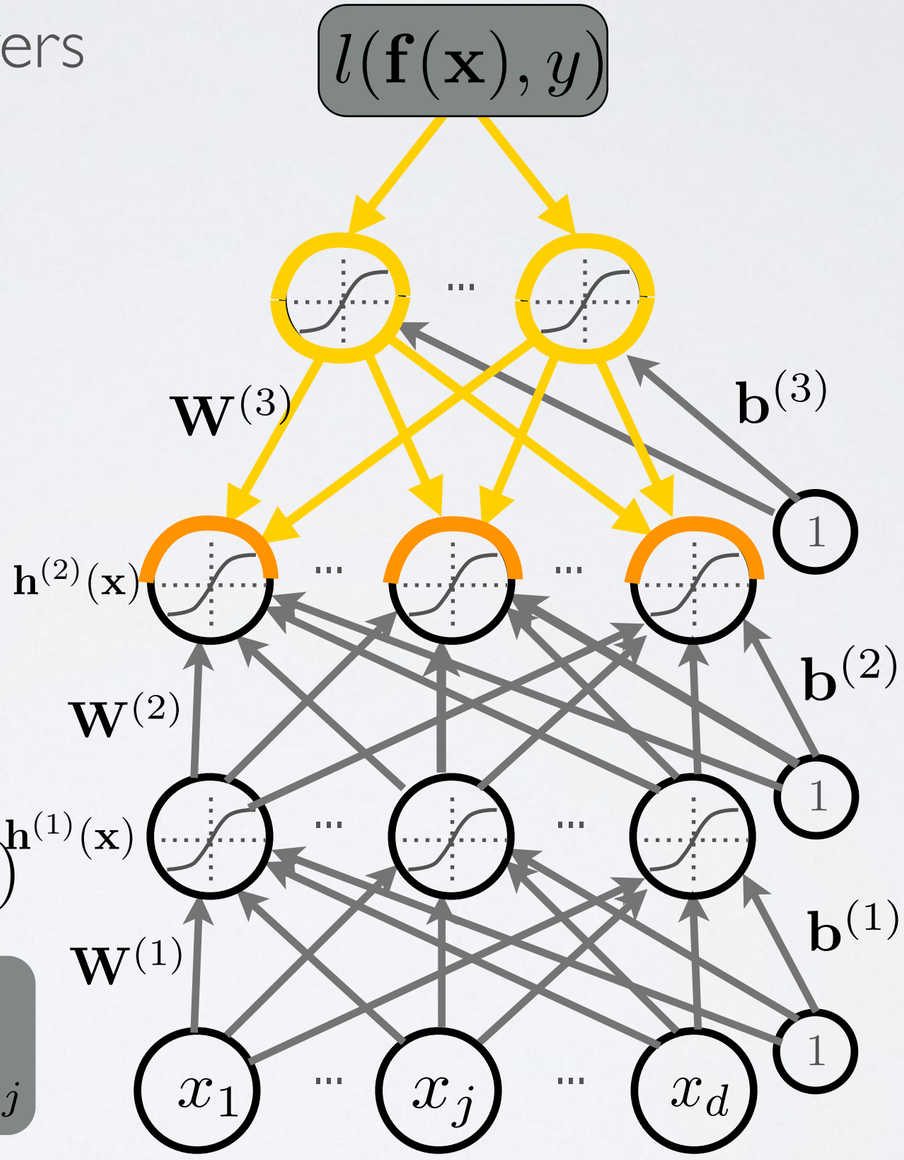
$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j}$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W^{(k+1)}_{i,j}$$

$$= (\mathbf{W}^{k+1}_{\cdot,j})^\top (\nabla_{\mathbf{a}^{k+1}(\mathbf{x})} - \log f(\mathbf{x})_y)$$

REMINDER

$$a^{(k)}(\mathbf{x})_i = b^{(k)}_i + \sum_j W^{(k)}_{i,j} h^{(k-1)}(\mathbf{x})_j$$



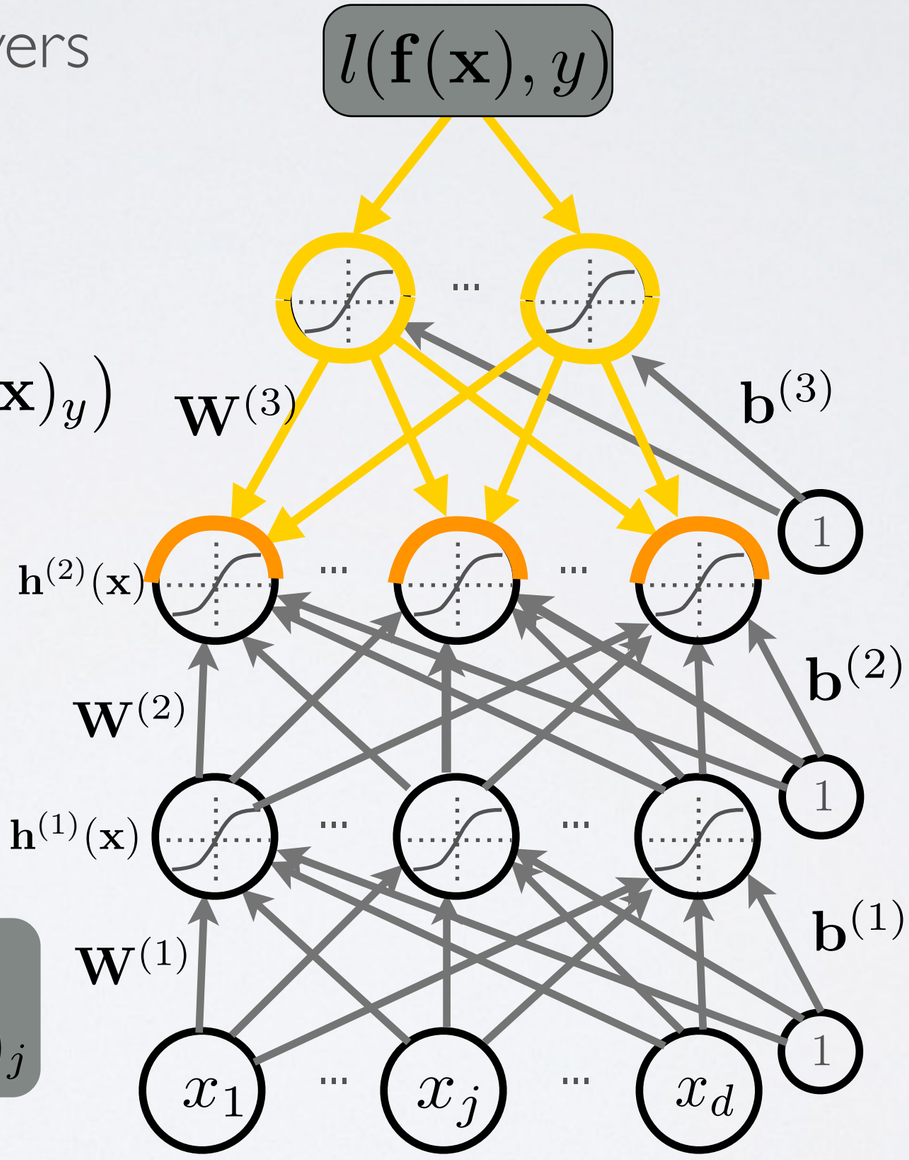
GRADIENT COMPUTATION

Topics: loss gradient at hidden layers

- Gradient:

$$\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \quad \mathbf{W}^{(k+1)\top} \left(\nabla_{\mathbf{a}^{(k+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \right)$$



REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

GRADIENT COMPUTATION

Topics: loss gradient at hidden layers pre-activation

- Partial derivative:

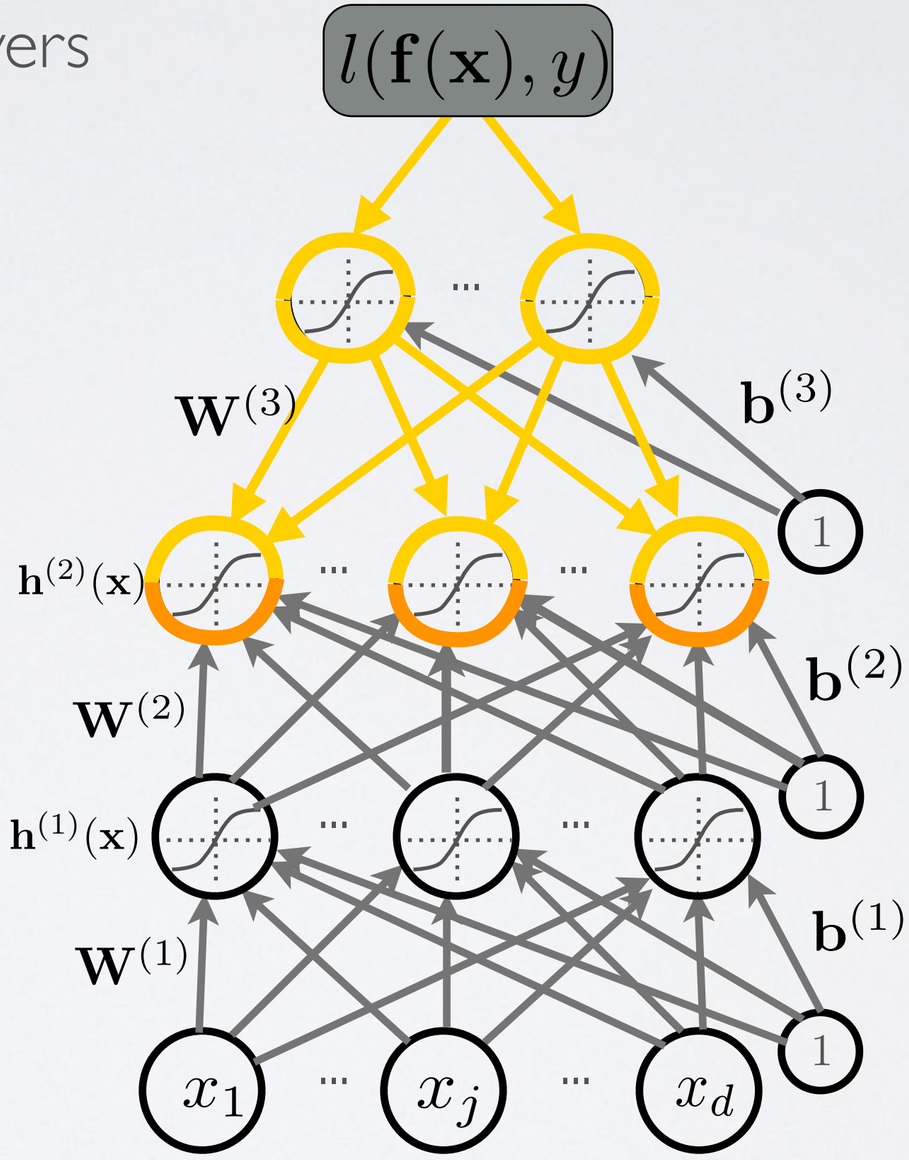
$$\frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j)$$

REMINDER

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$



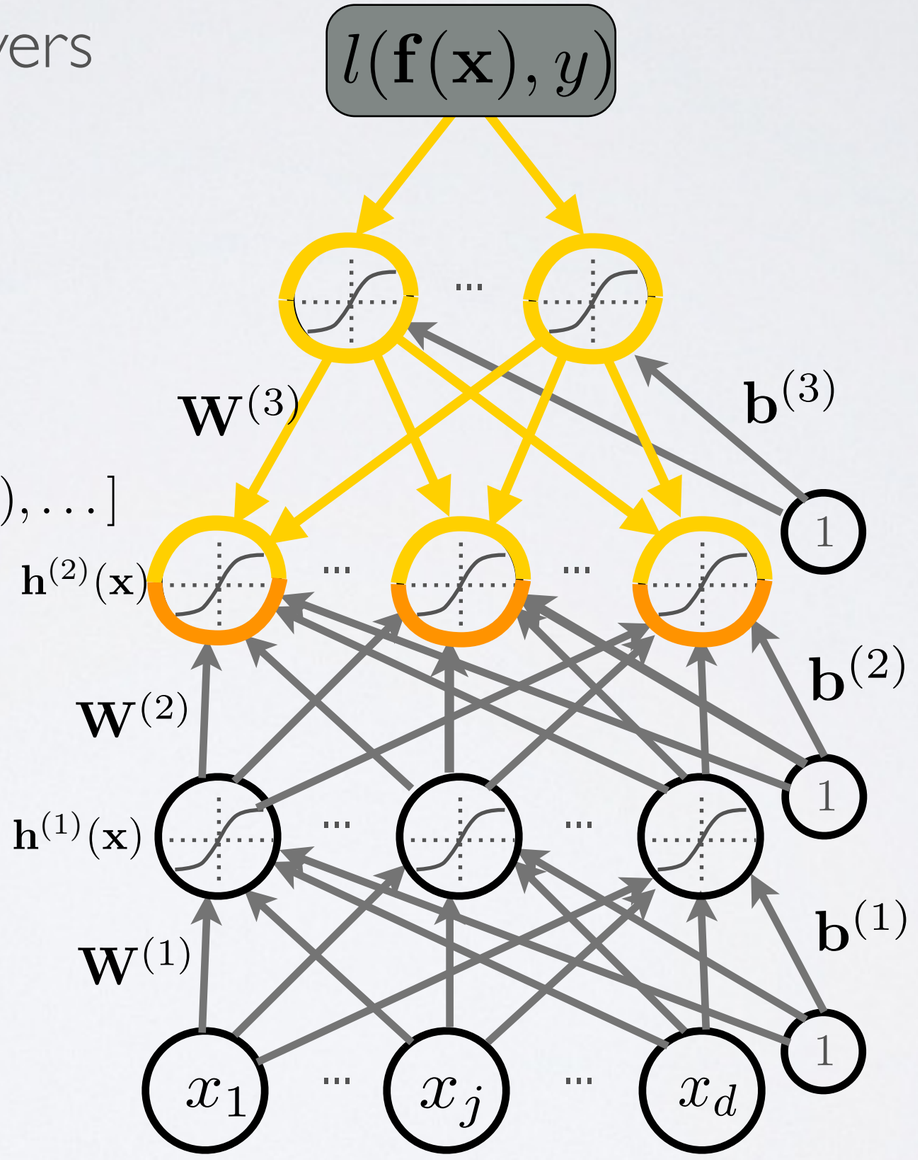
GRADIENT COMPUTATION

Topics: loss gradient at hidden layers pre-activation

- Gradient:

$$\begin{aligned}
& \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\
= \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\
= \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots]
\end{aligned}$$

element-wise product



REMINDER

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$

Neural networks

Training neural networks - activation function derivative

GRADIENT COMPUTATION

Topics: loss gradient at hidden layers pre-activation

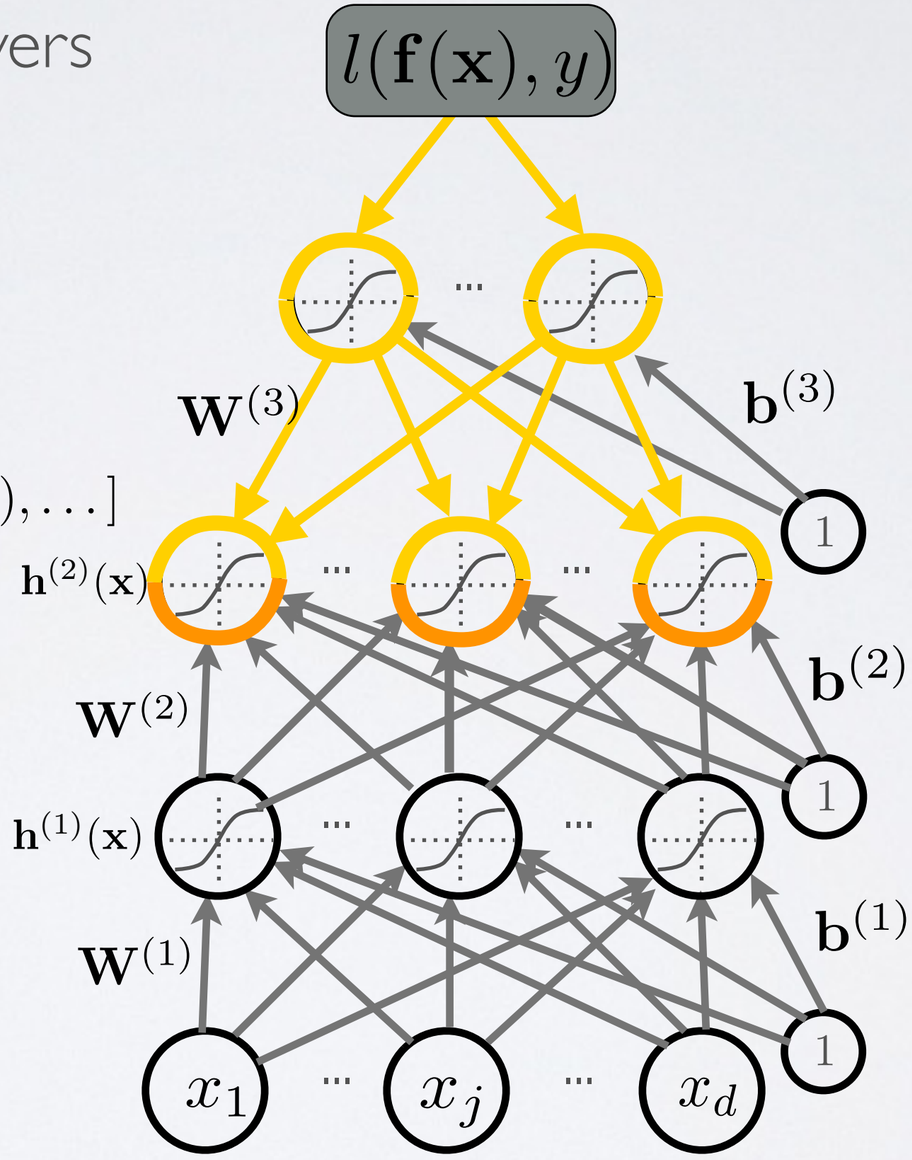
- Gradient:

$$\begin{aligned} & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\ = \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\ = \quad & \left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots] \end{aligned}$$

element-wise product

REMINDER

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$

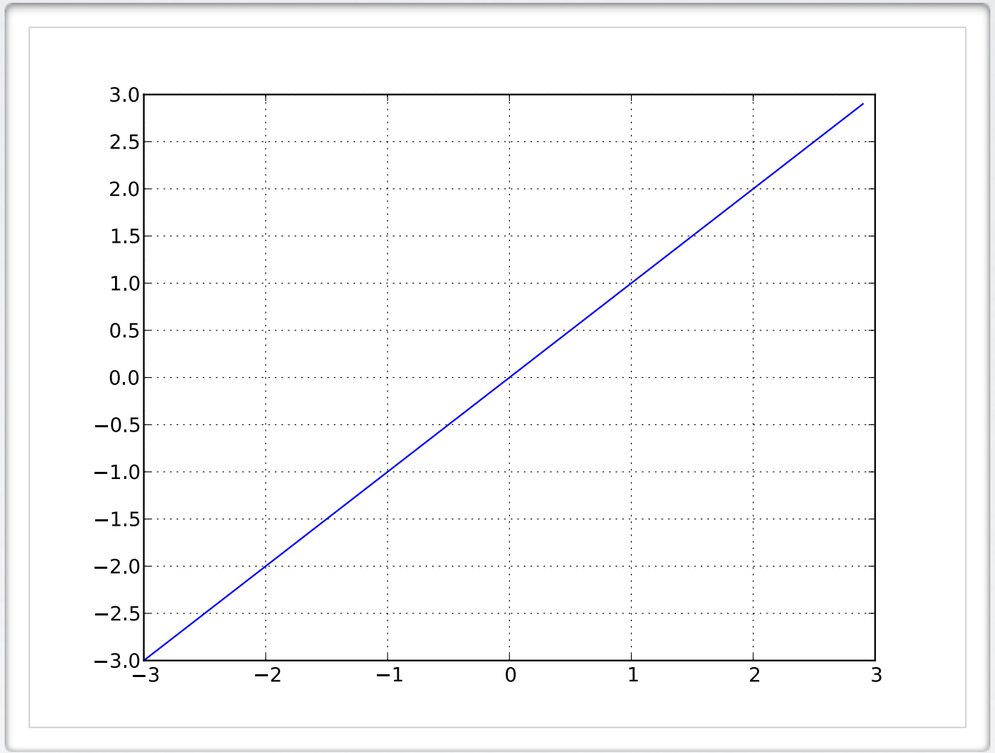


ACTIVATION FUNCTION

Topics: linear activation function gradient

- Partial derivative:

$$g'(a) = 1$$



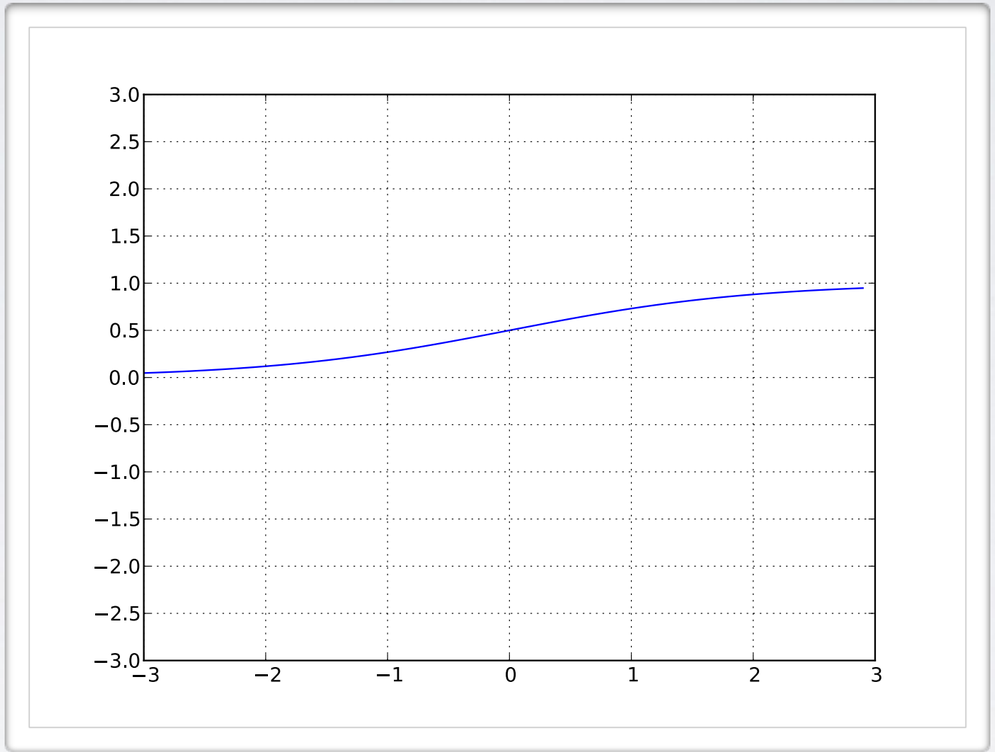
$$g(a) = a$$

ACTIVATION FUNCTION

Topics: sigmoid activation function gradient

- Partial derivative:

$$g'(a) = g(a)(1 - g(a))$$



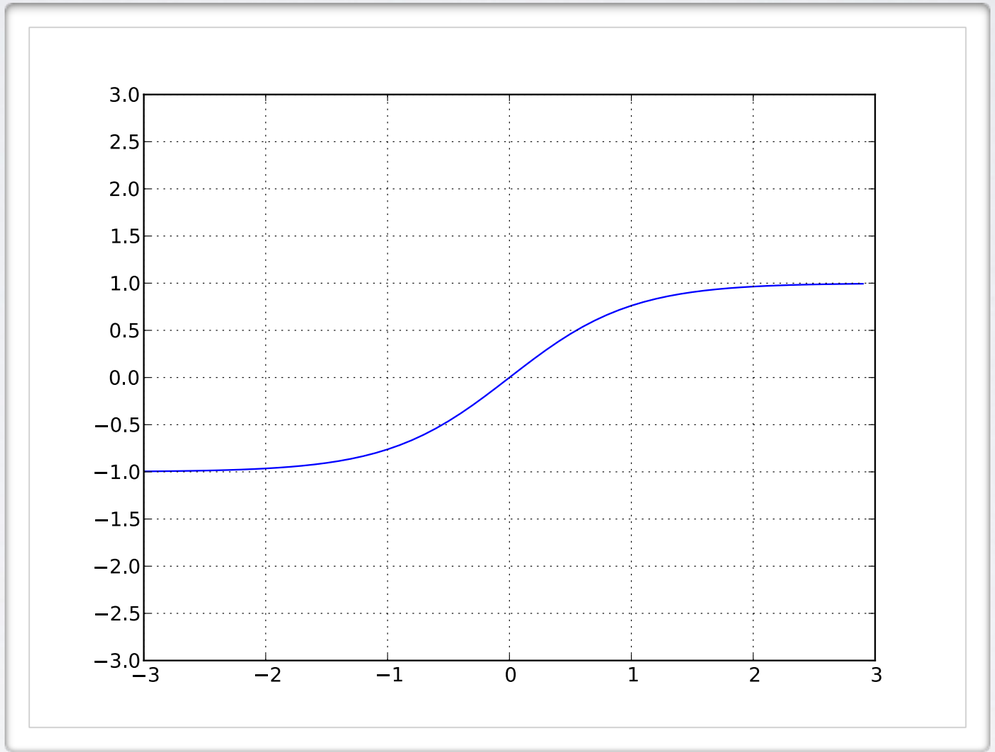
$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

ACTIVATION FUNCTION

Topics: tanh activation function gradient

- Partial derivative:

$$g'(a) = 1 - g(a)^2$$



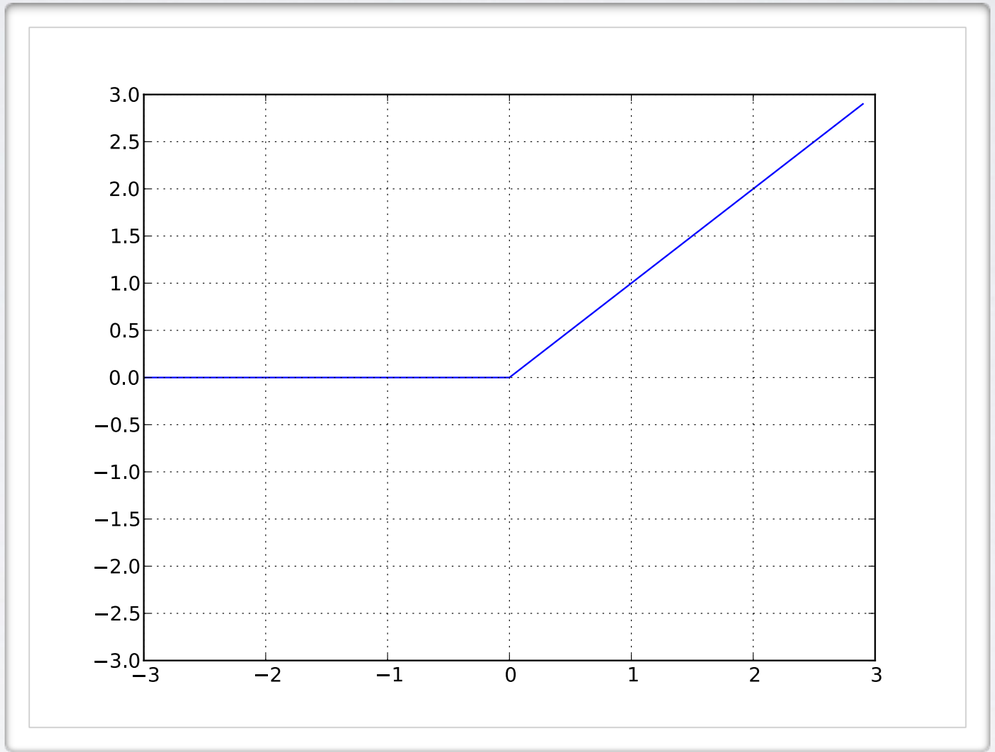
$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

ACTIVATION FUNCTION

Topics: rectified linear activation function gradient

- Partial derivative:

$$g'(a) = 1_{a>0}$$



$$g(a) = \text{reclin}(a) = \max(0, a)$$

Neural networks

Training neural networks - parameter gradient

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

GRADIENT COMPUTATION

Topics: loss gradient of parameters

- Partial derivative (weights):

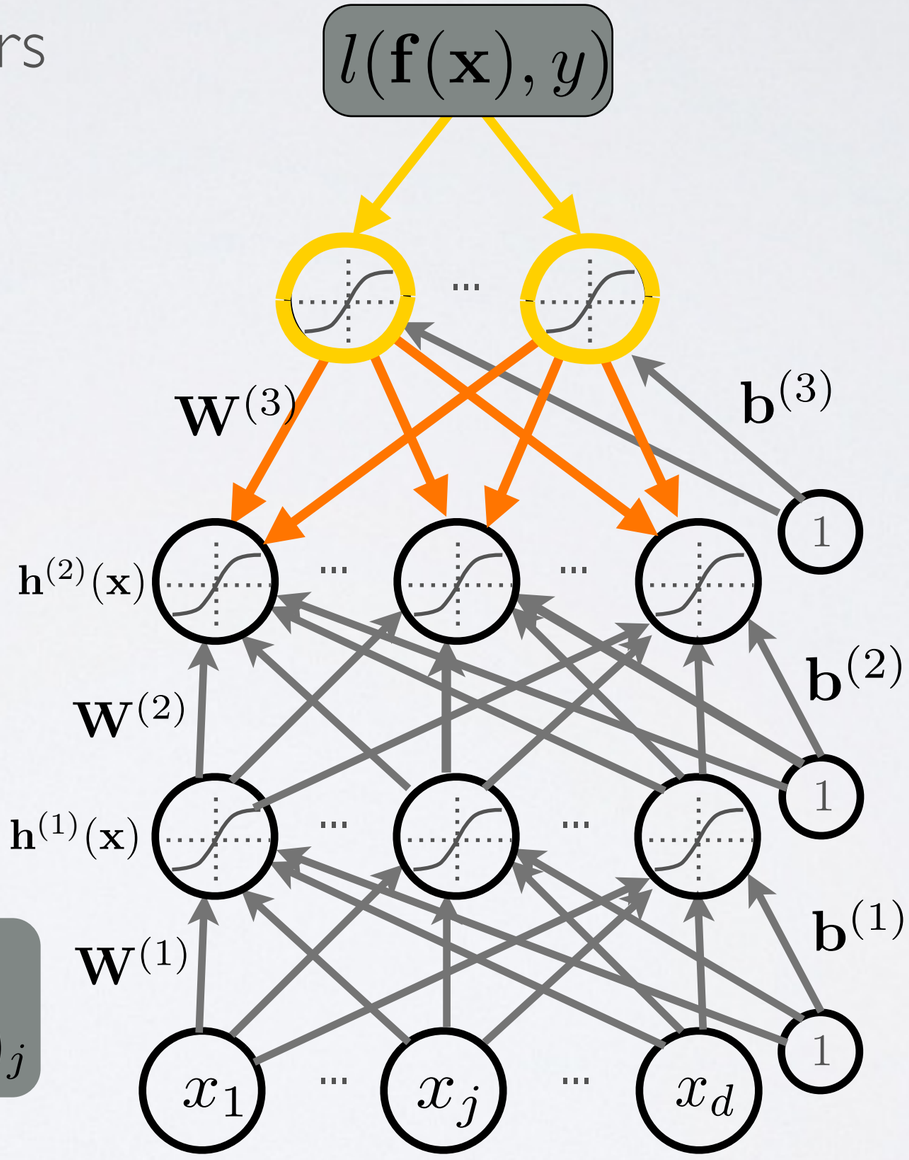
$$\frac{\partial}{\partial W_{i,j}^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial W_{i,j}^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} h_j^{(k-1)}(\mathbf{x})$$

REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

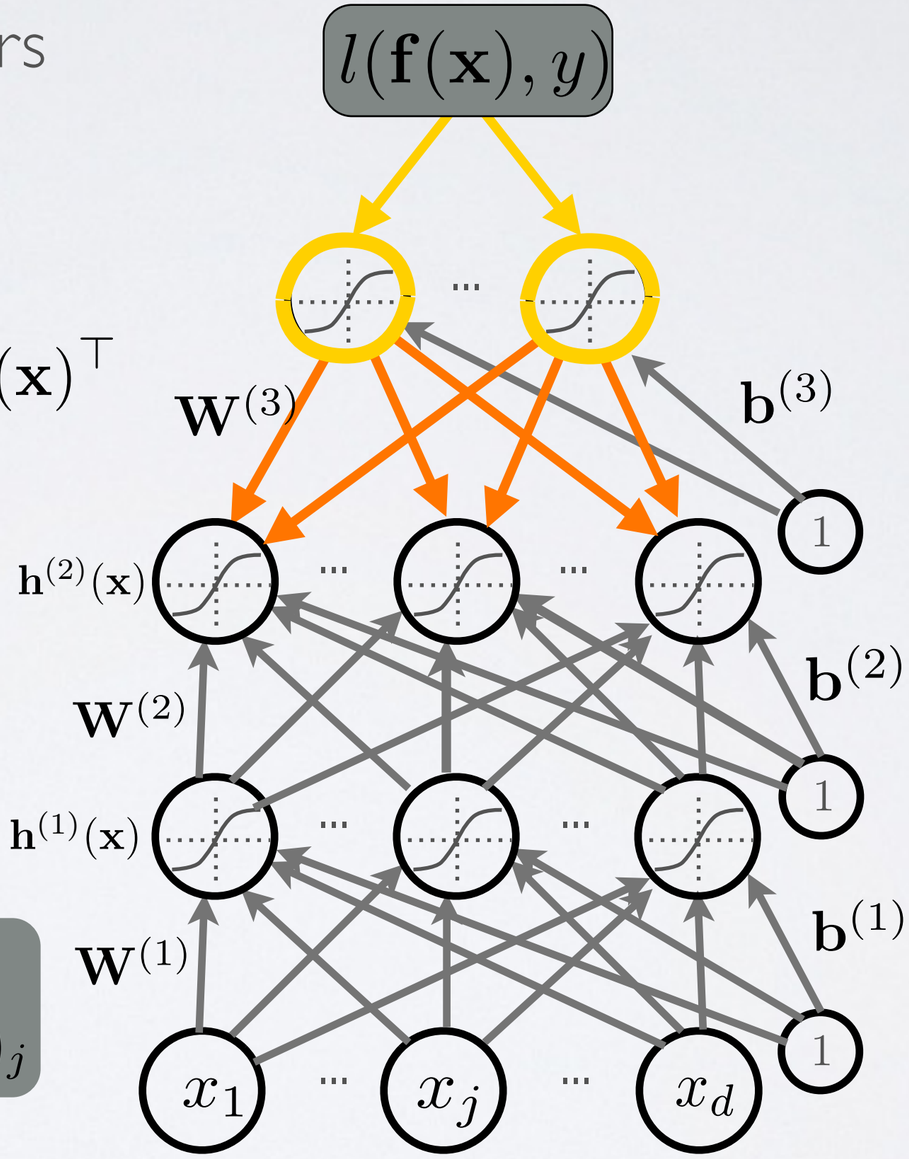


GRADIENT COMPUTATION

Topics: loss gradient of parameters

- Gradient (weights):

$$\begin{aligned} & \nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \\ = \quad & \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \quad \mathbf{h}^{(k-1)}(\mathbf{x})^\top \end{aligned}$$



REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

GRADIENT COMPUTATION

Topics: loss gradient of parameters

- Partial derivative (biases):

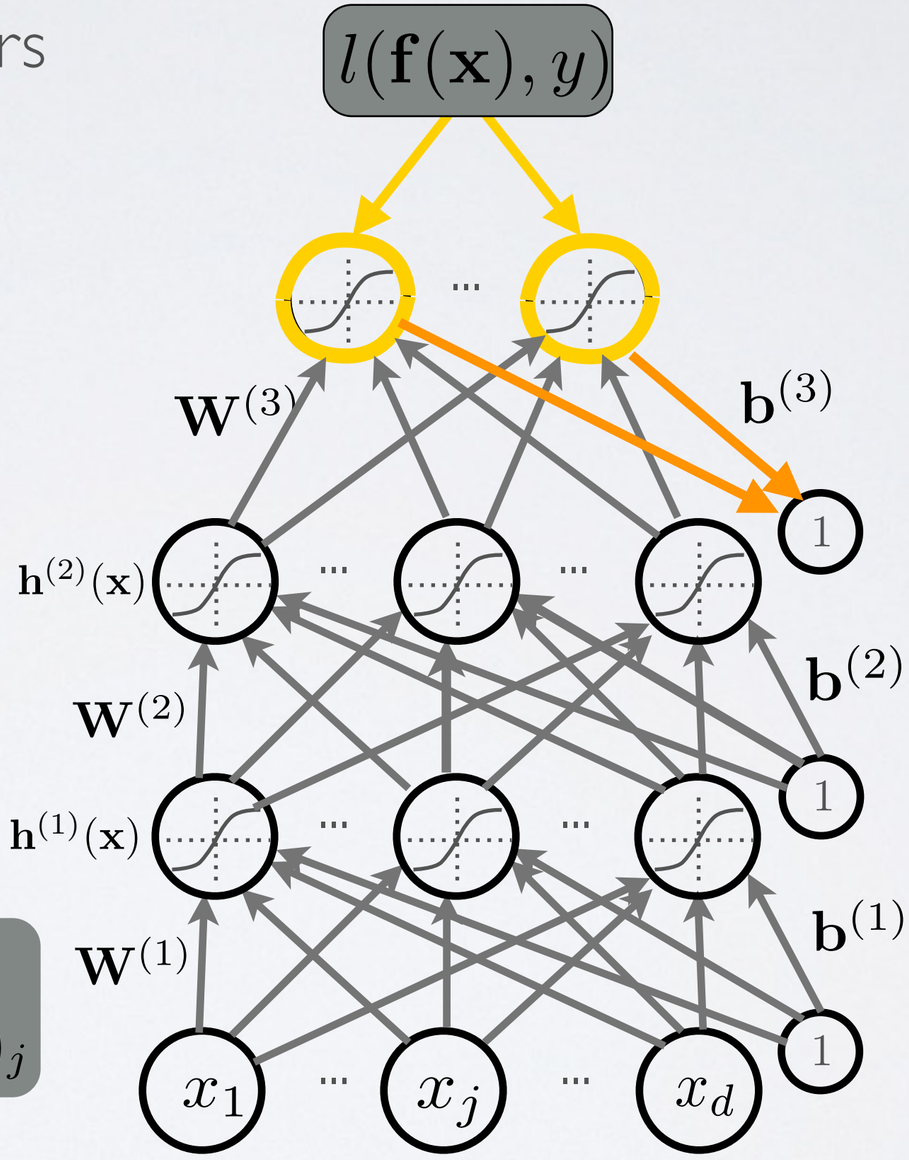
$$\frac{\partial}{\partial b_i^{(k)}} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i} \frac{\partial a^{(k)}(\mathbf{x})_i}{\partial b_i^{(k)}}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k)}(\mathbf{x})_i}$$

REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

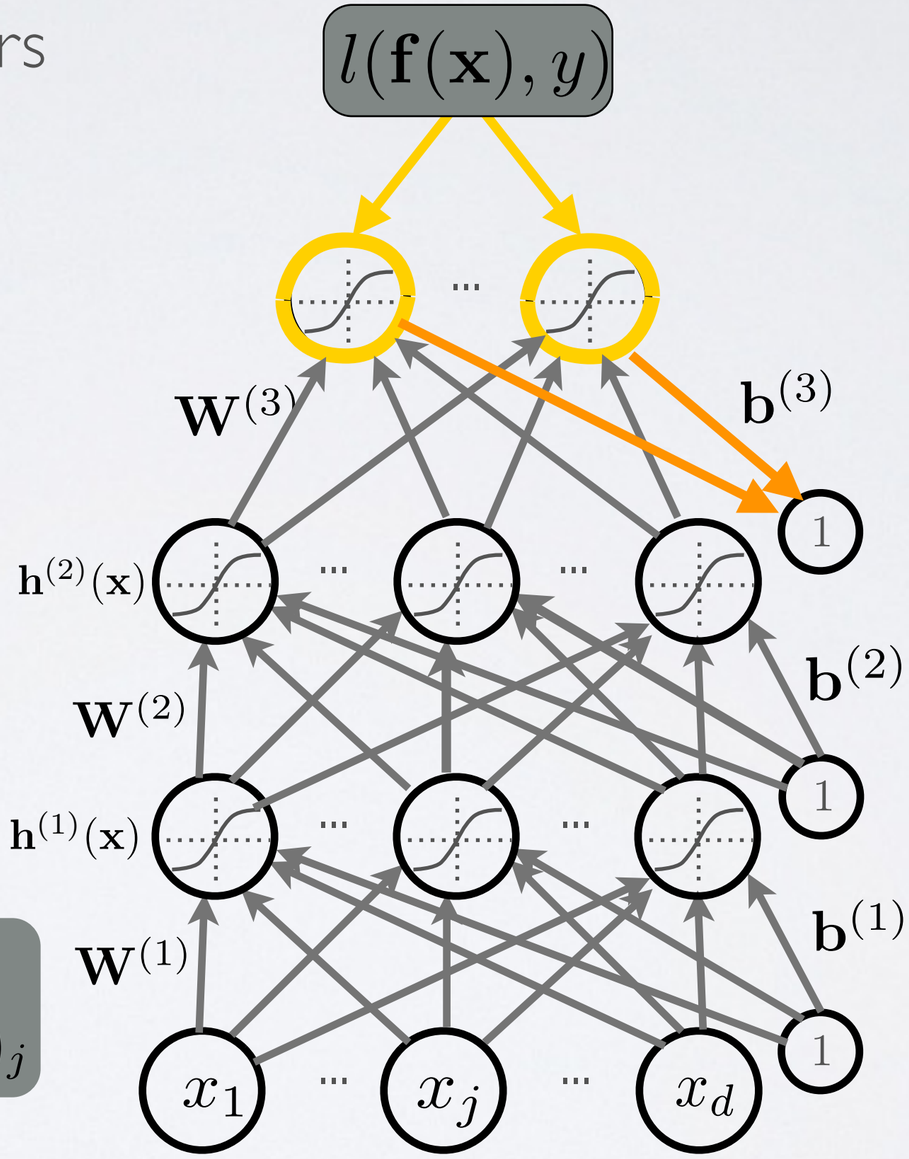


GRADIENT COMPUTATION

Topics: loss gradient of parameters

- Gradient (biases):

$$\begin{aligned} & \nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \\ = \quad & \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \end{aligned}$$



REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$

Neural networks

Training neural networks - backpropagation algorithm

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

BACKPROPAGATION

Topics: backpropagation algorithm

- This assumes a forward propagation has been made before
  - compute output gradient (before activation)

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

  - for k from $L+1$ to 1
    - compute gradients of hidden layer parameter

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \quad \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

    - compute gradient of hidden layer below

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad {\mathbf{W}^{(k)}}^\top \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)$$

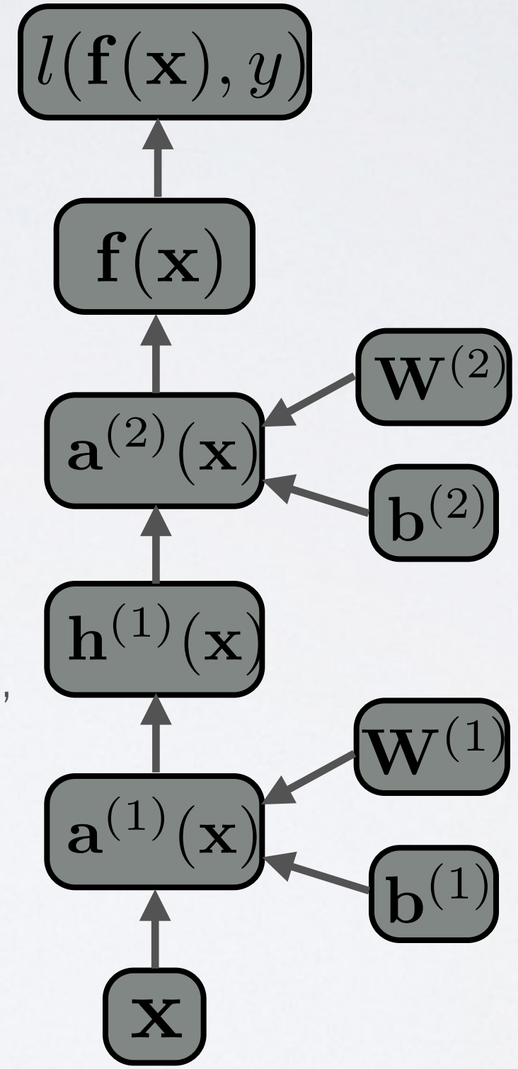
    - compute gradient of hidden layer below (before activation)

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \quad \Longleftarrow \quad \left(\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k-1)}(\mathbf{x})_j), \ldots]$$

FLOW GRAPH

Topics: flow graph

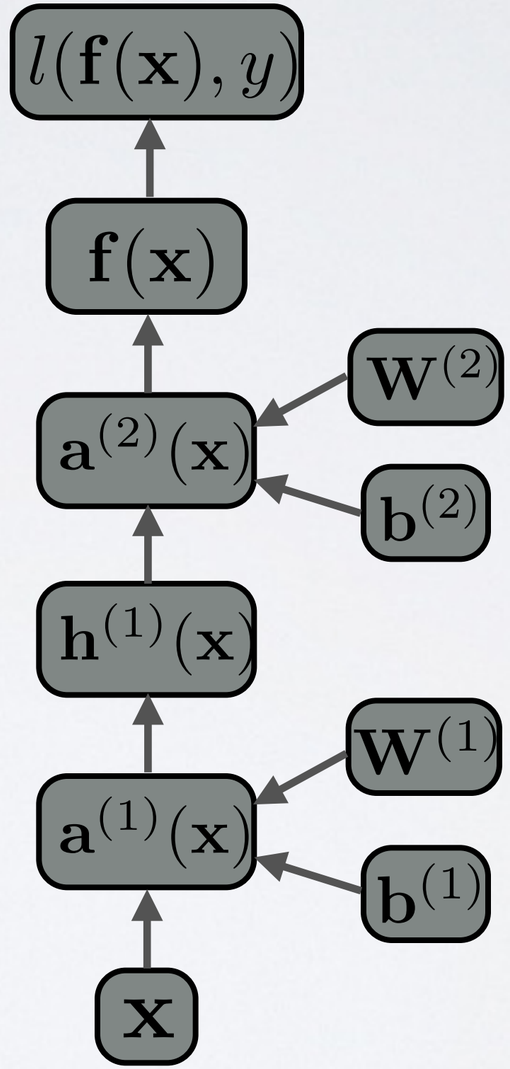
- Forward propagation can be represented as an acyclic flow graph
- It's a nice way of implementing forward propagation in a modular way
  - each box could be an object with an fprop method, that computes the value of the box given its children
  - calling the fprop method of each box in the right order yield forward propagation



FLOW GRAPH

Topics: automatic differentiation

- Each object also has a bprop method
  - it computes the gradient of the loss with respect to each children
  - fprop depends on the fprop of a box's children, while bprop depends the bprop of a box's parents
- By calling bprop in the reverse order, we get backpropagation
  - only need to reach the parameters



Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

Plan

- forward propagation (compute output)
- backpropagation (compute gradients)

lunch

- **complete training algorithm**
- deep learning

Neural networks

Training neural networks - regularization

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

REGULARIZATION

Topics: L2 regularization

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

- Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = 2\mathbf{W}^{(k)}$

- Only applied on weights, not on biases (weight decay)
- Can be interpreted as having a Gaussian prior over the weights

REGULARIZATION

Topics: L1 regularization

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$
  - where $\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$
- Also only applied on weights
- Unlike L2, L1 will push certain weights to be exactly 0
- Can be interpreted as having a Laplacian prior over the weights

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

MACHINE LEARNING

REMINDER

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)

INITIALIZATION

Topics: initialization

- For biases
  - initialize all to 0
- For weights
  - Can't initialize weights to 0 with tanh activation
    - we can show that all gradients would then be 0 (saddle point)
  - Can't initialize all weights to the same value
    - we can show that all hidden units in a layer will always behave the same
    - need to break symmetry
  - Recipe: sample $\mathbf{W}_{i,j}^{(k)}$ from $U\left[-b, b\right]$ where $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$

size of $\mathbf{h}^{(k)}(\mathbf{x})$

    - the idea is to sample around 0 but break symmetry
    - other values of b could work well (not an exact science) (see Glorot & Bengio, 2010)

Neural networks

Training neural networks - model selection

MACHINE LEARNING

Topics: training, validation and test sets, generalization

- Training set $\mathcal{D}^{\text{train}}$ serves to train a model
- Validation set $\mathcal{D}^{\text{valid}}$ serves to select hyper-parameters
  - hidden layer size(s), learning rate, number of iterations/epochs, etc.
- Test set $\mathcal{D}^{\text{test}}$ serves to estimate the generalization performance (error)

- Generalization is the behavior of the model on **unseen examples**
  - this is what we care about in machine learning!

MODEL SELECTION

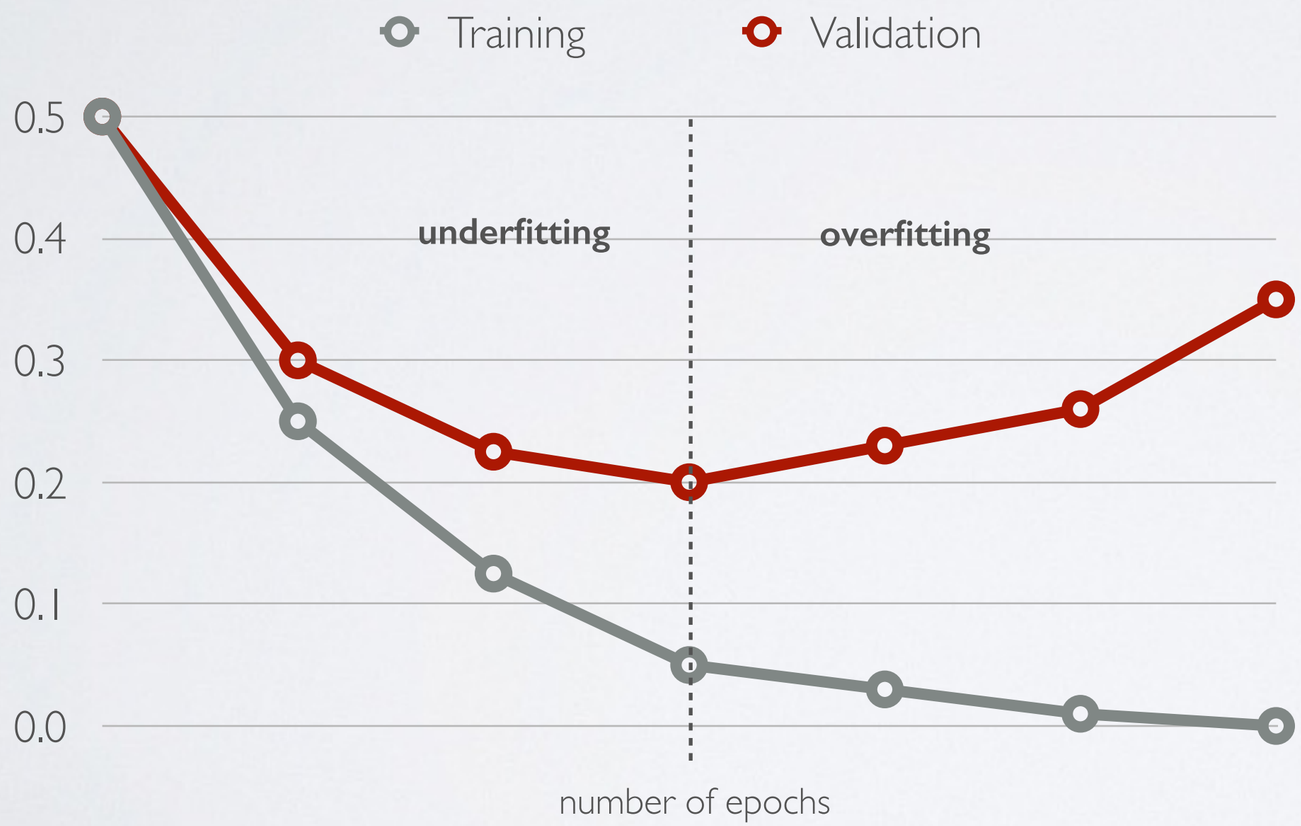
Topics: grid search, random search

- To search for the best configuration of the hyper-parameters:
  - you can perform a grid search
    - specify a set of values you want to test for each hyper-parameter
    - try all possible configurations of these values
  - you can perform a random search (Bergstra and Bengio, 2012)
    - specify a distribution over the values of each hyper-parameters (e.g. uniform in some range)
    - sample independently each hyper-parameter to get configurations
  - bayesian optimization or sequential model-based optimization …
- Use a validation set performance to select the best configuration
- You can go back and refine the grid/distributions if needed

KNOWING WHEN TO STOP

Topics: early stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead)



Neural networks

Training neural networks - other tricks of the trade

OTHER TRICKS OF THE TRADE

Topics: normalization of data, decaying learning rate

- Normalizing your (real-valued) data
  - for dimension x_i subtract its training set mean
  - divide by dimension x_i by its training set standard deviation
  - this can speed up training (in number of epochs)

- Decaying the learning rate
  - as we get closer to the optimum, makes sense to take smaller update steps
    (i) start with large learning rate (e.g. 0.1)
    (ii) maintain until validation error stops improving
    (iii) divide learning rate by 2 and go back to (ii)

OTHER TRICKS OF THE TRADE

Topics: mini-batch, momentum

- Can update based on a mini-batch of example (instead of 1 example):
  - the gradient is the average regularized loss for that mini-batch
  - can give a more accurate estimate of the risk gradient
  - can leverage matrix/matrix operations, which are more efficient

- Can use an exponential average of previous gradients:

$$\overline{\nabla}_{\boldsymbol{\theta}}^{(t)} = \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\boldsymbol{\theta}}^{(t-1)}$$

  - can get through plateaus more quickly, by "gaining momentum"

OTHER TRICKS OF THE TRADE

Topics: Adagrad, RMSProp, Adam

- Updates with adaptive learning rates ("one learning rate per parameter")
  - **Adagrad**: learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\gamma^{(t)} = \gamma^{(t-1)} + \left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2 \qquad \overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

  - **RMSProp**: instead of cumulative sum, use exponential moving average

$$\gamma^{(t)} = \beta\gamma^{(t-1)} + (1-\beta)\left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2 \qquad \overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

  - **Adam**: essentially combines RMSProp with momentum

GRADIENT CHECKING

Topics: finite difference approximation

- To debug your implementation of fprop/bprop, you can compare with a finite-difference approximation of the gradient

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

  - $f(x)$ would be the loss
  - x would be a parameter
  - $f(x+\epsilon)$ would be the loss if you add ϵ to the parameter
  - $f(x-\epsilon)$ would be the loss if you subtract ϵ to the parameter

DEBUGGING ON SMALL DATASET

Topics: debugging on small dataset

- Next, make sure your model is able to (over)fit on a very small dataset (~50 examples)

- If not, investigate the following situations:
  - Are some of the units saturated, even before the first update?
    - scale down the initialization of your parameters for these units
    - properly normalize the inputs
  - Is the training error bouncing up and down?
    - decrease the learning rate
- Note that this isn't a replacement for gradient checking
  - could still overfit with some of the gradients being wrong

Neural networks

Hugo Larochelle (@hugo_larochelle)
Twitter / Université de Sherbrooke

Plan

- forward propagation (compute output)
- backpropagation (compute gradients)

lunch

- complete training algorithm
- **deep learning**

Neural networks

Deep learning - motivation

NEURAL NETWORK

REMINDER

Topics: multilayer neural network

- Could have L hidden layers:
  - layer input activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

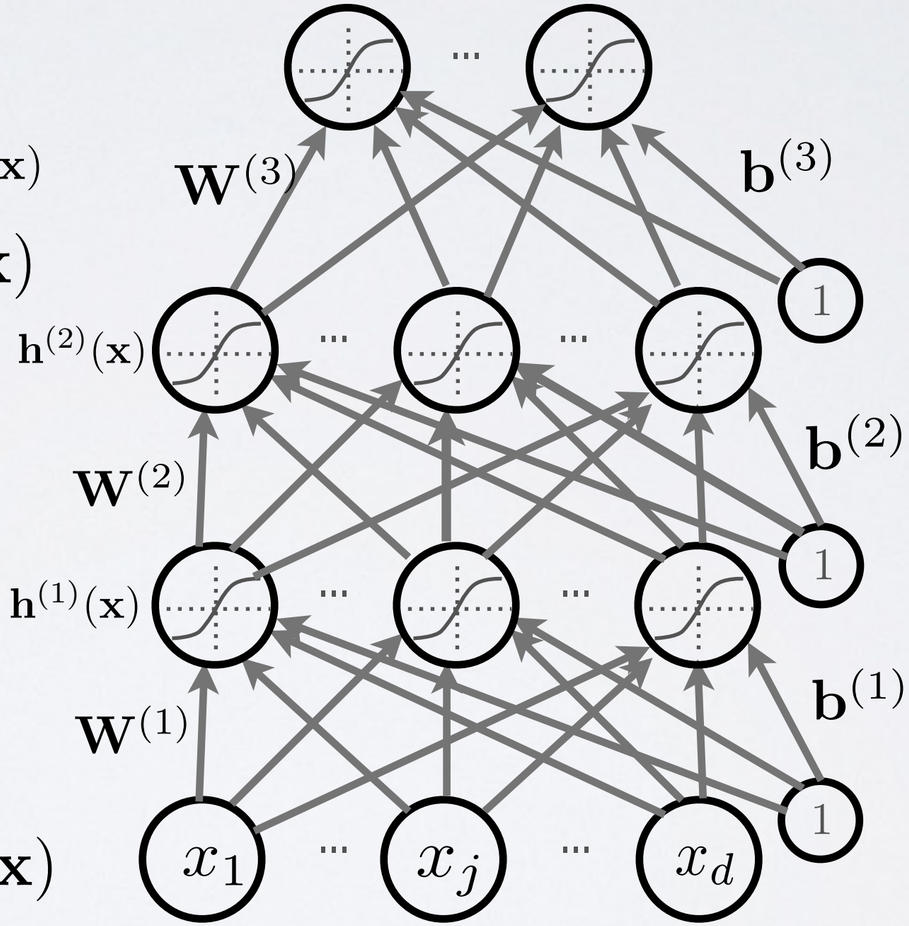
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

  - output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



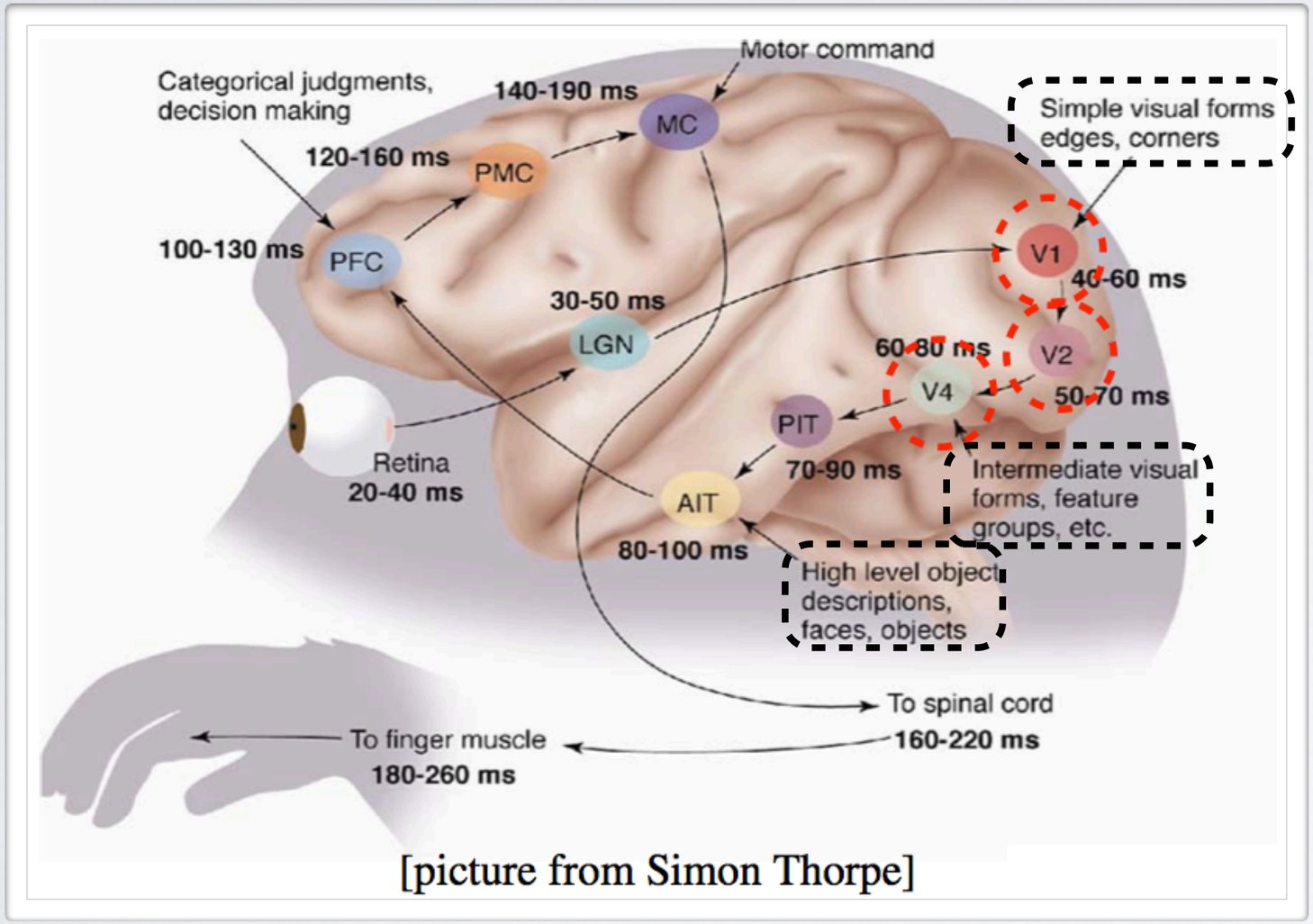
DEEP LEARNING

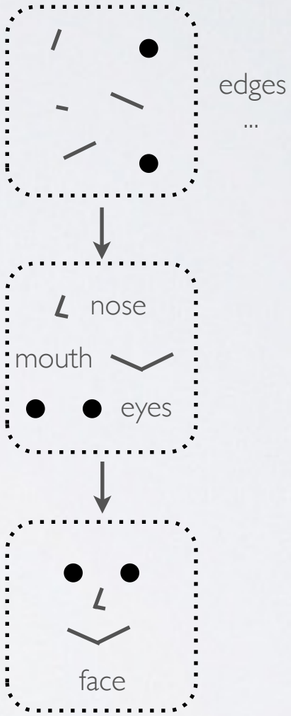
Topics: deep learning, distributed representation

- Deep learning is research on learning models with multilayer representations
  - multilayer (feed-forward) neural network
  - multilayer graphical model (deep belief network, deep Boltzmann machine)
- Each layer corresponds to a "distributed representation"
  - units in layer are not mutually exclusive
    - each unit is a separate feature of the input
    - two units can be "active" at the same time
  - they do not correspond to a partitioning (clustering) of the inputs
    - in clustering, an input can only belong to a single cluster

DEEP LEARNING

Topics: inspiration from visual cortex





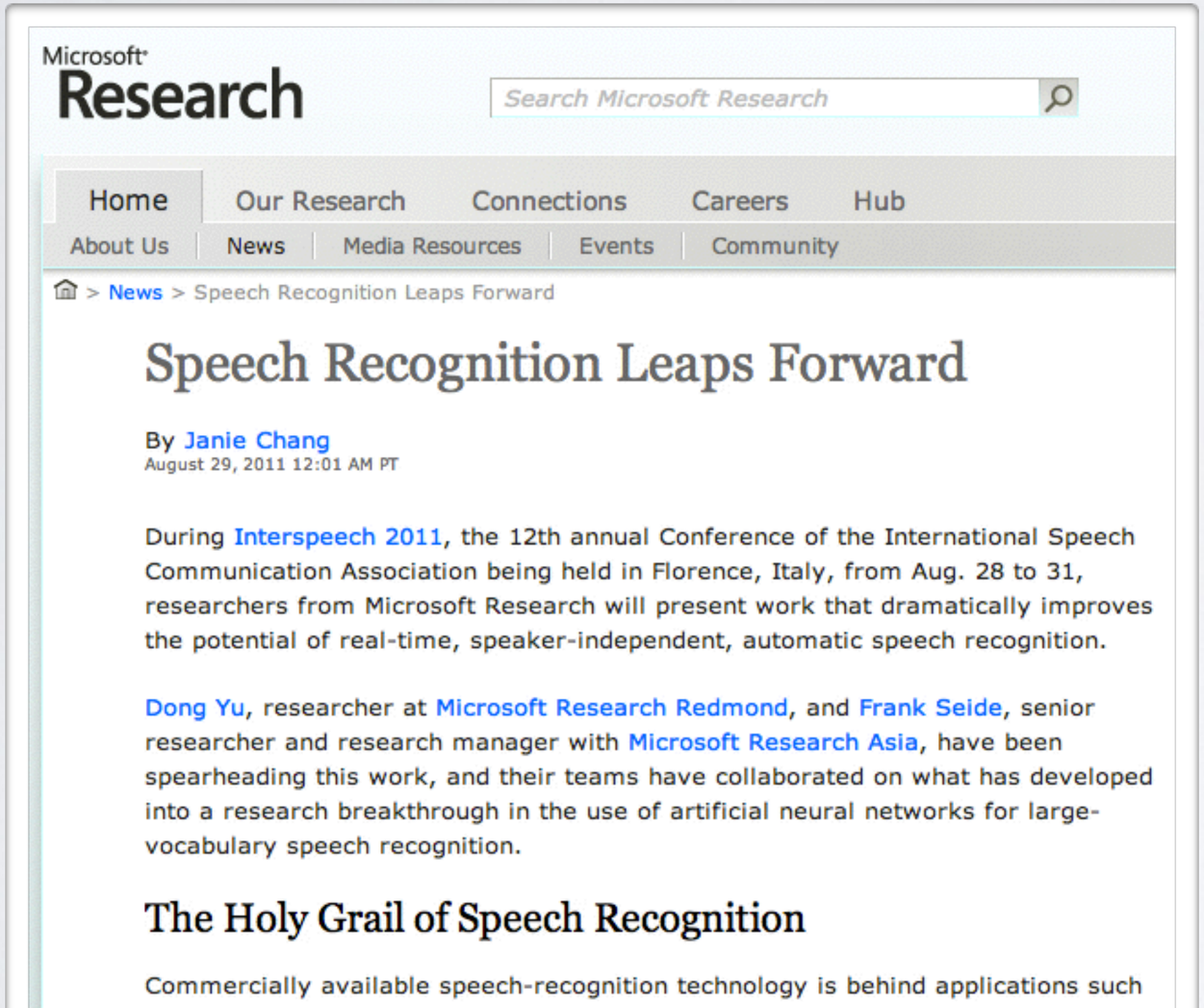
DEEP LEARNING

Topics: theoretical justification

- A deep architecture can represent certain functions (exponentially) more compactly
- Example: Boolean functions
  - a Boolean circuit is a sort of feed-forward network where hidden units are logic gates (i.e. AND, OR or NOT functions of their arguments)
  - any Boolean function can be represented by a "single hidden layer" Boolean circuit
    - however, it might require an exponential number of hidden units
  - it can be shown that there are Boolean functions which
    - require an exponential number of hidden units in the single layer case
    - require a polynomial number of hidden units if we can adapt the number of layers
  - See "Exploring Strategies for Training Deep Neural Networks" for a discussion

DEEP LEARNING

Topics: success story: speech recognition



Microsoft Research

Search Microsoft Research

Home | Our Research | Connections | Careers | Hub

About Us | News | Media Resources | Events | Community

> News > Speech Recognition Leaps Forward

Speech Recognition Leaps Forward

By Janie Chang
August 29, 2011 12:01 AM PT

During Interspeech 2011, the 12th annual Conference of the International Speech Communication Association being held in Florence, Italy, from Aug. 28 to 31, researchers from Microsoft Research will present work that dramatically improves the potential of real-time, speaker-independent, automatic speech recognition.

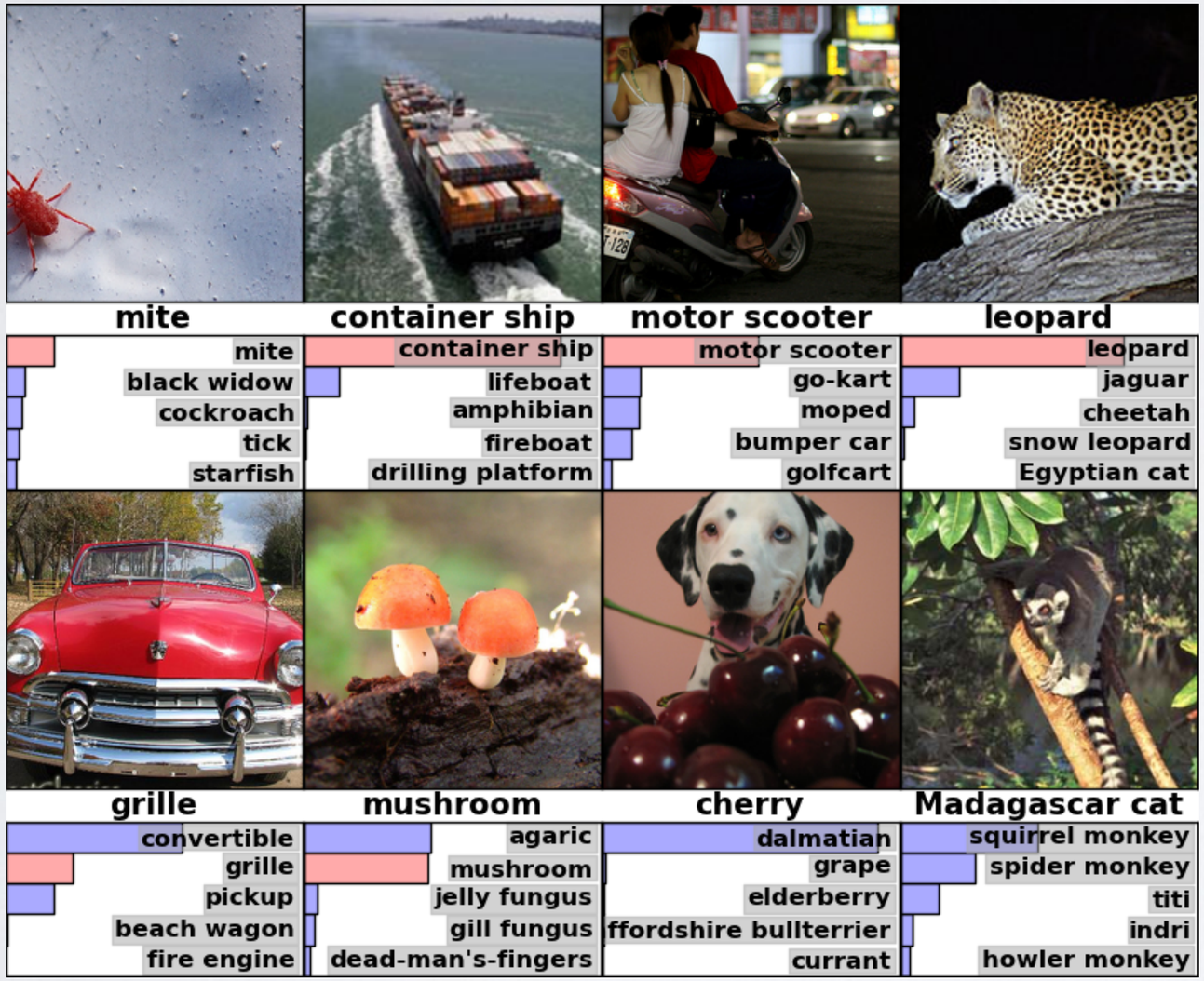
Dong Yu, researcher at Microsoft Research Redmond, and Frank Seide, senior researcher and research manager with Microsoft Research Asia, have been spearheading this work, and their teams have collaborated on what has developed into a research breakthrough in the use of artificial neural networks for large-vocabulary speech recognition.

The Holy Grail of Speech Recognition

Commercially available speech-recognition technology is behind applications such

DEEP LEARNING

Topics: success story: computer vision



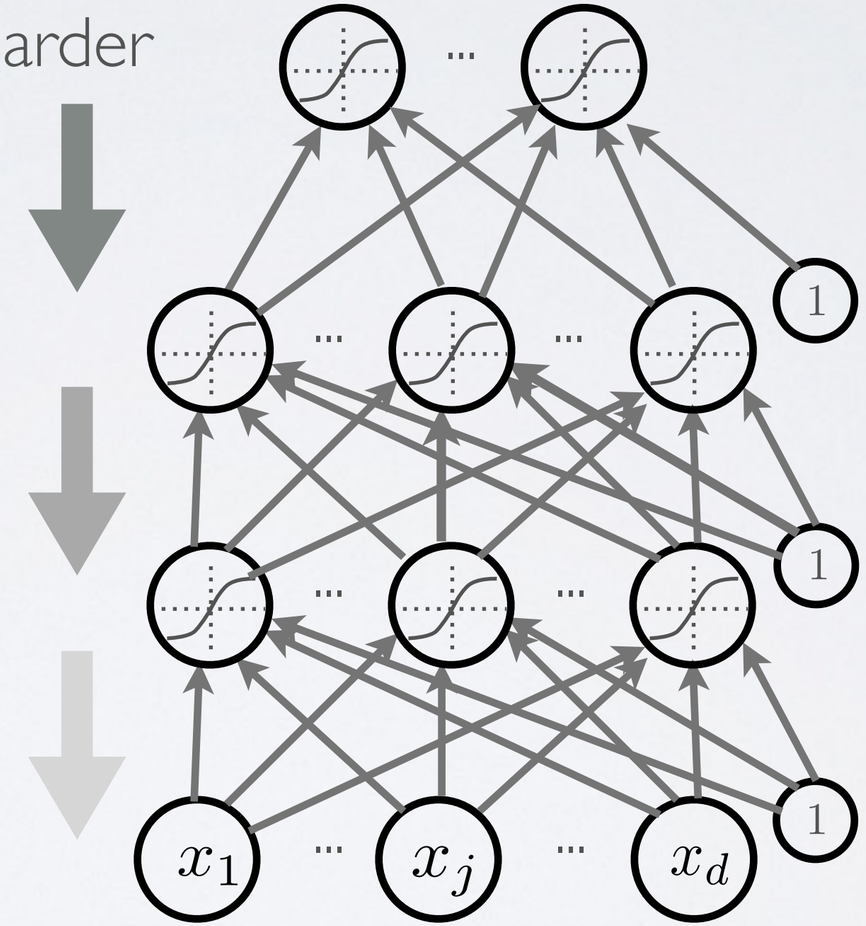
Neural networks

Deep learning - difficulty of training

DEEP LEARNING

Topics: why training is hard

- First hypothesis: optimization is harder (underfitting)
  - vanishing gradient problem
  - saturated units block gradient propagation

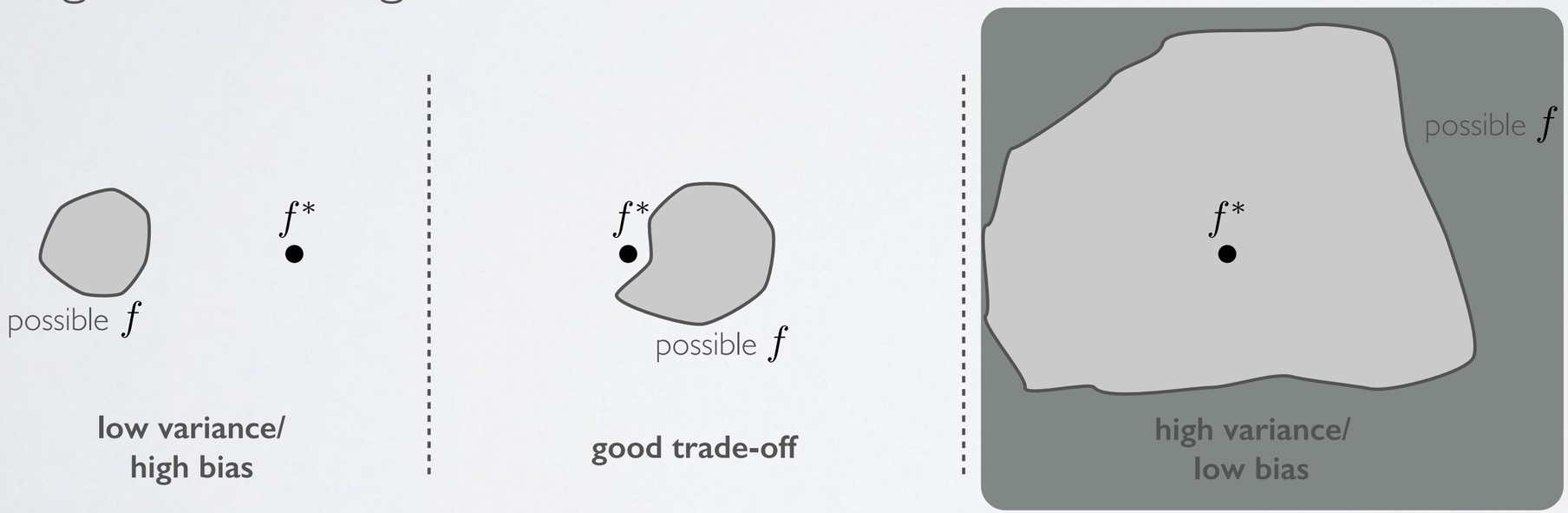


- This is a well known problem in recurrent neural networks

DEEP LEARNING

Topics: why training is hard

- Second hypothesis: overfitting
  - we are exploring a space of complex functions
  - deep nets usually have lots of parameters
- Might be in a high variance / low bias situation



DEEP LEARNING

Topics: why training is hard

- Depending on the problem, one or the other situation will tend to dominate

- If first hypothesis (underfitting): better optimize
  - use better optimization methods
  - use GPUs

- If second hypothesis (overfitting): use better regularization
  - unsupervised learning
  - stochastic «dropout» training

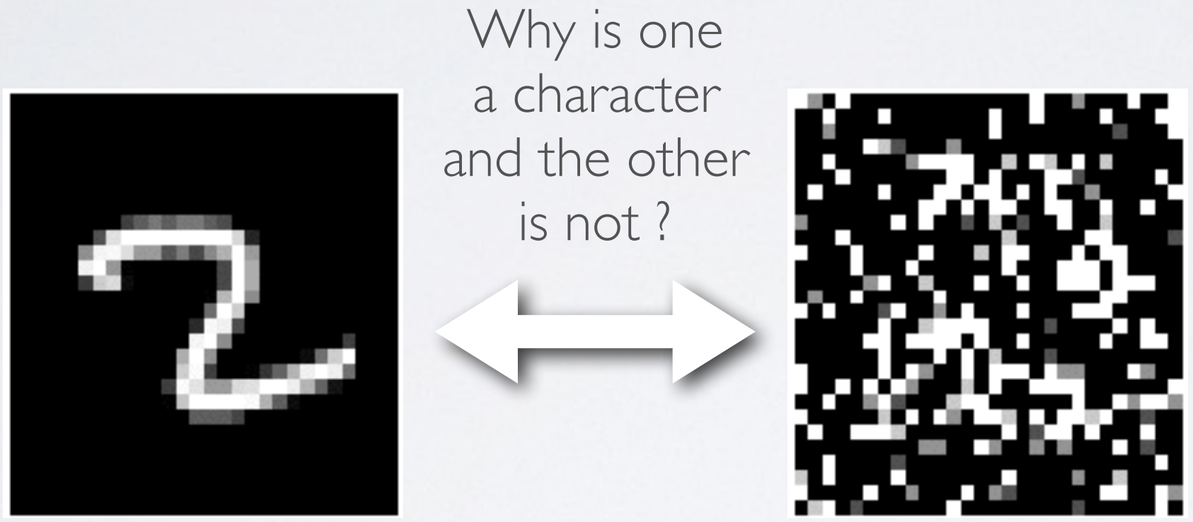
Neural networks

Deep learning - unsupervised pre-training

UNSUPERVISED PRE-TRAINING

Topics: unsupervised pre-training

- Solution: initialize hidden layers using unsupervised learning
  - force network to represent latent structure of input distribution

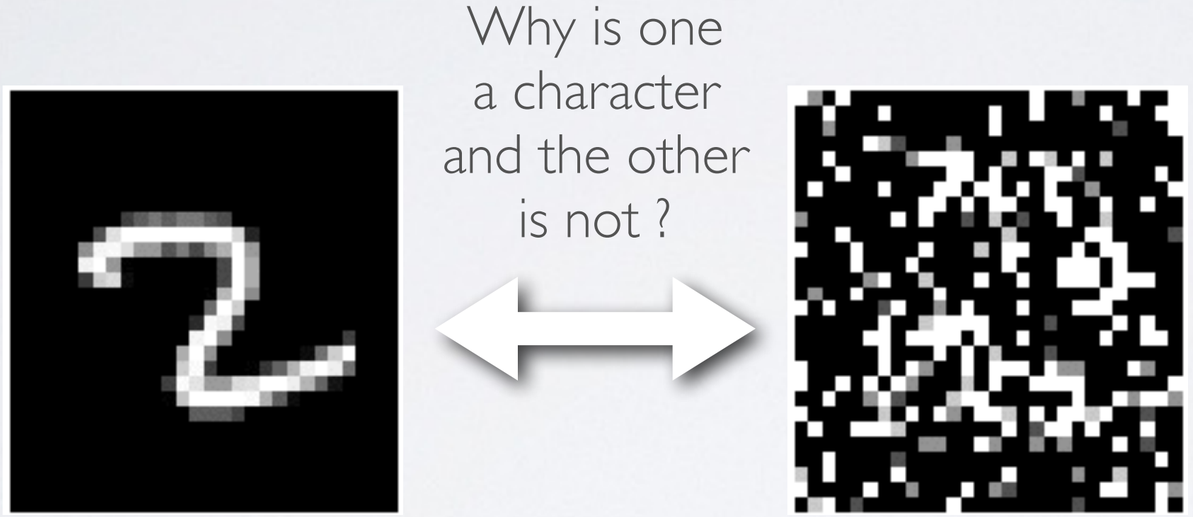


  - encourage hidden layers to encode that structure

UNSUPERVISED PRE-TRAINING

Topics: unsupervised pre-training

- Solution: initialize hidden layers using unsupervised learning
  - this is a harder task than supervised learning (classification)



character image

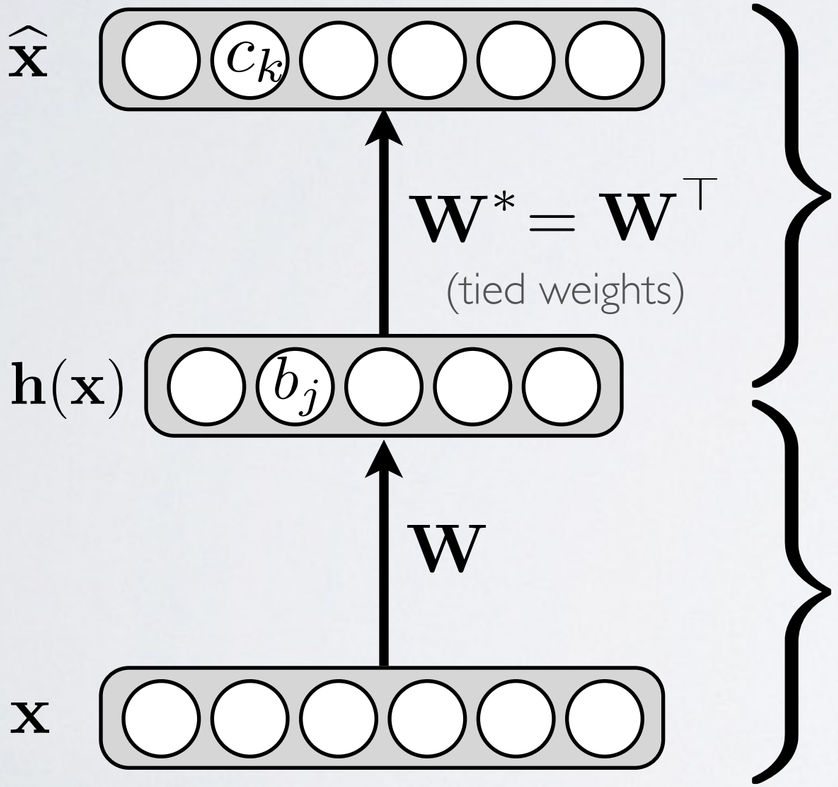
random image

  - hence we expect less overfitting

AUTOENCODER

Topics: autoencoder, encoder, decoder, tied weights

- Feed-forward neural network trained to reproduce its input at the output layer



Decoder

$$\begin{aligned}\hat{\mathbf{x}} &= o(\hat{\mathbf{a}}(\mathbf{x})) \\ &= \underbrace{\mathrm{sigm}}_{\text{for binary inputs}}(\mathbf{c} + \mathbf{W}^*\mathbf{h}(\mathbf{x}))\end{aligned}$$

Encoder

$$\begin{aligned}\mathbf{h}(\mathbf{x}) &= g(\mathbf{a}(\mathbf{x})) \\ &= \mathrm{sigm}(\mathbf{b} + \mathbf{W}\mathbf{x})\end{aligned}$$

AUTOENCODER

Topics: loss function

$$f(\mathbf{x}) \equiv \widehat{\mathbf{x}}$$

- For binary inputs:

$$l(f(\mathbf{x})) = -\sum_k \left(x_k \log(\widehat{x}_k) + (1 - x_k) \log(1 - \widehat{x}_k)\right)$$

  - cross-entropy (more precisely: sum of Bernoulli cross-entropies)

- For real-valued inputs:

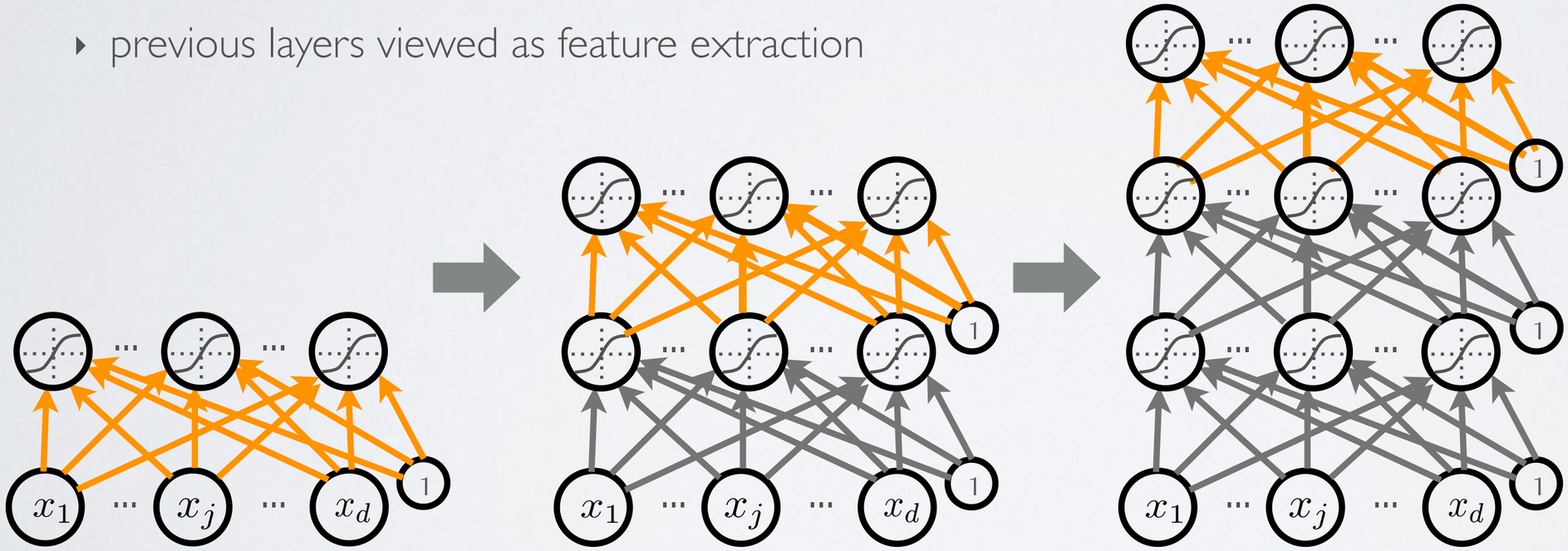
$$l(f(\mathbf{x})) = \frac{1}{2} \sum_k (\widehat{x}_k - x_k)^2$$

  - sum of squared differences (squared euclidean distance)
  - we use a linear activation function at the output

UNSUPERVISED PRE-TRAINING

Topics: unsupervised pre-training

- We will use a greedy, layer-wise procedure
  - train one layer at a time, from first to last, with unsupervised criterion
  - fix the parameters of previous hidden layers
  - previous layers viewed as feature extraction



UNSUPERVISED PRE-TRAINING

Topics: unsupervised pre-training

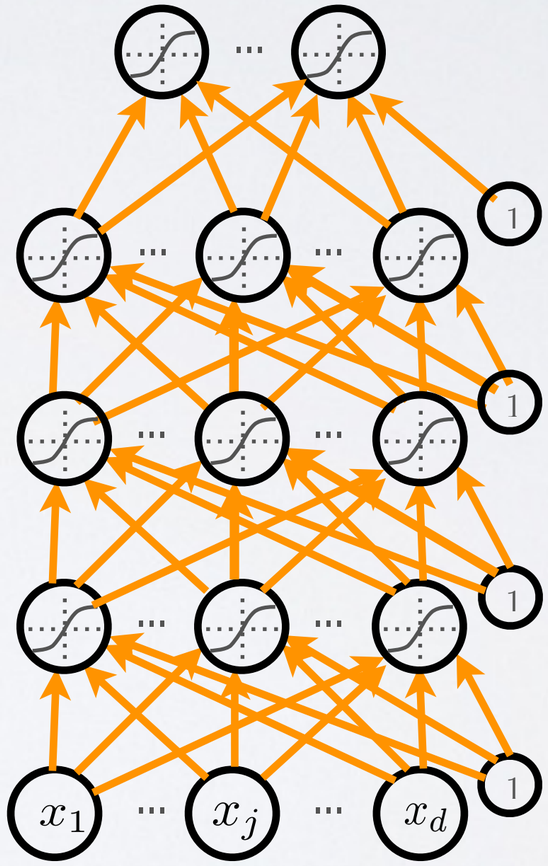
- We call this procedure unsupervised pre-training
  - **first layer:** find hidden unit features that are more common in training inputs than in random inputs
  - **second layer:** find *combinations* of hidden unit features that are more common than random hidden unit features
  - **third layer:** find *combinations of combinations* of ...
  - etc.

- Pre-training initializes the parameters in a region such that the near local optima overfit less the data

FINE-TUNING

Topics: fine-tuning

- Once all layers are pre-trained
  - add output layer
  - train the whole network using supervised learning
- Supervised learning is performed as in a regular feed-forward network
  - forward propagation, backpropagation and update
- We call this last phase fine-tuning
  - all parameters are "tuned" for the supervised task at hand
  - representation is adjusted to be more discriminative



DEEP LEARNING

Topics: pseudocode

- for $l{=}1$ to L
  - build unsupervised training set (with $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$):

$$\mathcal{D} = \left\{ \mathbf{h}^{(l-1)}(\mathbf{x}^{(t)}) \right\}_{t=1}^{T}$$

  - train "greedy module" (RBM, autoencoder) on $\mathcal{D}$
  - use hidden layer weights and biases of greedy module to initialize the deep network parameters $\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$

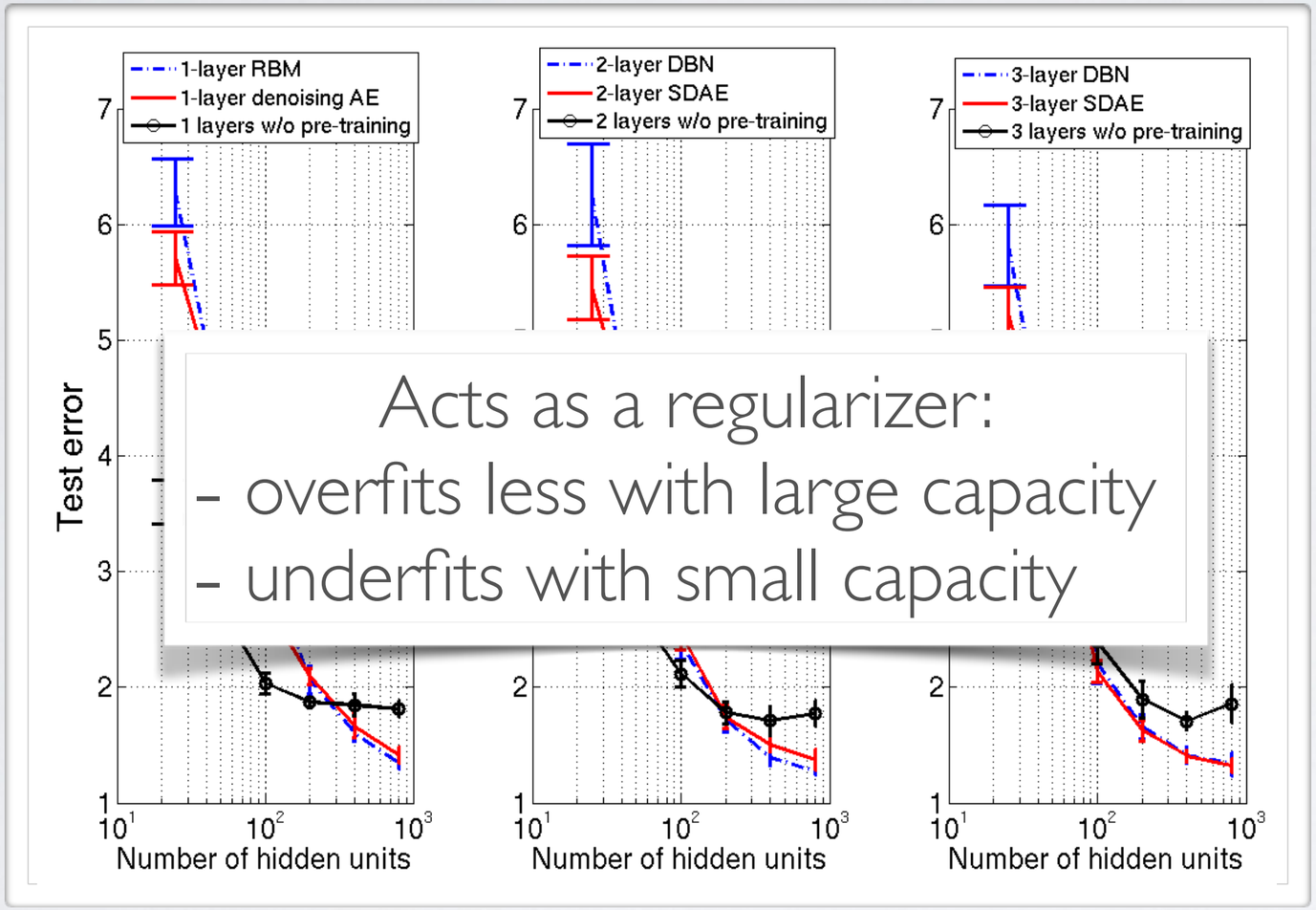
pre-training

- Initialize $\mathbf{W}^{(L+1)}$, $\mathbf{b}^{(L+1)}$ randomly (as usual)
- Train the whole neural network using (supervised) stochastic gradient descent (with backprop)

fine-tuning

DEEP LEARNING

Topics: impact of initialization



Why Does Unsupervised Pre-training Help Deep Learning?
Erhan, Bengio, Courville, Manzagol, Vincent and Bengio, 2011

Neural networks

Deep learning - dropout

DEEP LEARNING

Topics: why training is hard

- Depending on the problem, one or the other situation will tend to dominate

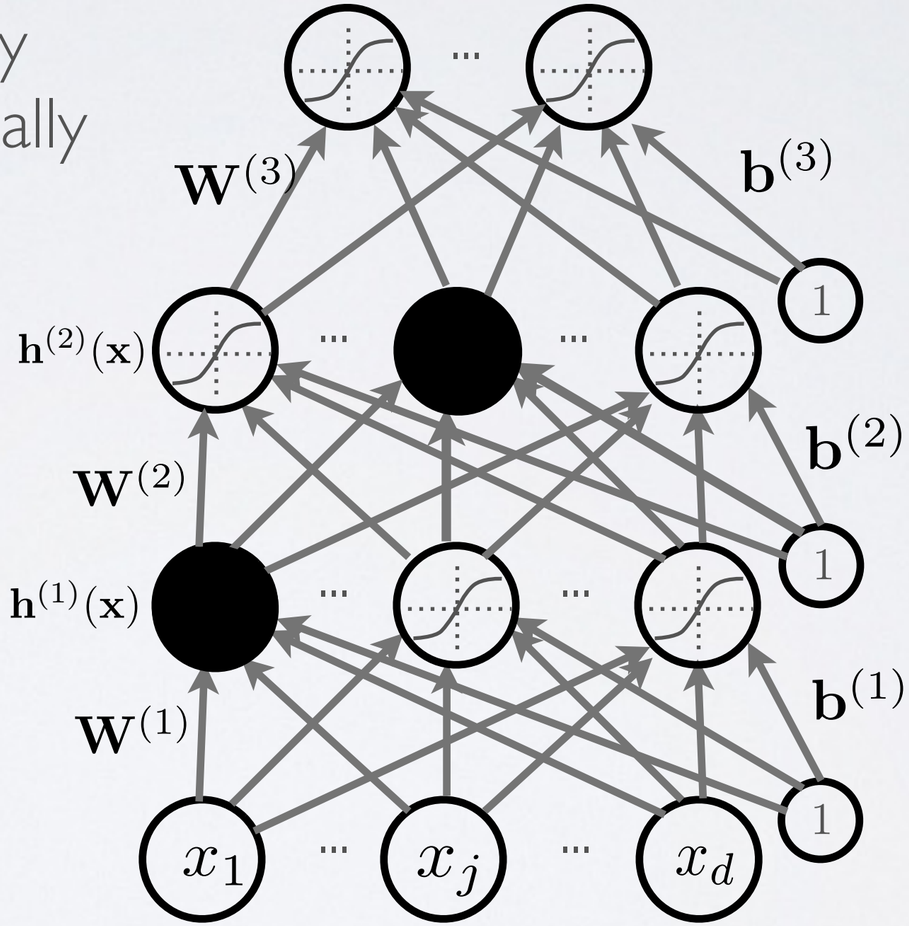
- If first hypothesis (underfitting): better optimize
  - use better optimization methods
  - use GPUs

- If second hypothesis (overfitting): use better regularization
  - unsupervised learning
  - stochastic «dropout» training

DROPOUT

Topics: dropout

- Idea: «cripple» neural network by removing hidden units stochastically
  - each hidden unit is set to 0 with probability 0.5
  - hidden units cannot co-adapt to other units
  - hidden units must be more generally useful



- Could use a different dropout probability, but 0.5 usually works well

DROPOUT

Topics: dropout

- Use random binary masks $\mathbf{m}^{(k)}$
  - layer pre-activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

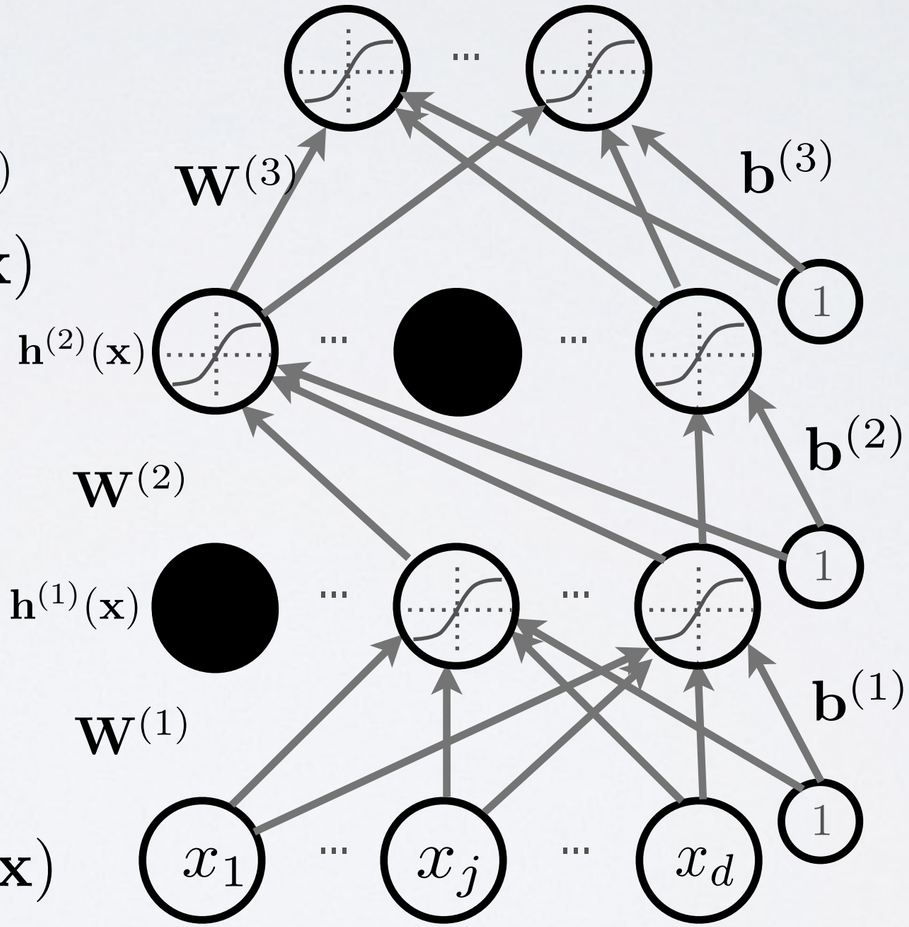
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \odot \mathbf{m}^{(k)}$$

  - output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



DROPOUT

Topics: dropout backpropagation

- This assumes a forward propagation has been made before
  - compute output gradient (before activation)

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

  - for k from $L+1$ to 1
    - compute gradients of hidden layer parameter

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \iff \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \iff \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

    - compute gradient of hidden layer below

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff \mathbf{W}^{(k)^\top} \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)$$

    - compute gradient of hidden layer below (before activation)

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff \left(\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k-1)}(\mathbf{x})_j), \ldots] \odot \mathbf{m}^{(k-1)}$$

includes the mask $\mathbf{m}^{(k-1)}$

DROPOUT

Topics: test time classification

- At test time, we replace the masks by their expectation
  - this is simply the constant vector 0.5 if dropout probability is 0.5
  - for single hidden layer, can show this is equivalent to taking the geometric average of all neural networks, with all possible binary masks
- Can be combined with unsupervised pre-training
- Beats regular backpropagation on many datasets
  - Improving neural networks by preventing co-adaptation of feature detectors. Hinton, Srivastava, Krizhevsky, Sutskever and Salakhutdinov, 2012.

Neural networks

Deep learning - batch normalization

DEEP LEARNING

Topics: why training is hard

- Depending on the problem, one or the other situation will tend to dominate

- If first hypothesis (underfitting): better optimize
  - use better optimization methods
  - use GPUs

- If second hypothesis (overfitting): use better regularization
  - unsupervised learning
  - stochastic «dropout» training

BATCH NORMALIZATION

Topics: batch normalization

- Normalizing the inputs will speed up training
  (Lecun et al. 1998)
  - could normalization also be useful at the level of the hidden layers?

- **Batch normalization** is an attempt to do that
  (Ioffe and Szegedy, 2014)
  - each unit's **pre-**activation is normalized (mean subtraction, stddev division)
  - during training, mean and stddev is computed for **each minibatch**
  - backpropagation **takes into account** the normalization
  - at test time, the **global mean / stddev is used**

BATCH NORMALIZATION

Topics: batch normalization

- **Batch normalization**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learned linear transformation to adapt to non-linear activation function

(γ and β are **trained**)

BATCH NORMALIZATION

Topics: batch normalization

- Why normalize the **pre**-activation?
  - can help keep the pre-activation in a non-saturating regime (though the linear transform $y_i \leftarrow \gamma \widehat{x}_i + \beta$ could cancel this effect)

- Why use minibatches?
  - since hidden units depend on parameters, can't compute mean/stddev once and for all
  - adds stochasticity to training, which might regularize (dropout not as useful)

BATCH NORMALIZATION

Topics: batch normalization

- How to take into account the normalization in backdrop?
  - derivative wrt x_i depends on the partial derivative of the mean and stddev
  - must also update γ and β

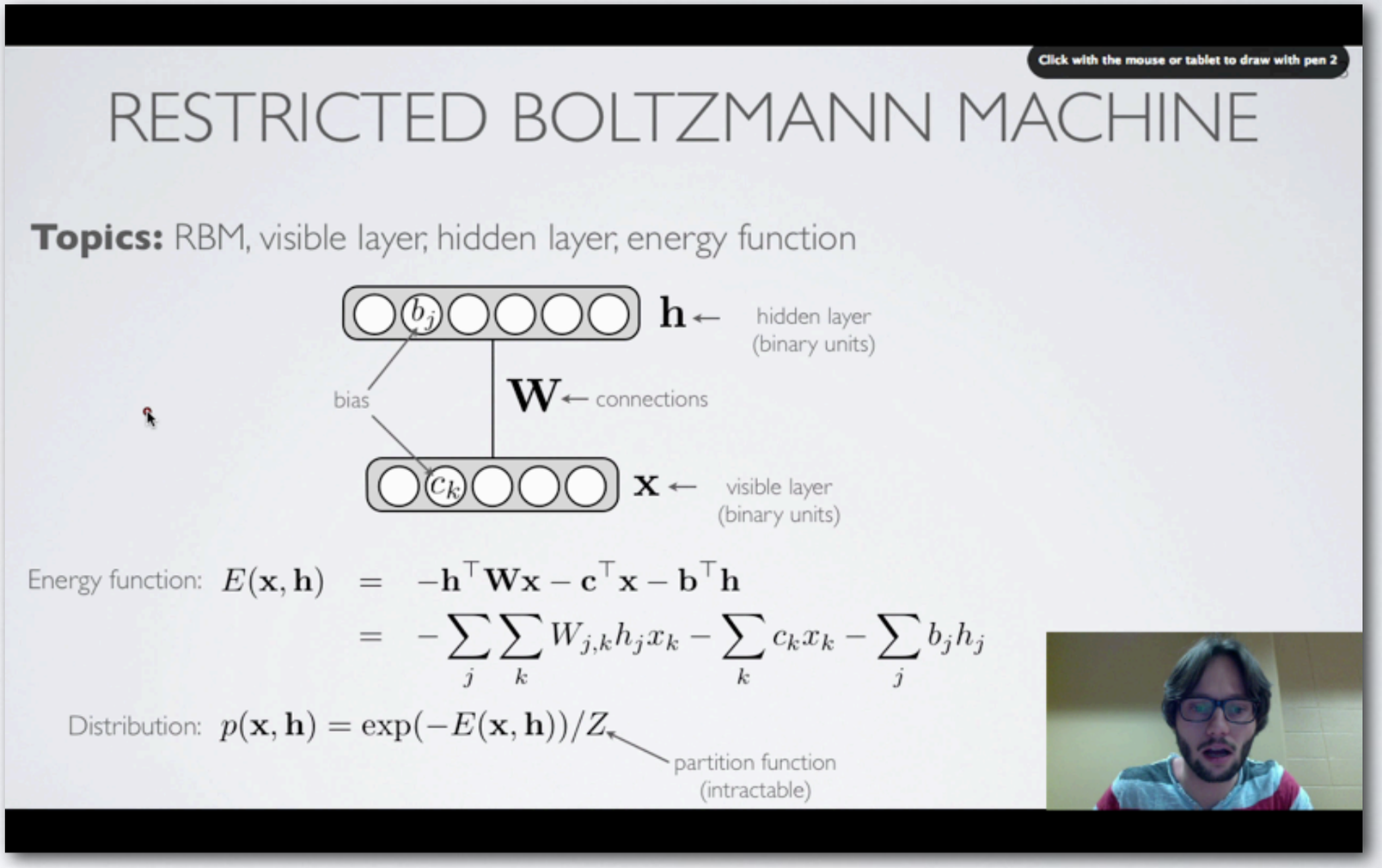
- Why use the global mean stddev at test time?
  - removes the stochasticity of the mean and stddev
  - requires a final phase where, from the first to the last hidden layer
    1. propagate all training data to that layer
    2. compute and store the global mean and stddev of each unit
  - for early stopping, could use a running average

NEURAL NETWORK ONLINE COURSE

Topics: online videos

- covers many other topics: convolutional networks, neural language model, restricted Boltzmann machines, autoencoders, sparse coding, etc.

http://info.usherbrooke.ca/hlarochelle/neural_networks



MERCI!