

# Foundations of Deep Learning

Hugo Larochelle ( @hugo\_larochelle )  
Twitter / Université de Sherbrooke

# NEURAL NETWORK ONLINE COURSE

**Topics:** online videos

- ▶ for a more detailed description of neural networks...
- ▶ ... and much more!

[http://info.usherbrooke.ca/hlarochelle/neural\\_networks](http://info.usherbrooke.ca/hlarochelle/neural_networks)

RESTRICTED BOLTZMANN MACHINE

**Topics:** RBM, visible layer, hidden layer, energy function

The diagram illustrates a Restricted Boltzmann Machine (RBM) architecture. It consists of two layers of binary units. The bottom layer, labeled  $\mathbf{x}$ , represents the visible layer with four units. The top layer, labeled  $\mathbf{h}$ , represents the hidden layer with five units. A vertical vector of weights, labeled  $\mathbf{W}$ , connects every unit in the visible layer to every unit in the hidden layer. Above the hidden layer, a bias vector  $\mathbf{b}$  is shown, with a specific unit  $b_j$  labeled. Below the visible layer, a bias vector  $\mathbf{c}$  is shown, with a specific unit  $c_k$  labeled. A cursor icon is positioned near the visible layer.

Energy function:

$$\begin{aligned} E(\mathbf{x}, \mathbf{h}) &= -\mathbf{h}^\top \mathbf{W} \mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h} \\ &= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j \end{aligned}$$

Distribution:

$$p(\mathbf{x}, \mathbf{h}) = \exp(-E(\mathbf{x}, \mathbf{h}))/Z$$

partition function  
(intractable)

# NEURAL NETWORK ONLINE COURSE

**Topics:** online videos

- ▶ for a more detailed description of neural networks...
- ▶ ... and much more!

[http://info.usherbrooke.ca/hlarochelle/neural\\_networks](http://info.usherbrooke.ca/hlarochelle/neural_networks)

RESTRICTED BOLTZMANN MACHINE

**Topics:** RBM, visible layer, hidden layer, energy function

The diagram illustrates a Restricted Boltzmann Machine (RBM) architecture. It consists of two layers of binary units. The bottom layer, labeled  $\mathbf{x}$ , represents the visible layer with four units. The top layer, labeled  $\mathbf{h}$ , represents the hidden layer with five units. A vertical vector of weights, labeled  $\mathbf{W}$ , connects each unit in the visible layer to each unit in the hidden layer. Arrows point from the labels  $b_j$  and  $c_k$  to their respective units in the hidden and visible layers, indicating they are bias units. A label 'bias' points to the  $c_k$  unit. A cursor icon is positioned near the visible layer.

Energy function:

$$\begin{aligned} E(\mathbf{x}, \mathbf{h}) &= -\mathbf{h}^\top \mathbf{W} \mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h} \\ &= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j \end{aligned}$$

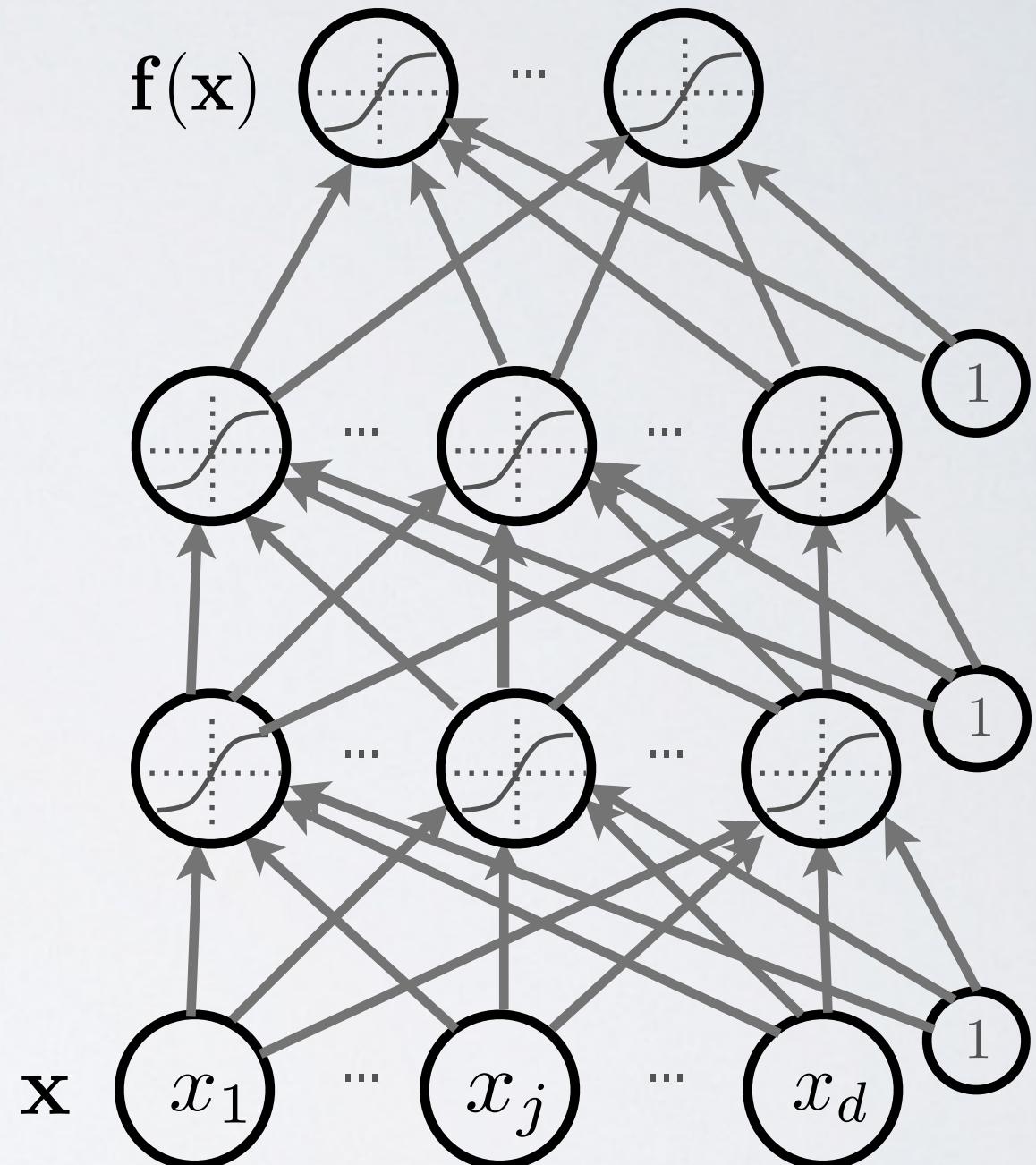
Distribution:

$$p(\mathbf{x}, \mathbf{h}) = \exp(-E(\mathbf{x}, \mathbf{h}))/Z$$

partition function  
(intractable)

# FOUNDATIONS OF DEEP LEARNING

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



# Foundations of Deep Learning

Making predictions with feedforward neural networks

# NEURAL NETWORK

**Topics:** multilayer neural network

- Could have  $L$  hidden layers:

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

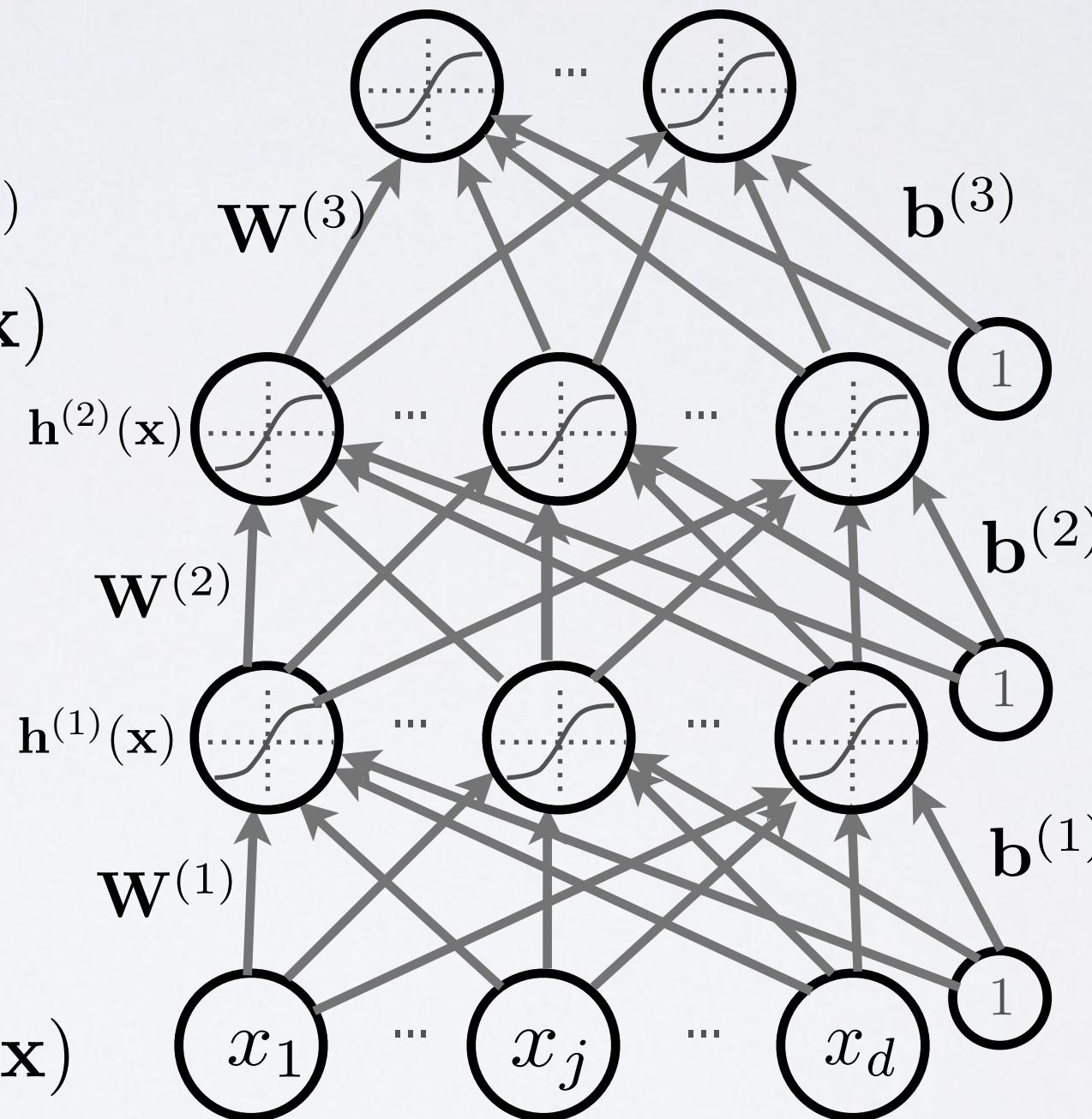
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

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

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

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

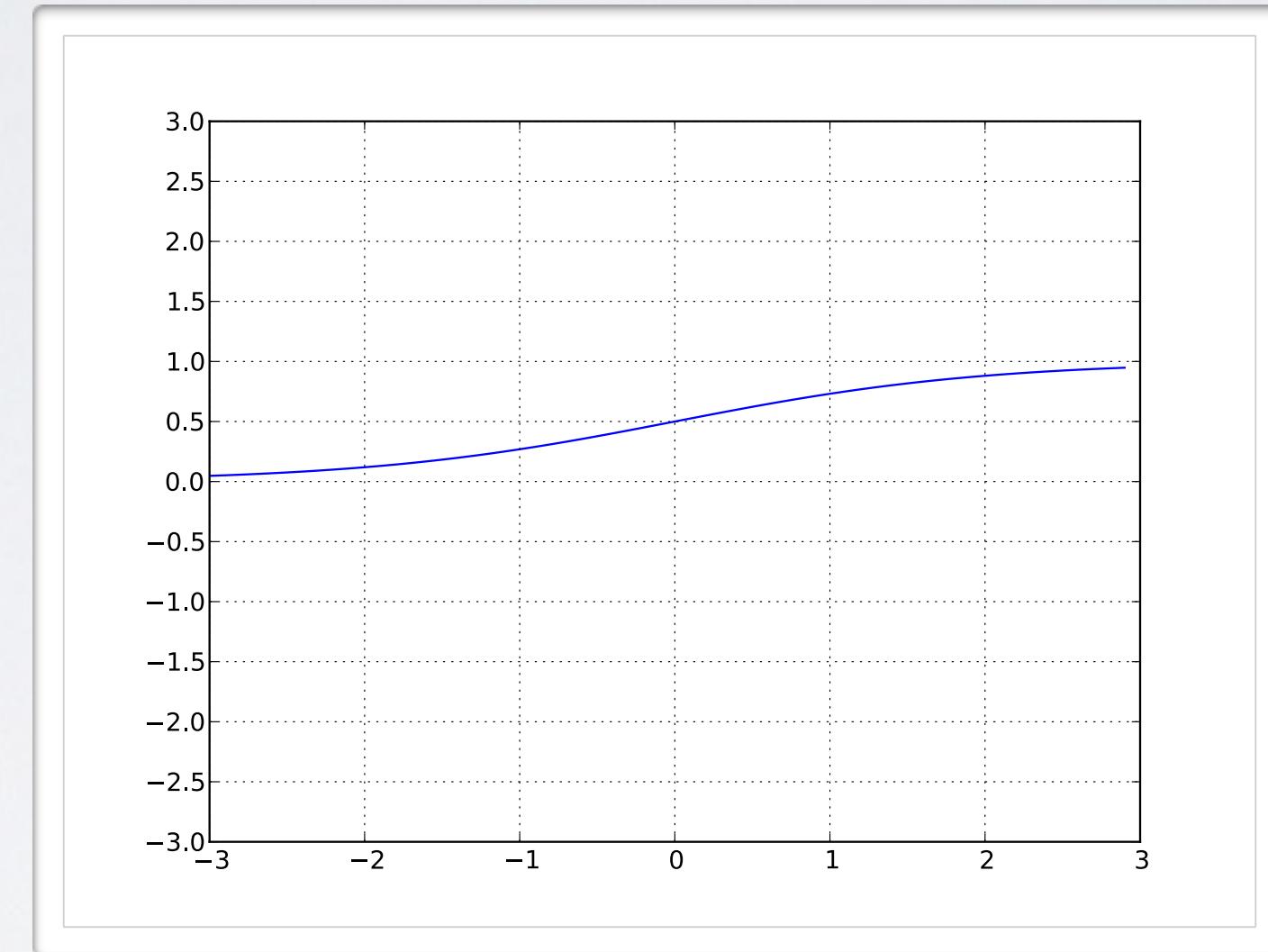
$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



# ACTIVATION FUNCTION

**Topics:** sigmoid activation function

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

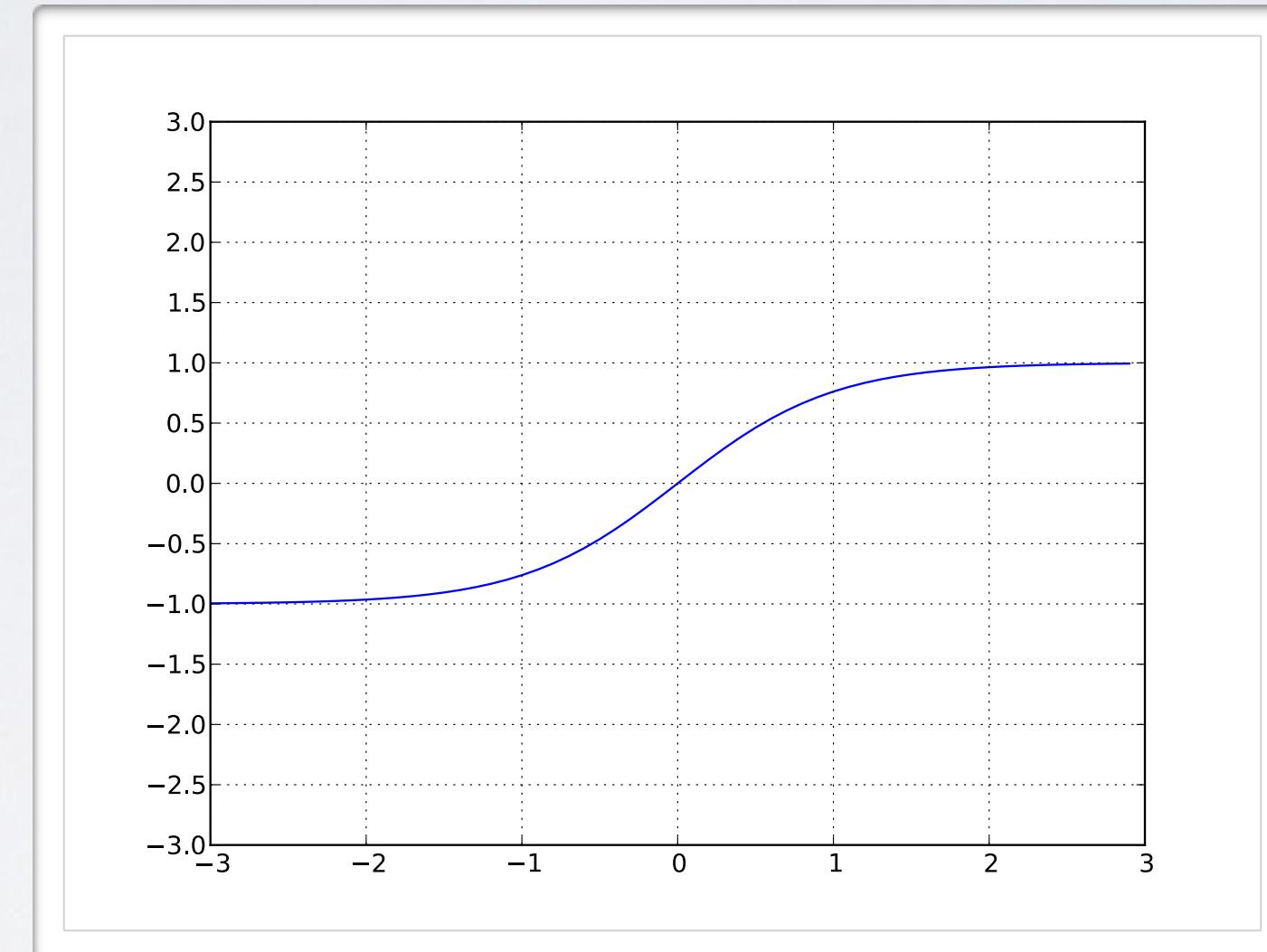


$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

# 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

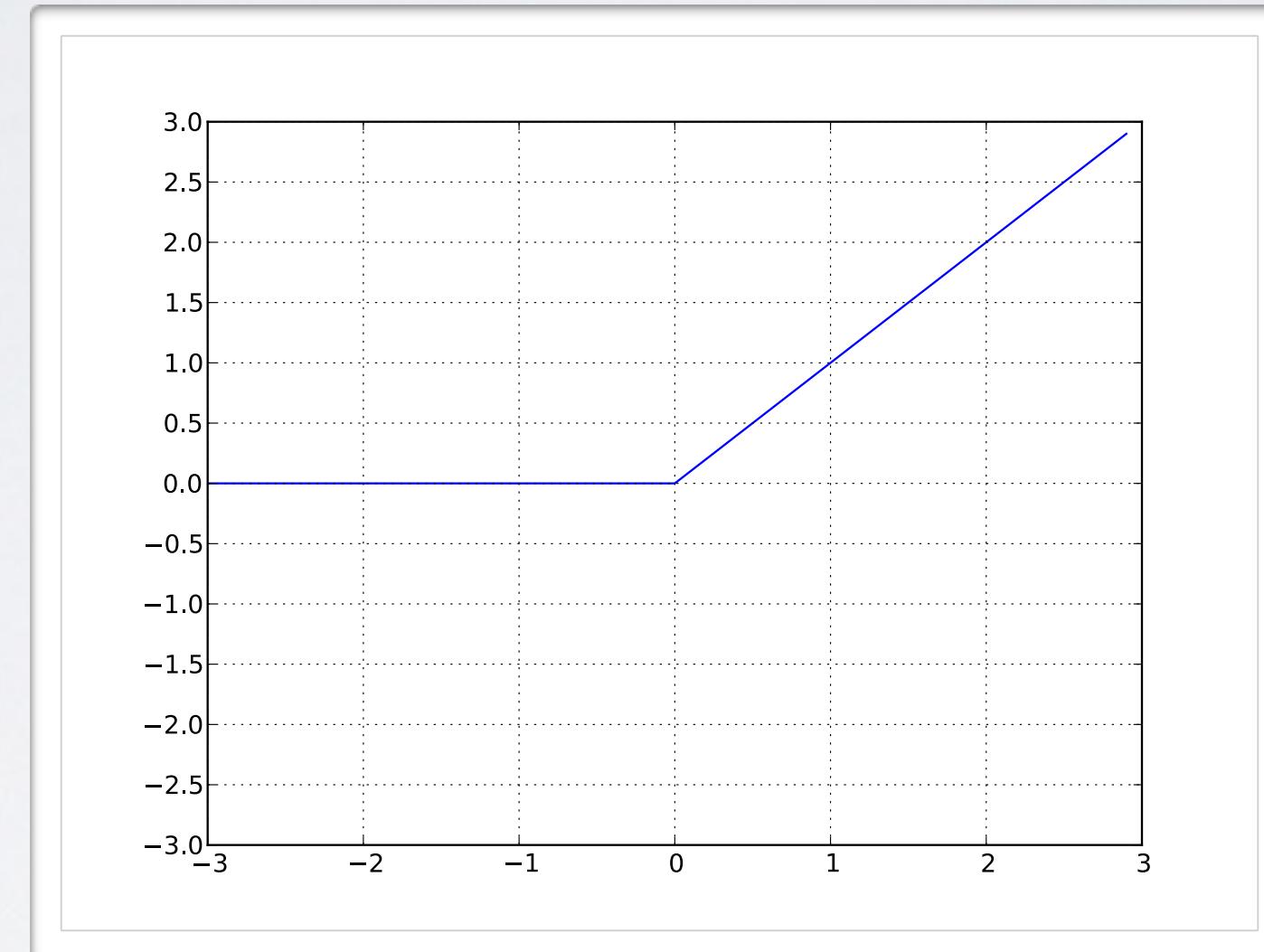


$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

# 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



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

# ACTIVATION FUNCTION

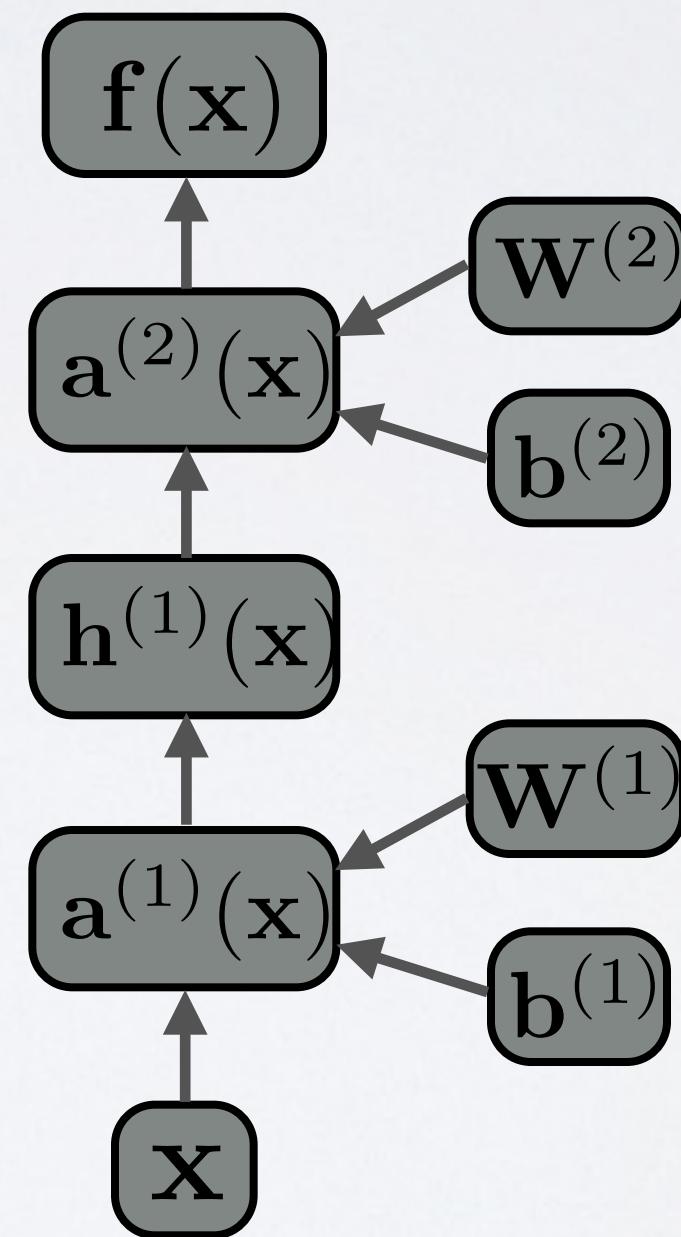
**Topics:** softmax activation function

- 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:
$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[ \frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^T$$
  - ▶ strictly positive
  - ▶ sums to one
- Predicted class is the one with highest estimated probability

# 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 parents
  - ▶ calling the fprop method of each box in the right order yield forward propagation



# 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!

# Foundations of Deep Learning

Training feedforward neural networks

# MACHINE LEARNING

**Topics:** empirical risk minimization, regularization

- Empirical (structural) 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  $\boldsymbol{\theta}$  ( $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$ )

  - ▶ for N epochs

  - 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 method for  $\boldsymbol{\theta}$

# 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

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

natural log ( $\ln$ )

The diagram shows a bracket above the term  $\log f(\mathbf{x})_c$  with the label "natural log ( $\ln$ )". Two arrows point from the top vertex of the bracket to the left and right sides of the  $\log$  term, indicating that the natural logarithm is being applied to the entire term.

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

# BACKPROPAGATION

**Topics:** backpropagation algorithm

- Use the chain rule to efficiently compute gradients, *top to bottom*

- ▶ 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

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

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

- compute gradient of hidden layer below

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

- compute gradient of hidden layer below (before activation)

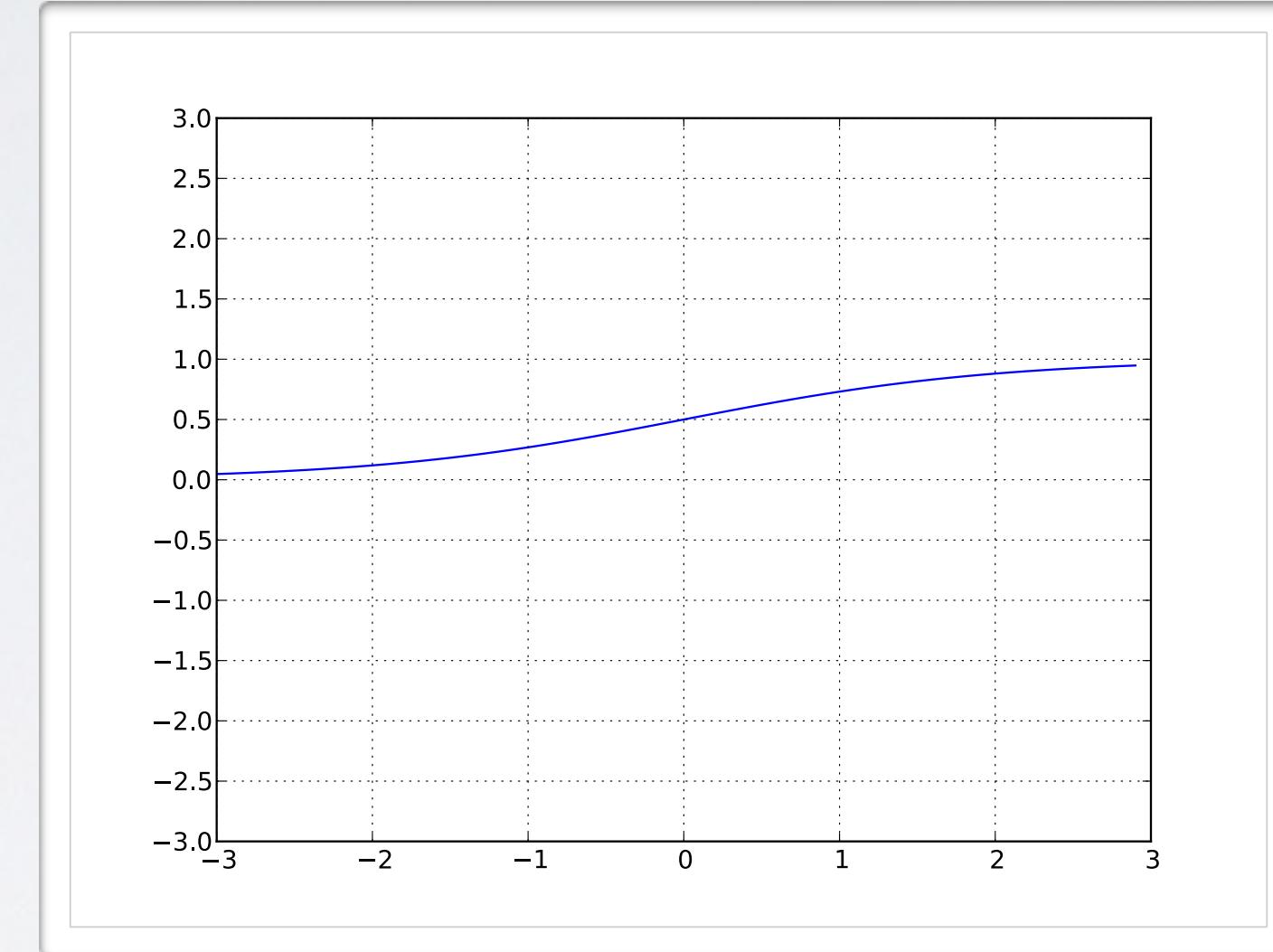
$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \iff (\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y) \odot [\dots, g'(a^{(k-1)}(\mathbf{x})_j), \dots]$$

# ACTIVATION FUNCTION

**Topics:** sigmoid activation function gradient

- Partial derivative:

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



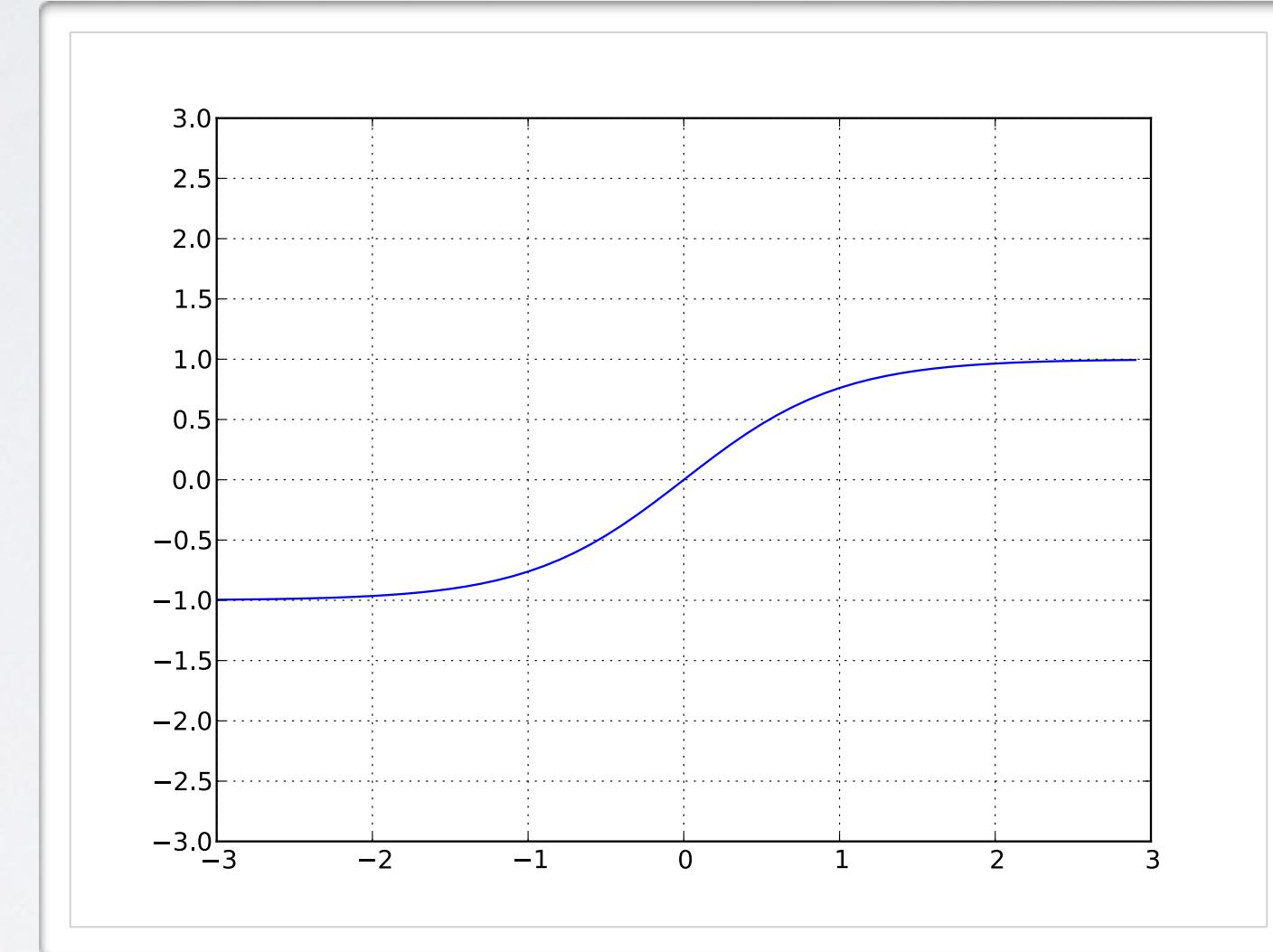
$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

# ACTIVATION FUNCTION

**Topics:** tanh activation function gradient

- Partial derivative:

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



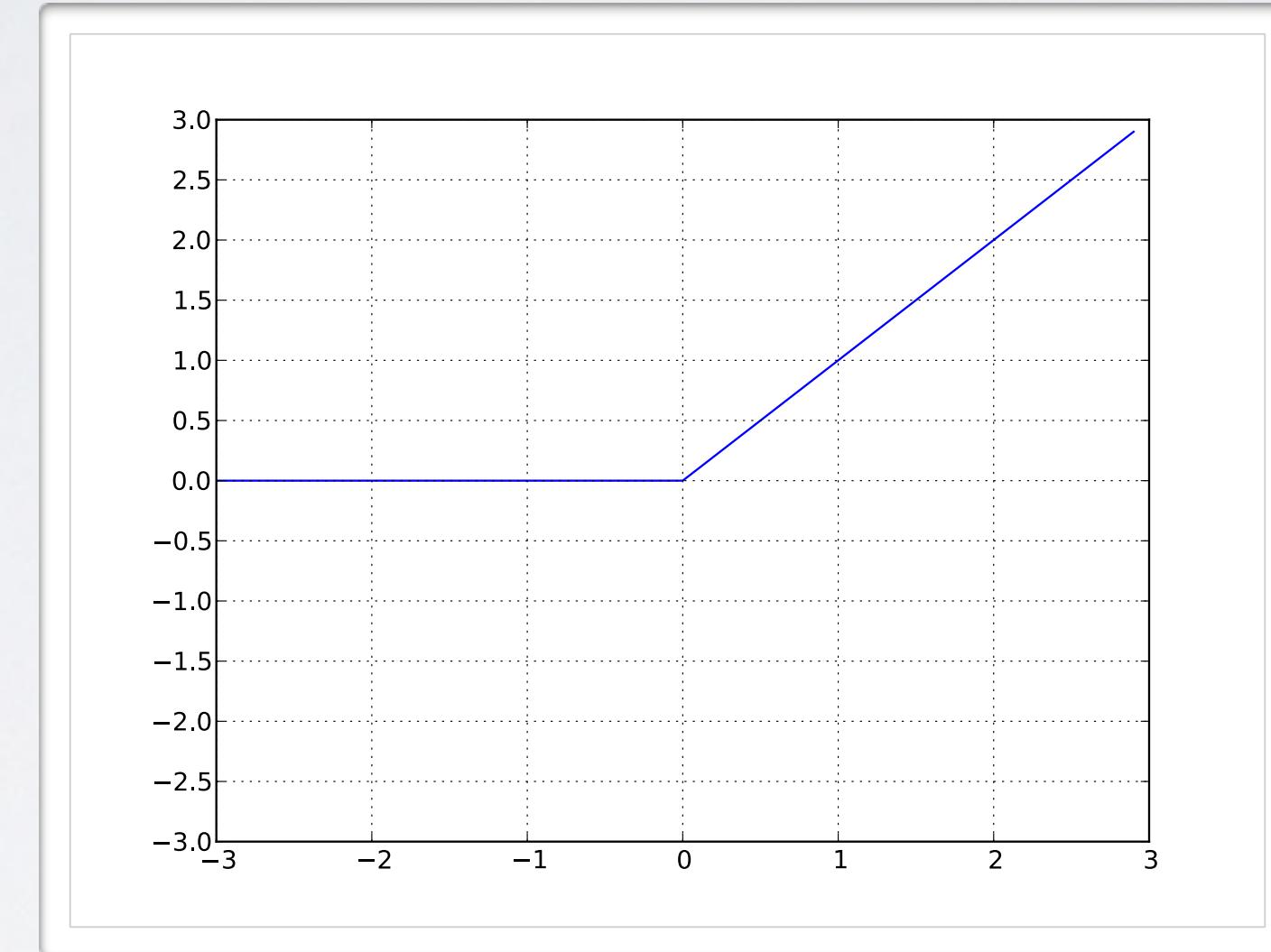
$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

# ACTIVATION FUNCTION

**Topics:** rectified linear activation function gradient

- Partial derivative:

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

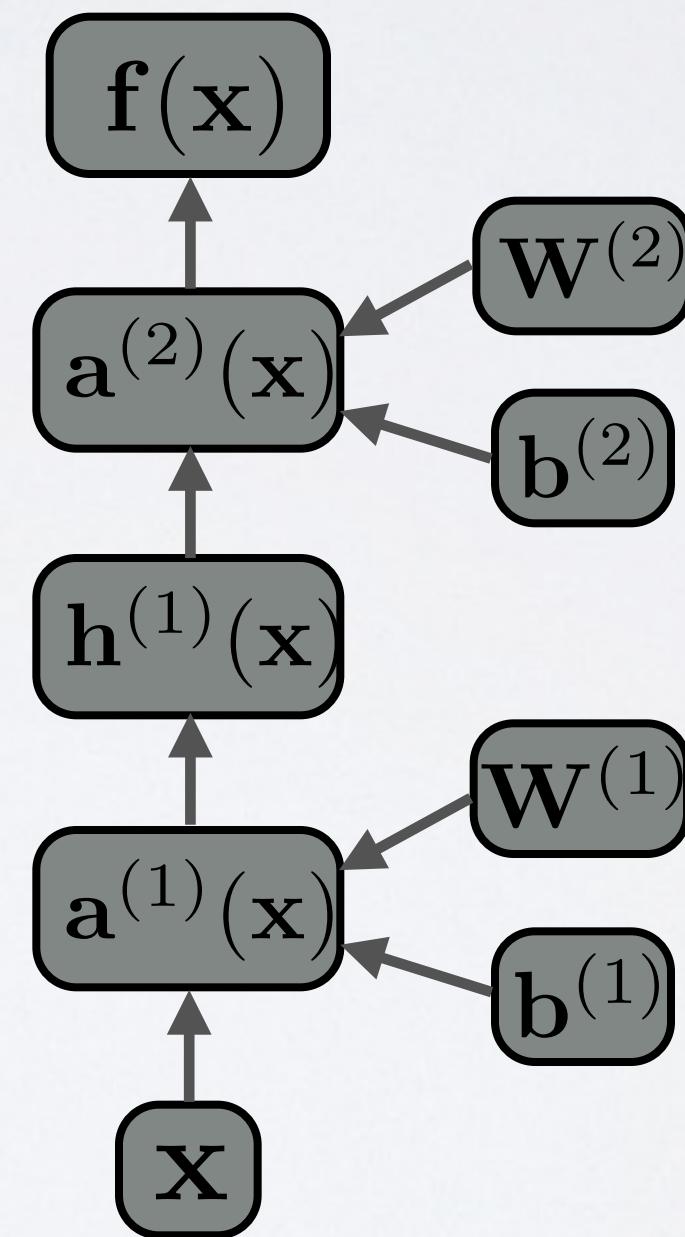


$$g(a) = \text{reclin}(a) = \max(0, a)$$

# FLOW GRAPH

## Topics: automatic differentiation

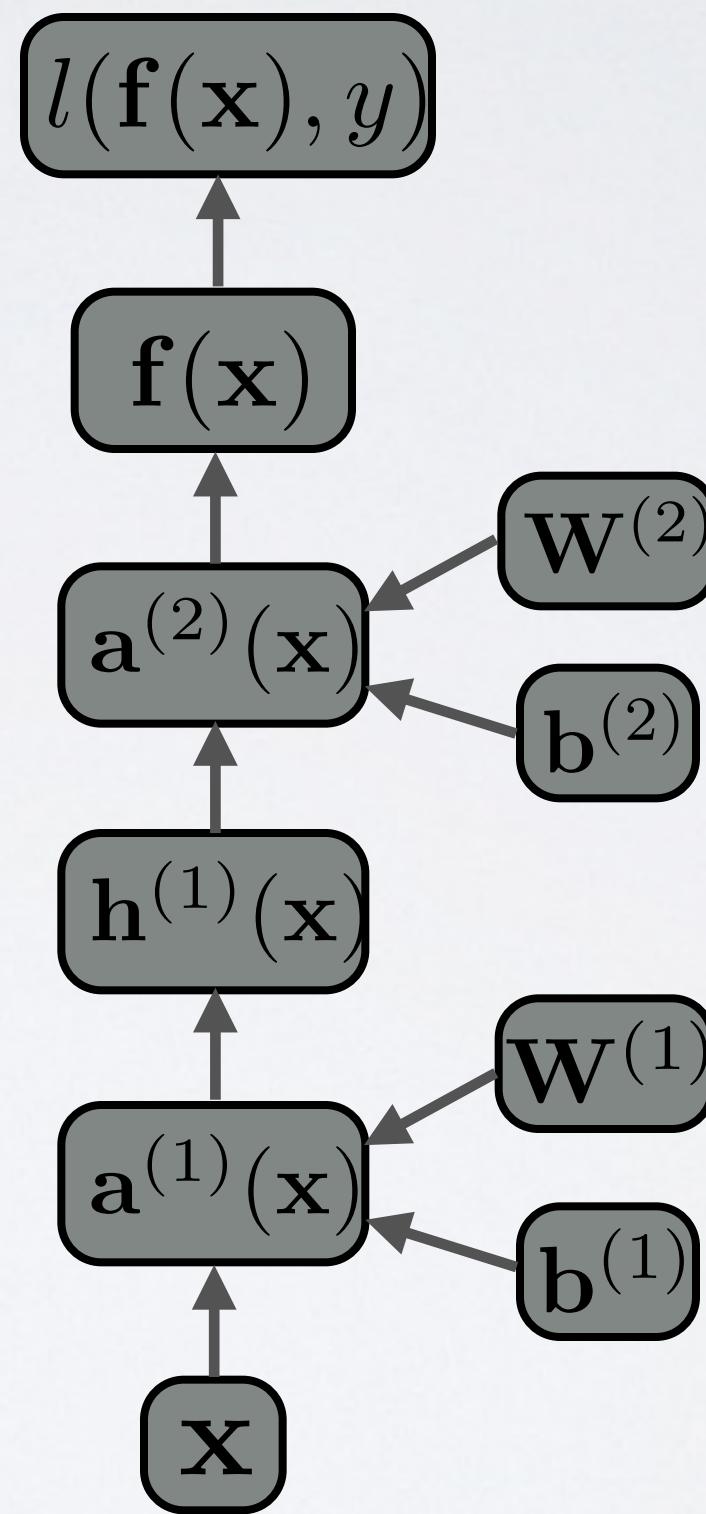
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

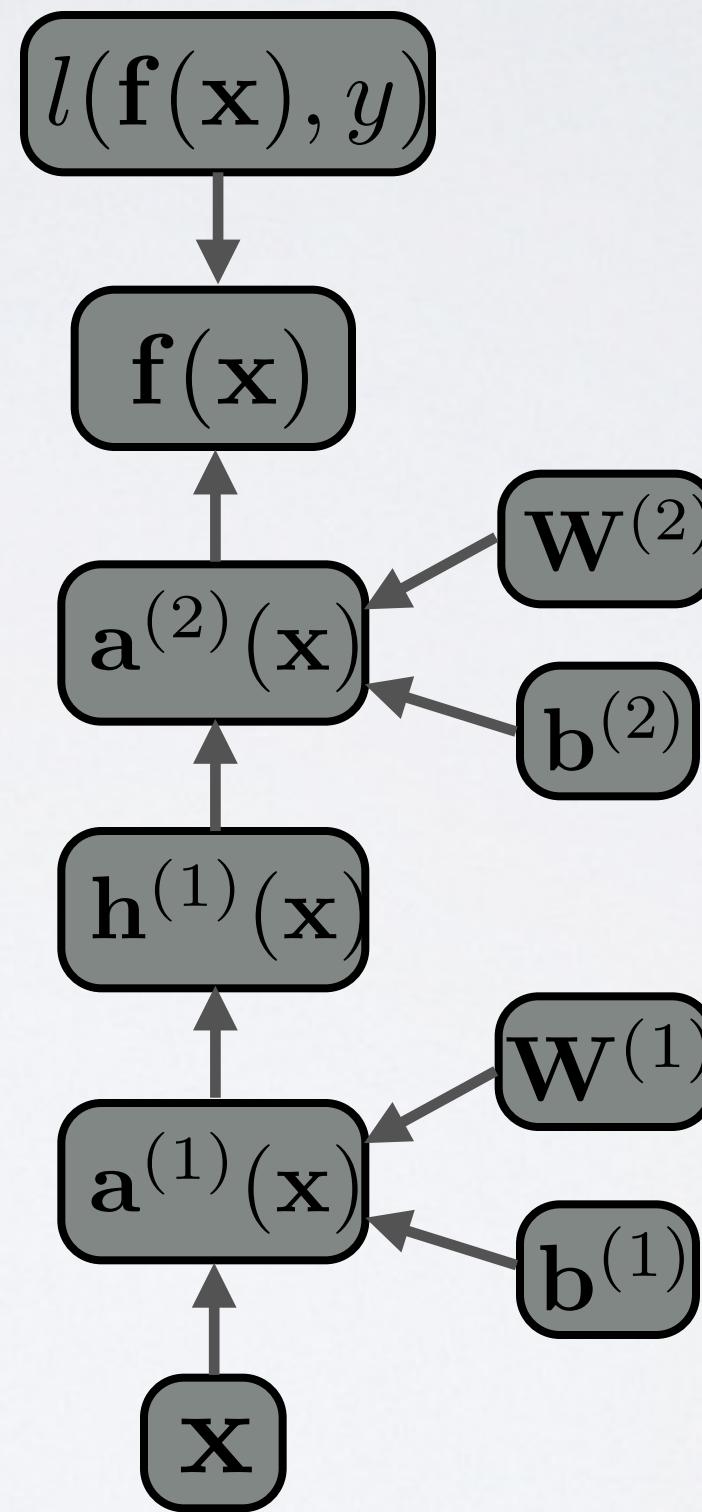
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

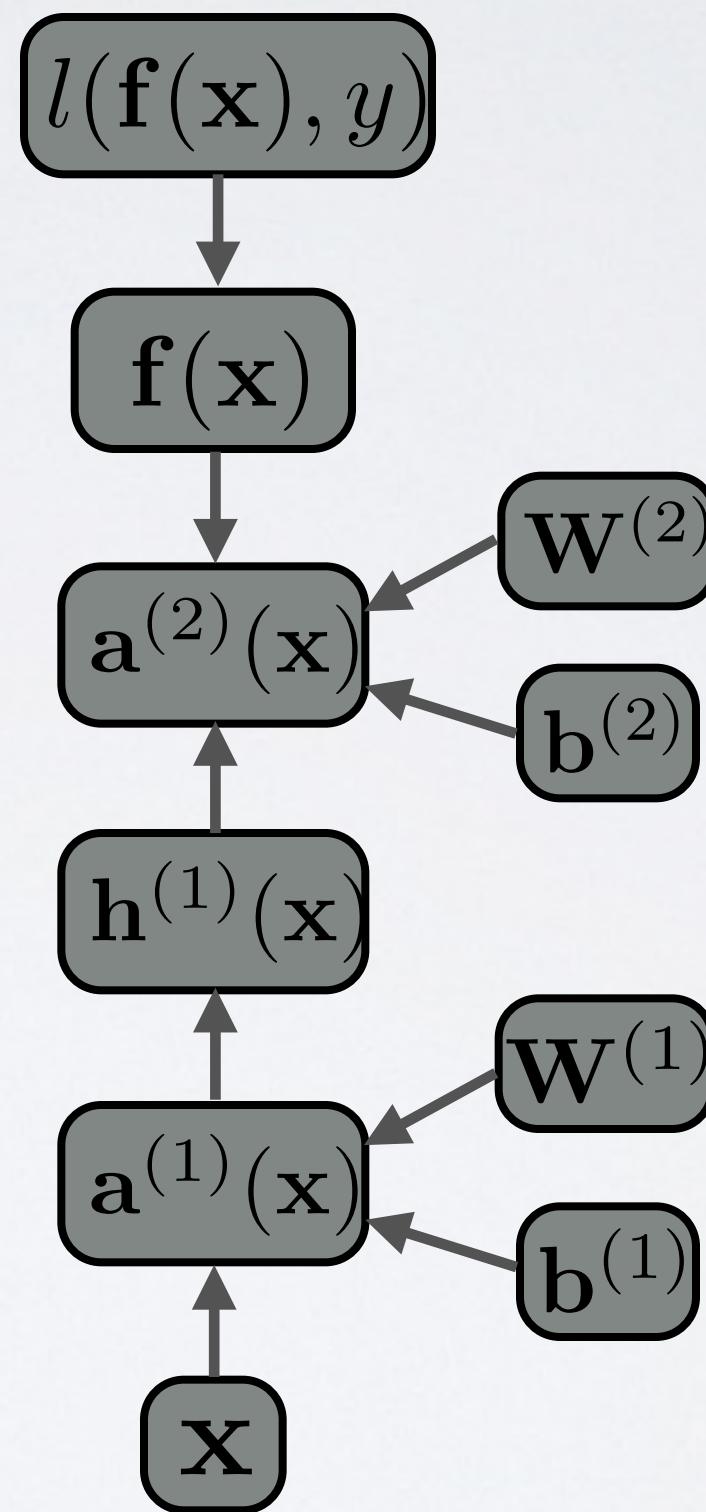
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

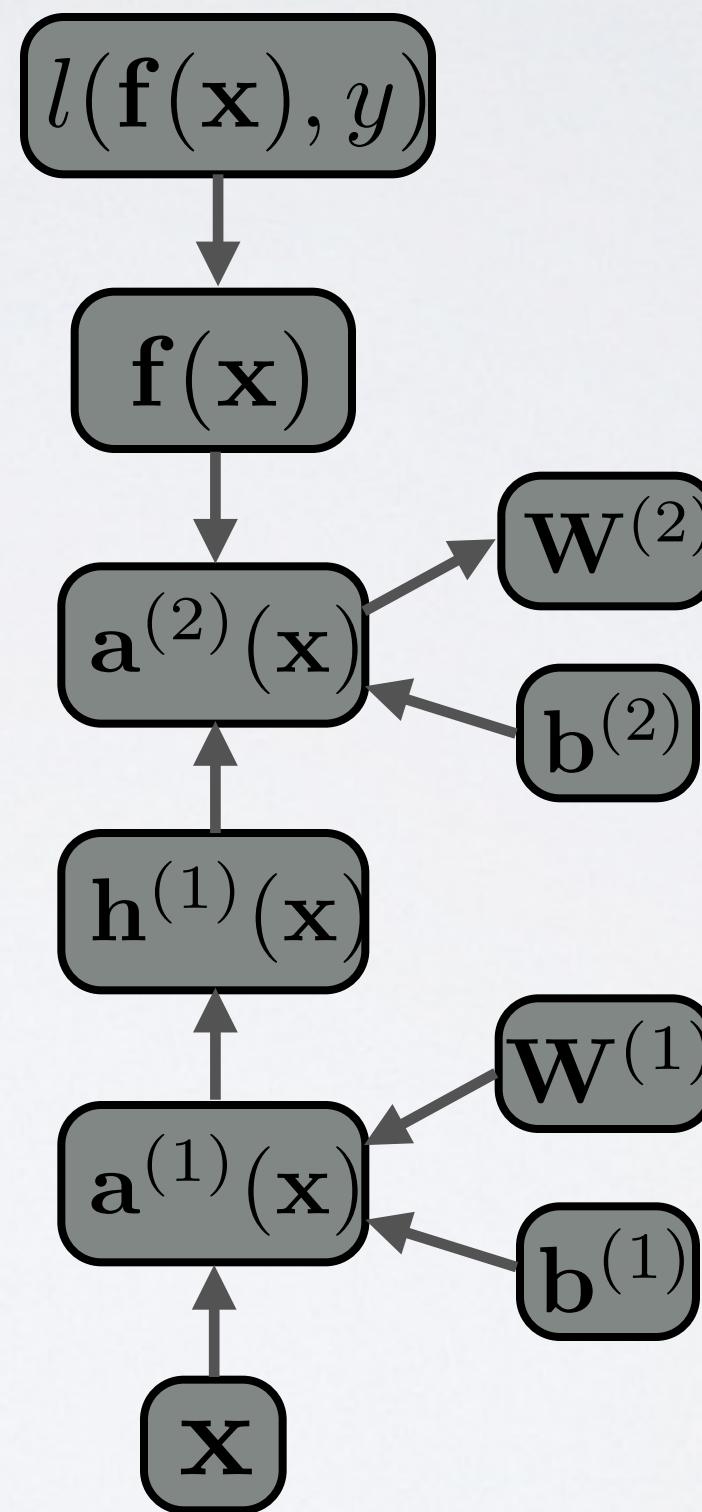
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

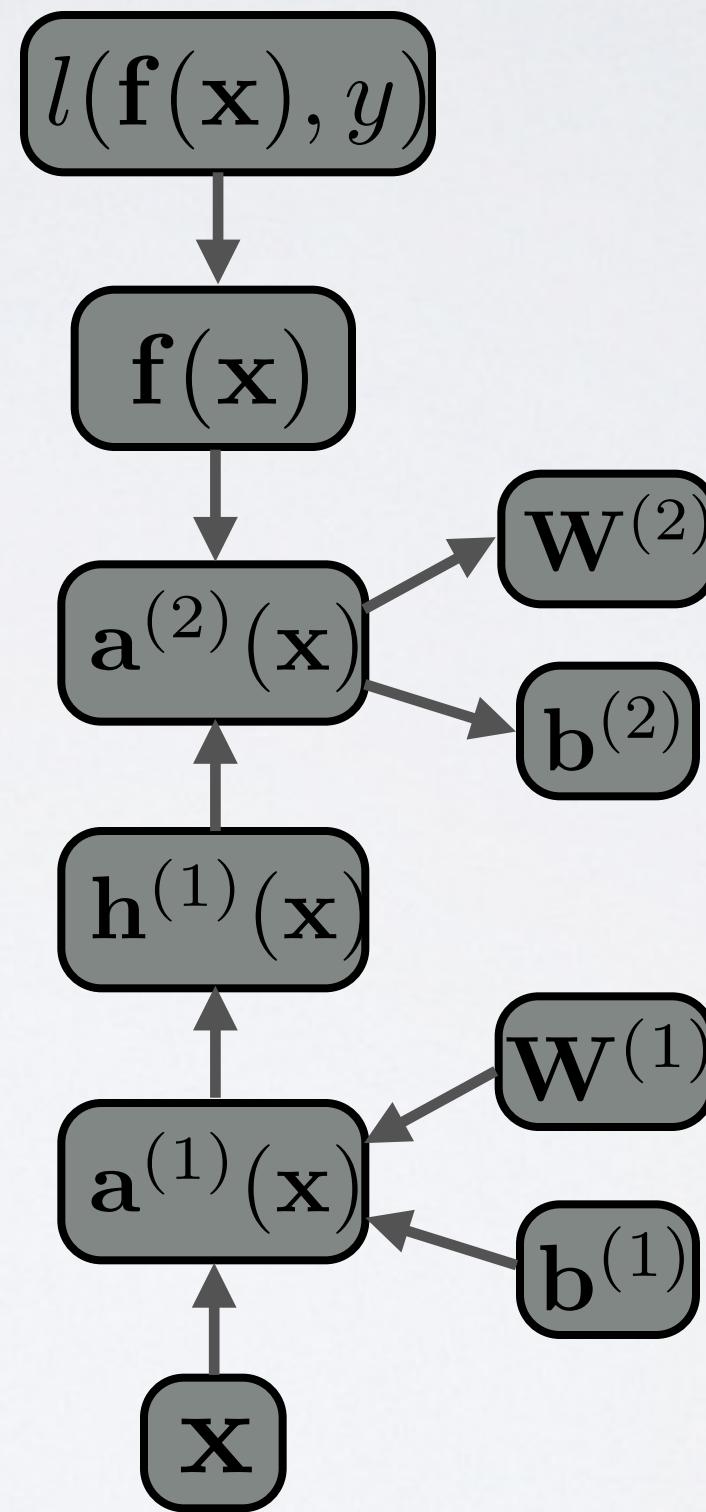
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

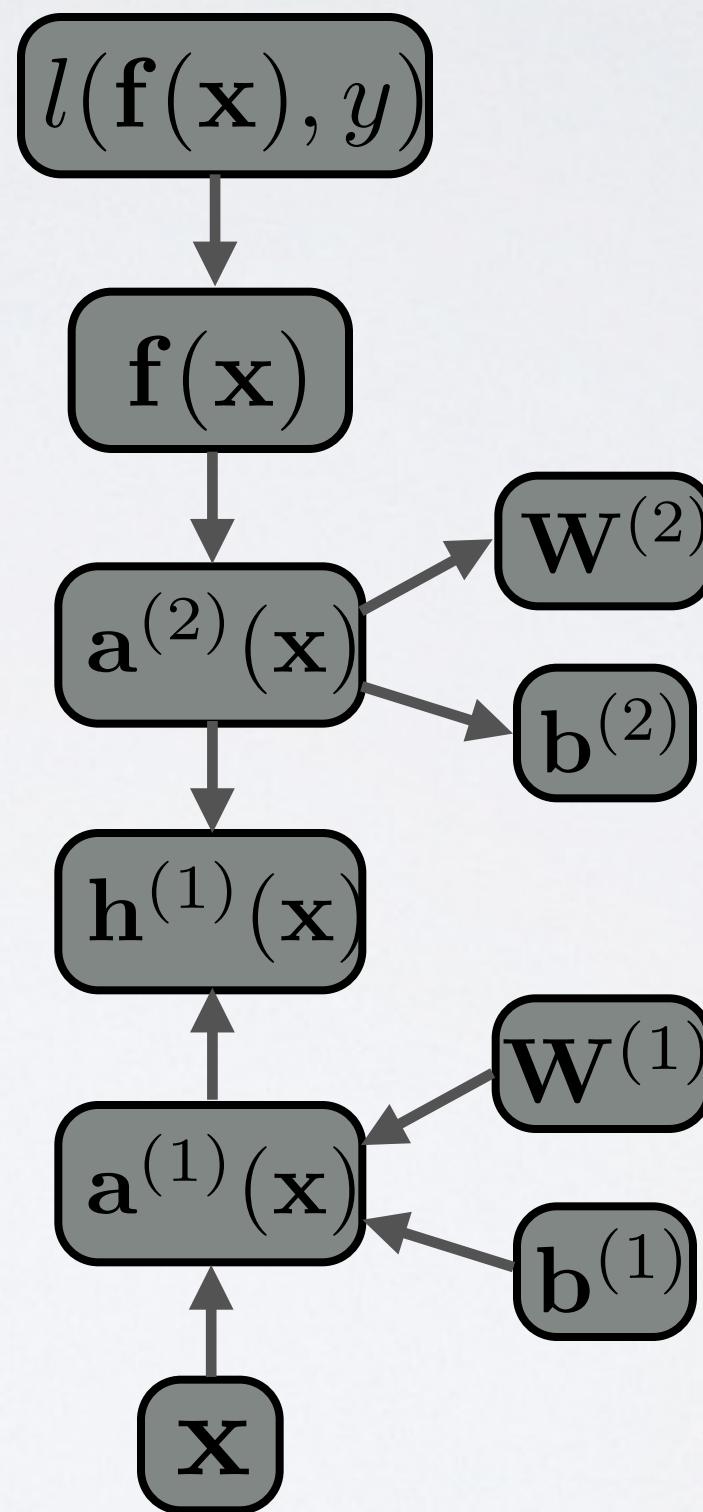
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

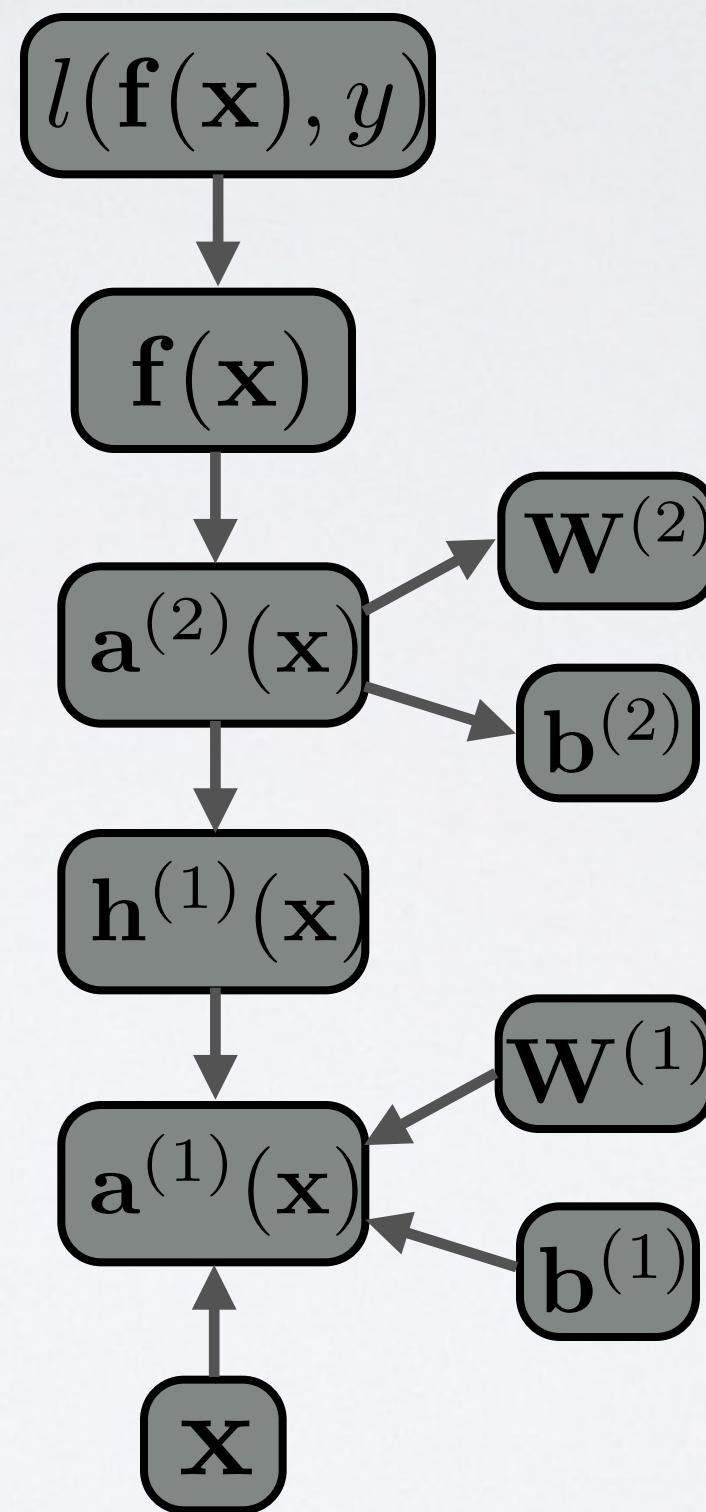
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

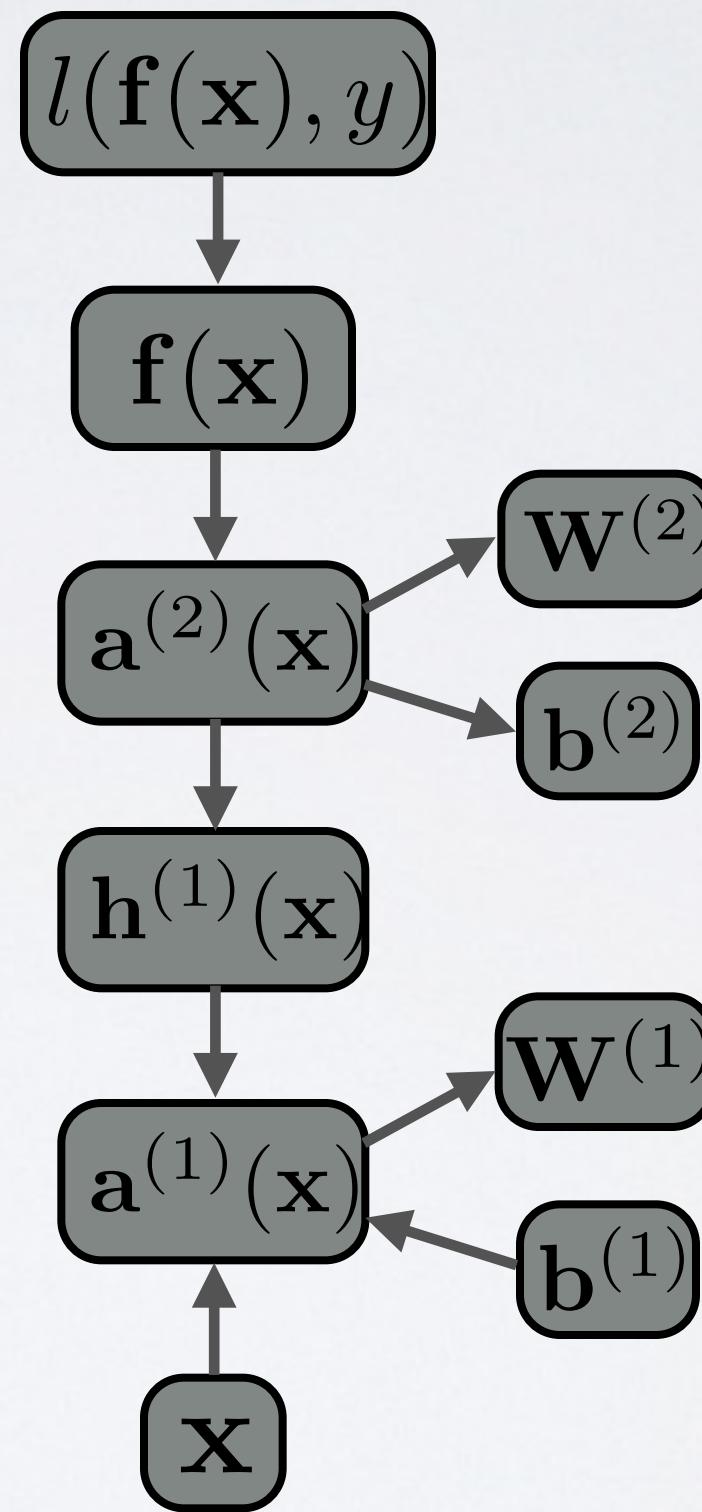
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

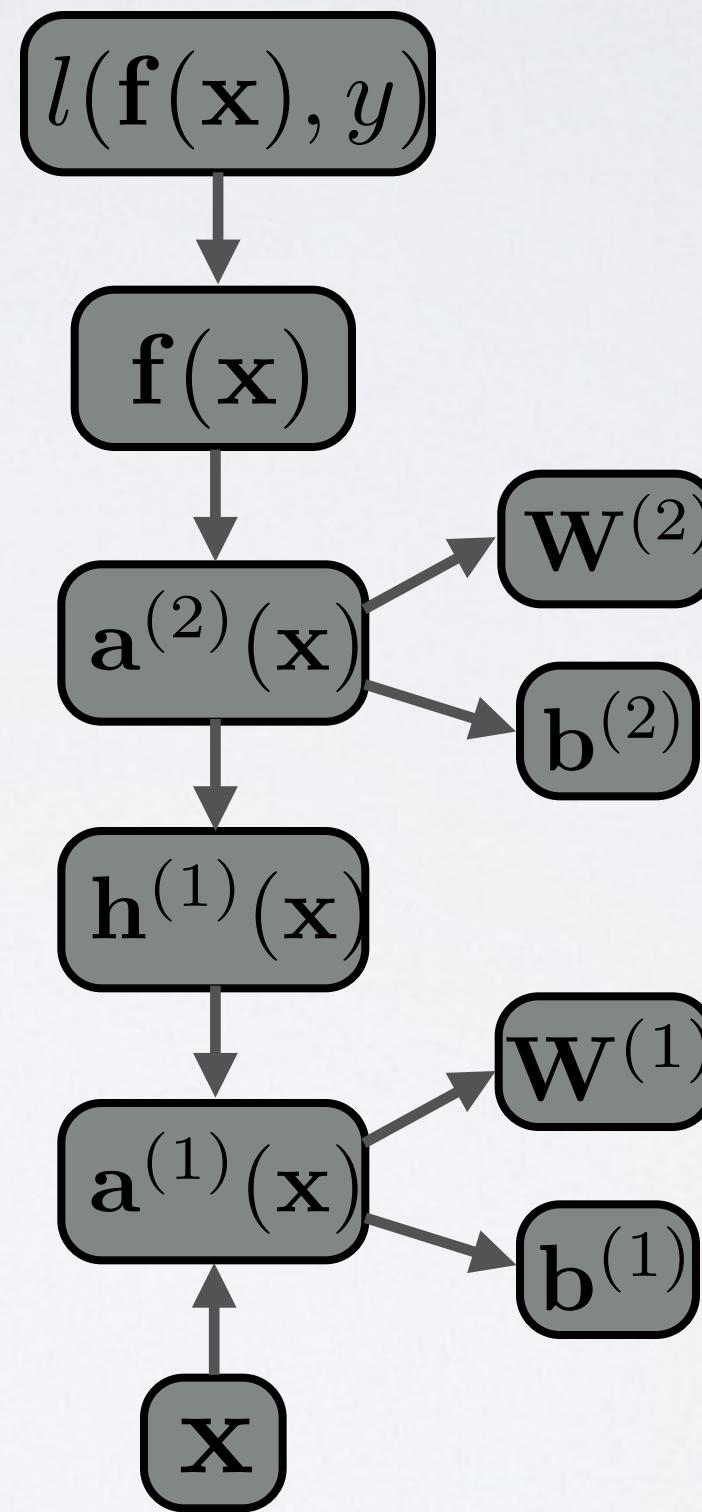
- Each object also has a bprop method
  - ▶ it computes the gradient of the loss with respect to each parent
  - ▶ fprop depends on the fprop of a box's parents, while bprop depends the bprop of a box's children
- By calling bprop in the reverse order, we get backpropagation
  - ▶ only need to reach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



# FLOW GRAPH

**Topics:** automatic differentiation

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



# REGULARIZATION

**Topics:** L2 regularization

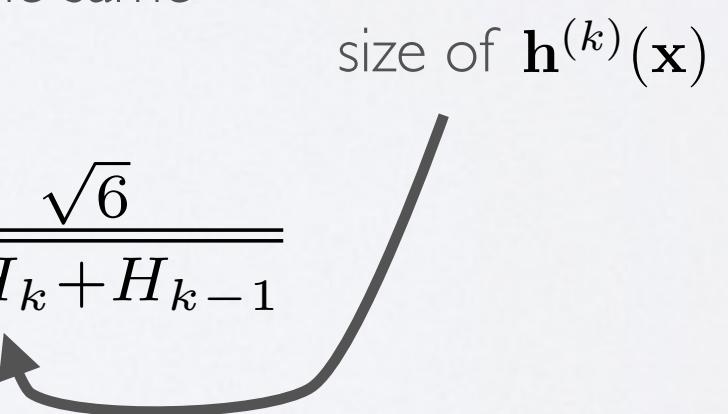
$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left( W_{i,j}^{(k)} \right)^2 = \sum_k \|\mathbf{W}^{(k)}\|_F^2$$

- 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

# 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  $U[-b, b]$  where  $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$ 
    - 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)



# MODEL SELECTION

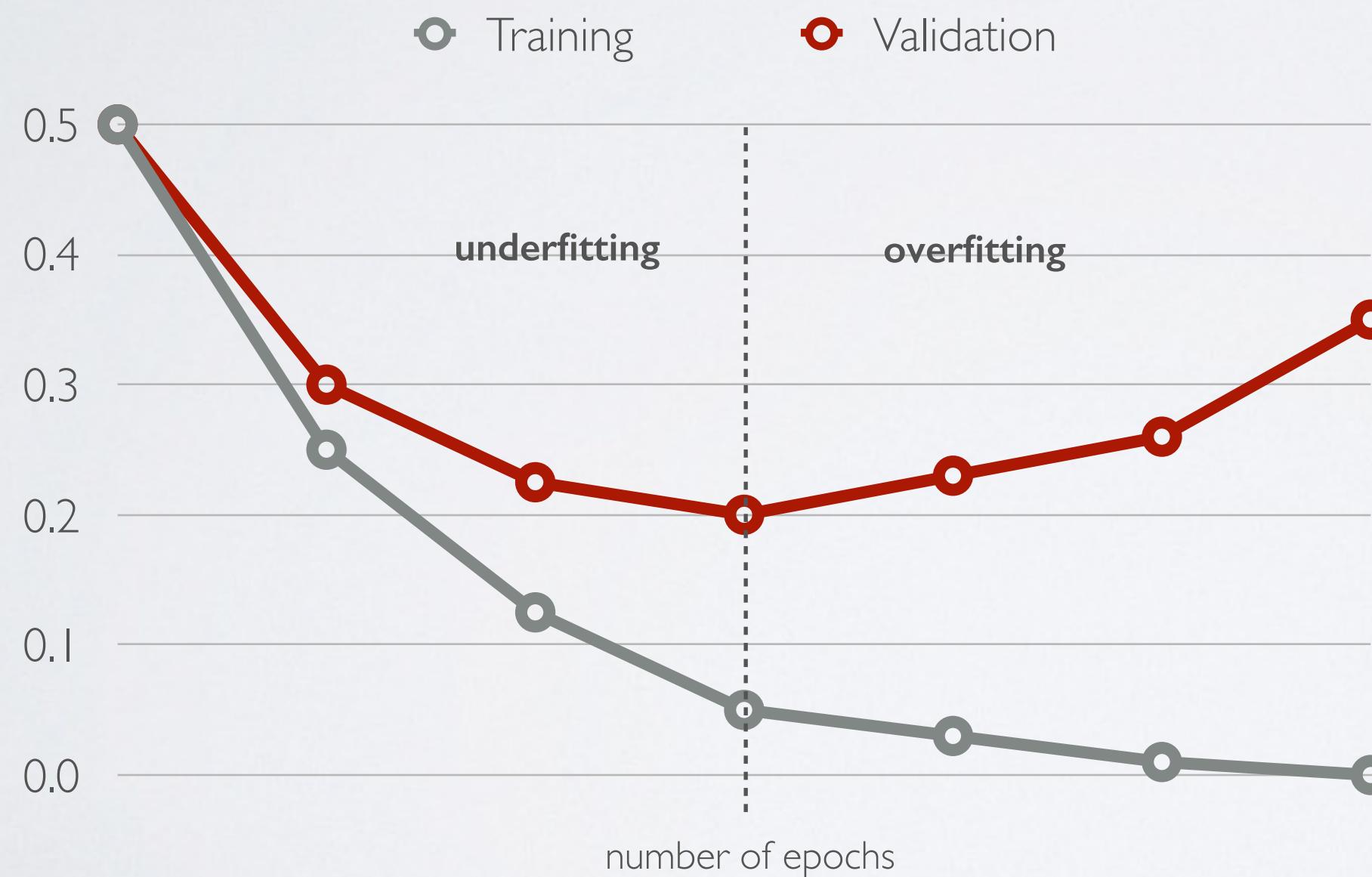
**Topics:** grid search, random search

- 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** (not the test 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)



# 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:

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

- ▶ 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

$$\gamma^{(t)} = \gamma^{(t-1)} + (\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}))^2$$

$$\bar{\nabla}_{\theta}^{(t)} = \frac{\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

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

$$\gamma^{(t)} = \beta \gamma^{(t-1)} + (1 - \beta) (\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}))^2$$

$$\bar{\nabla}_{\theta}^{(t)} = \frac{\nabla_{\theta} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

- ▶ **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  $\epsilon$  to the parameter
- $f(x - \epsilon)$  would be the loss if you subtract  $\epsilon$  to the parameter

# DEBUGGING ON SMALL DATASET

**Topics:** debugging on small dataset

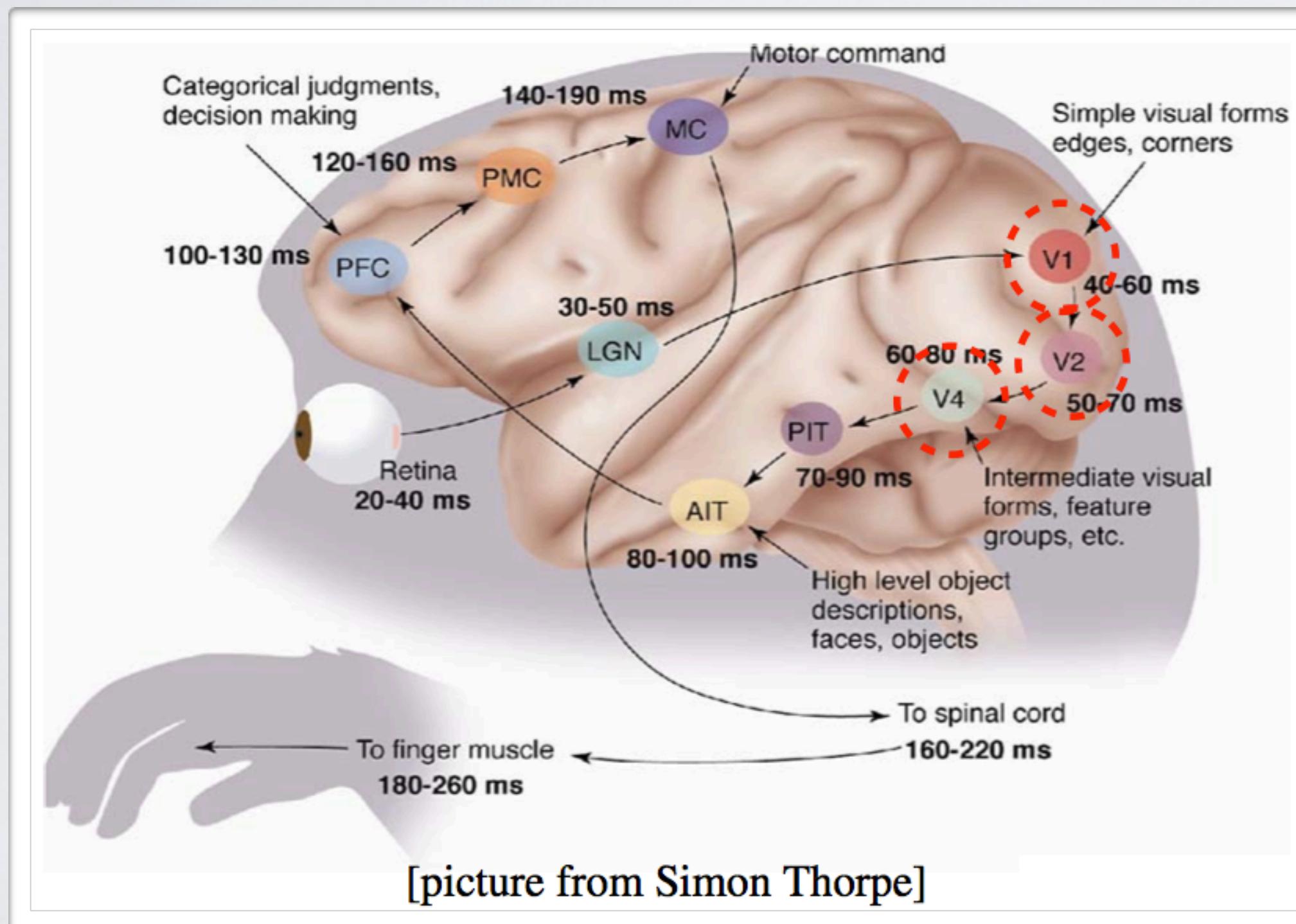
- Next, make sure your model is able to (over)fit on a very small dataset (~50 examples)
- If not, investigate the following situations:
  - ▶ Are some of the units saturated, even before the first update?
    - scale down the initialization of your parameters for these units
    - properly normalize 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

# Foundations of Deep Learning

Training deep feed-forward neural networks

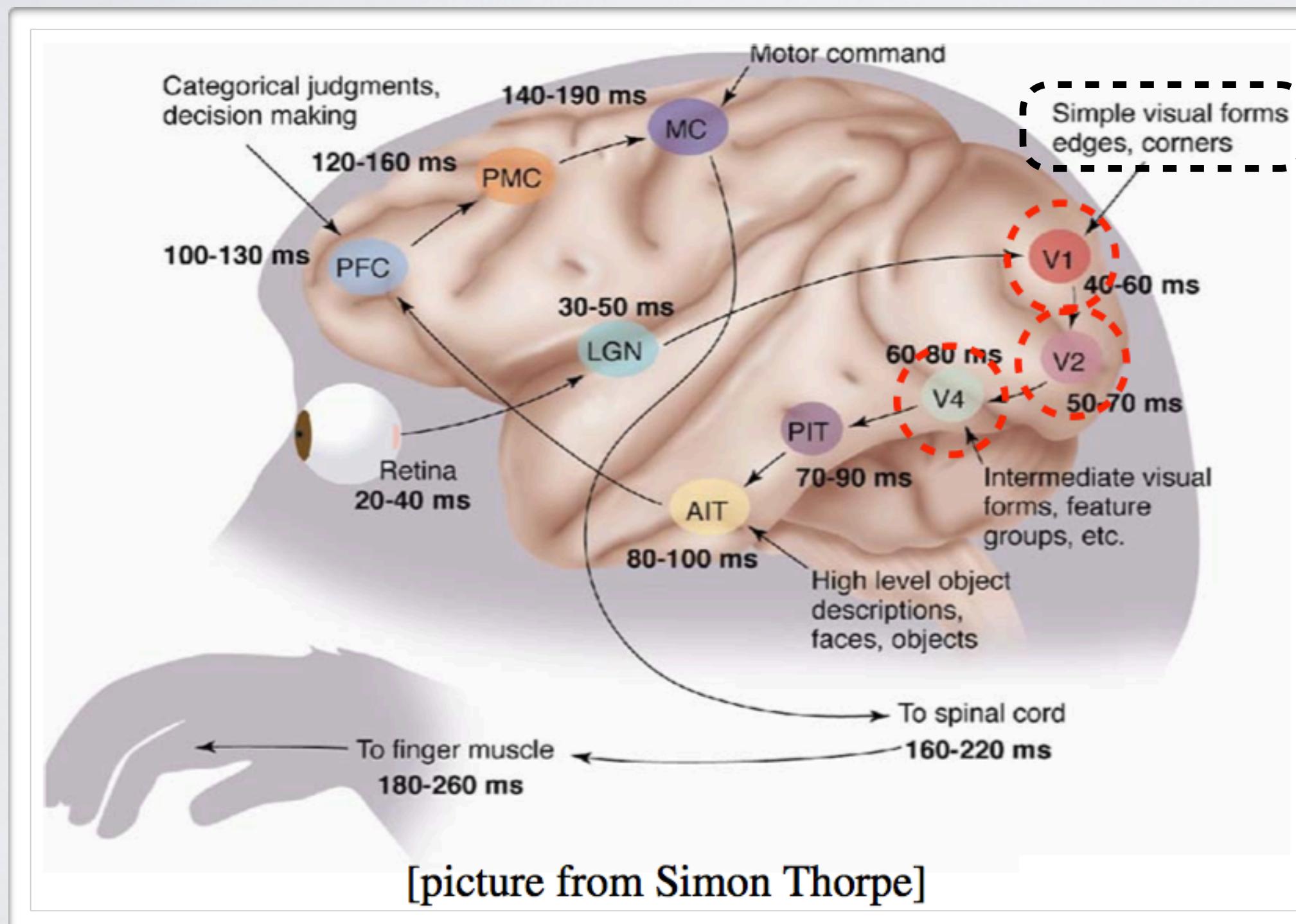
# DEEP LEARNING

**Topics:** inspiration from visual cortex



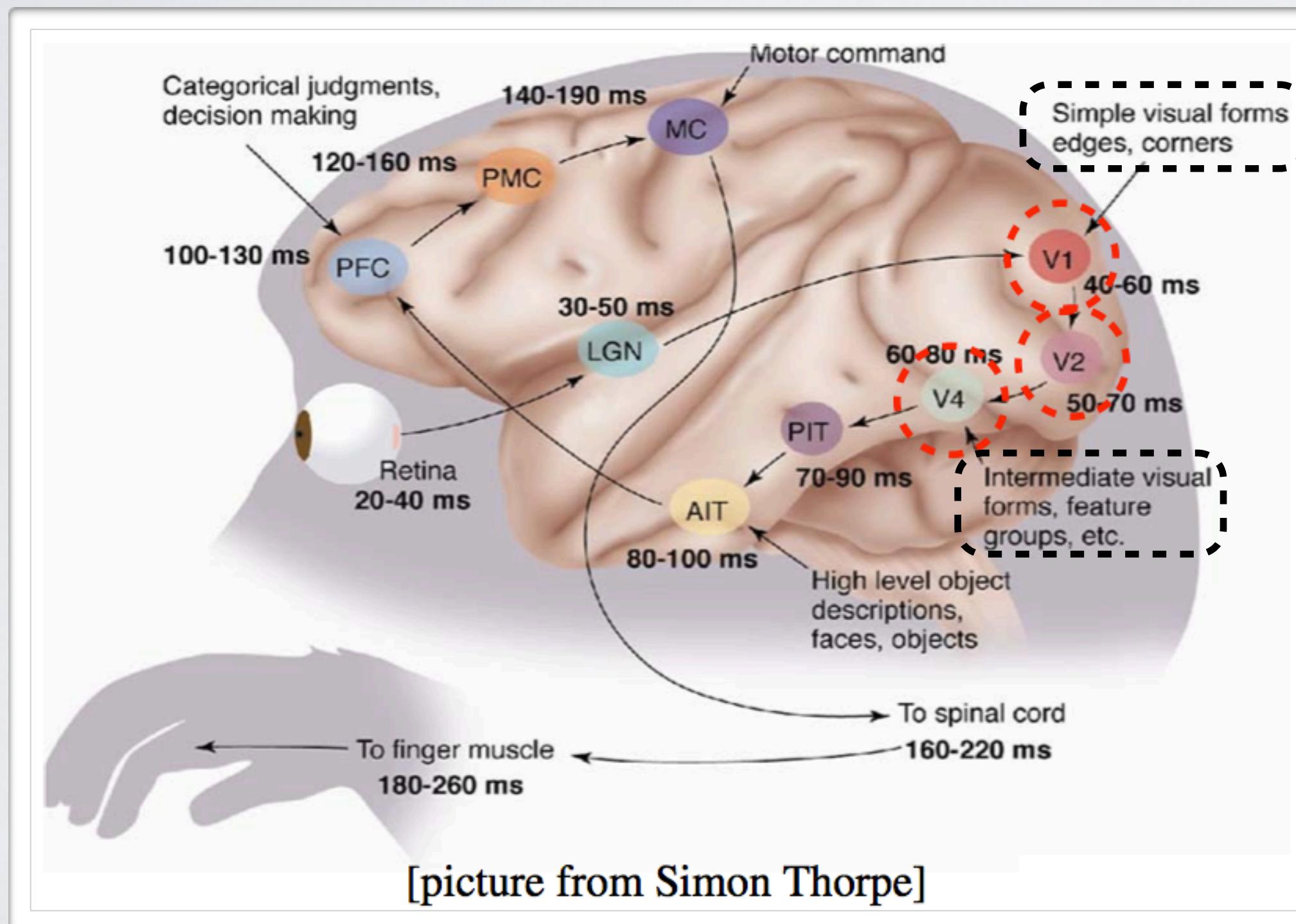
# DEEP LEARNING

**Topics:** inspiration from visual cortex



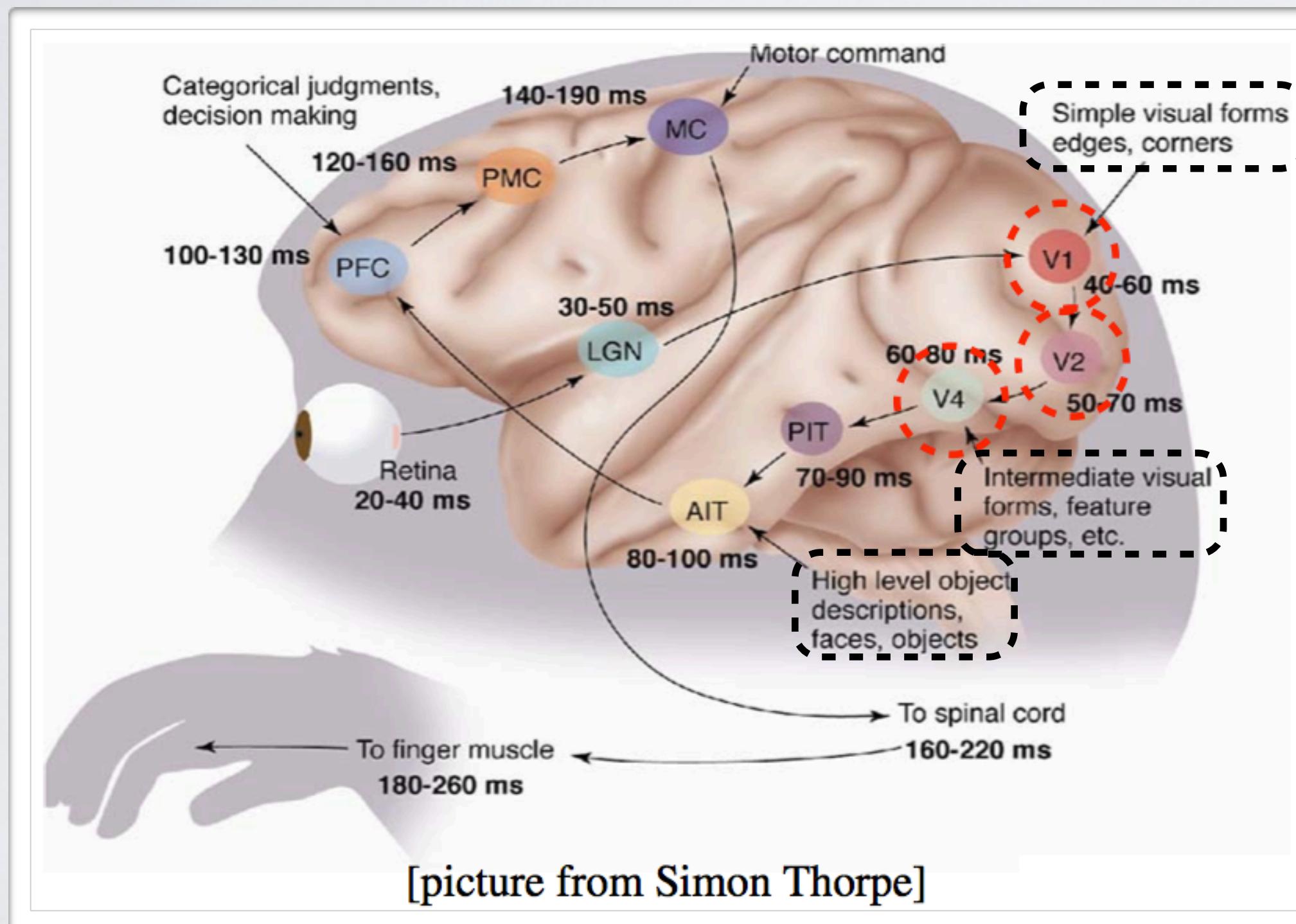
# DEEP LEARNING

**Topics:** inspiration from visual cortex



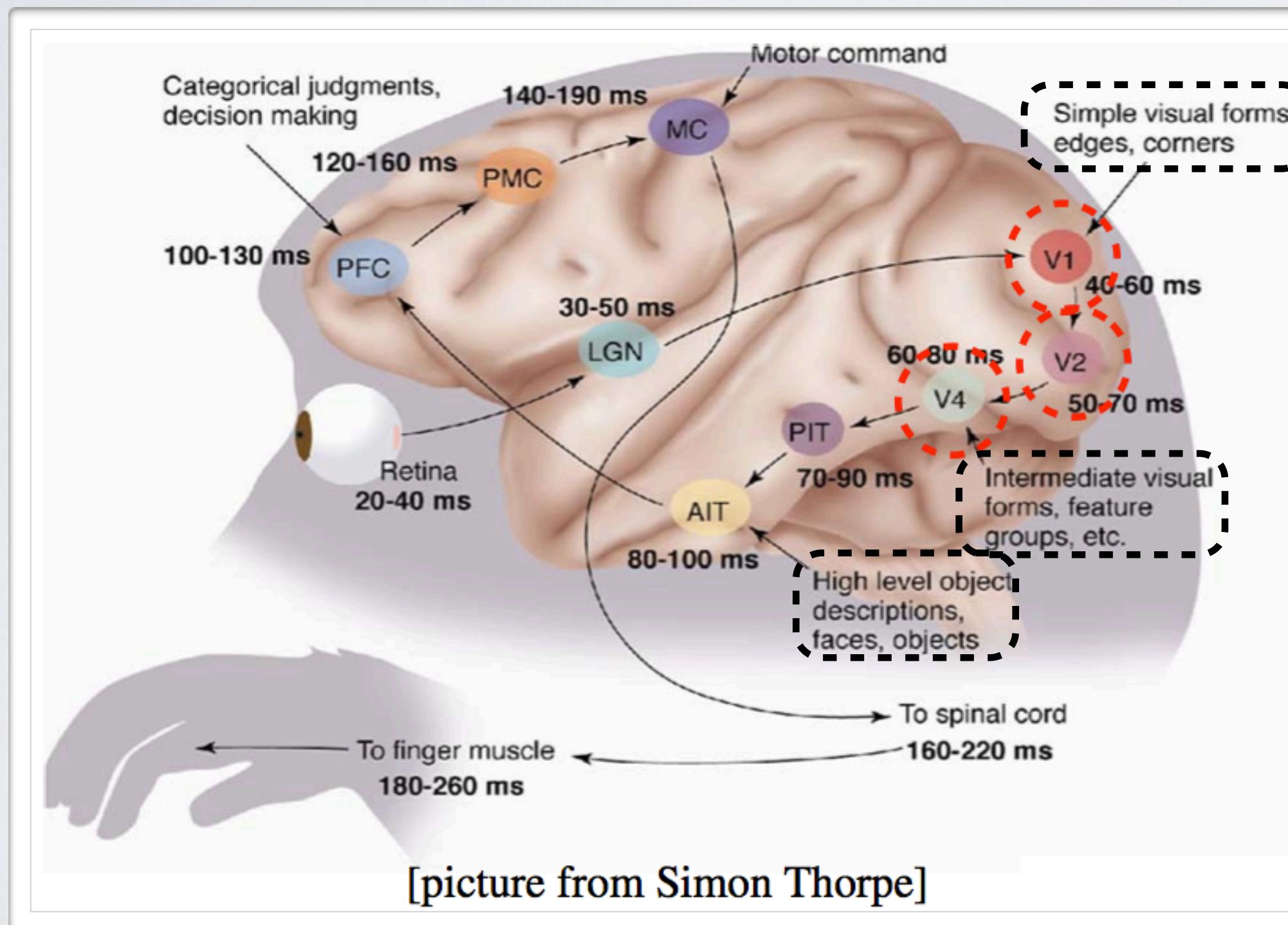
# DEEP LEARNING

**Topics:** inspiration from visual cortex



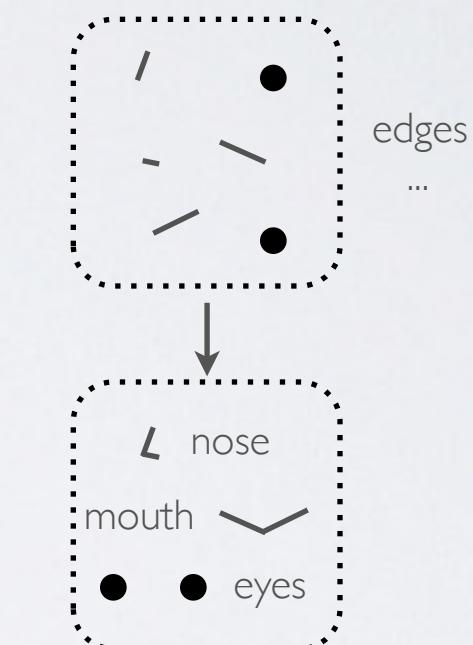
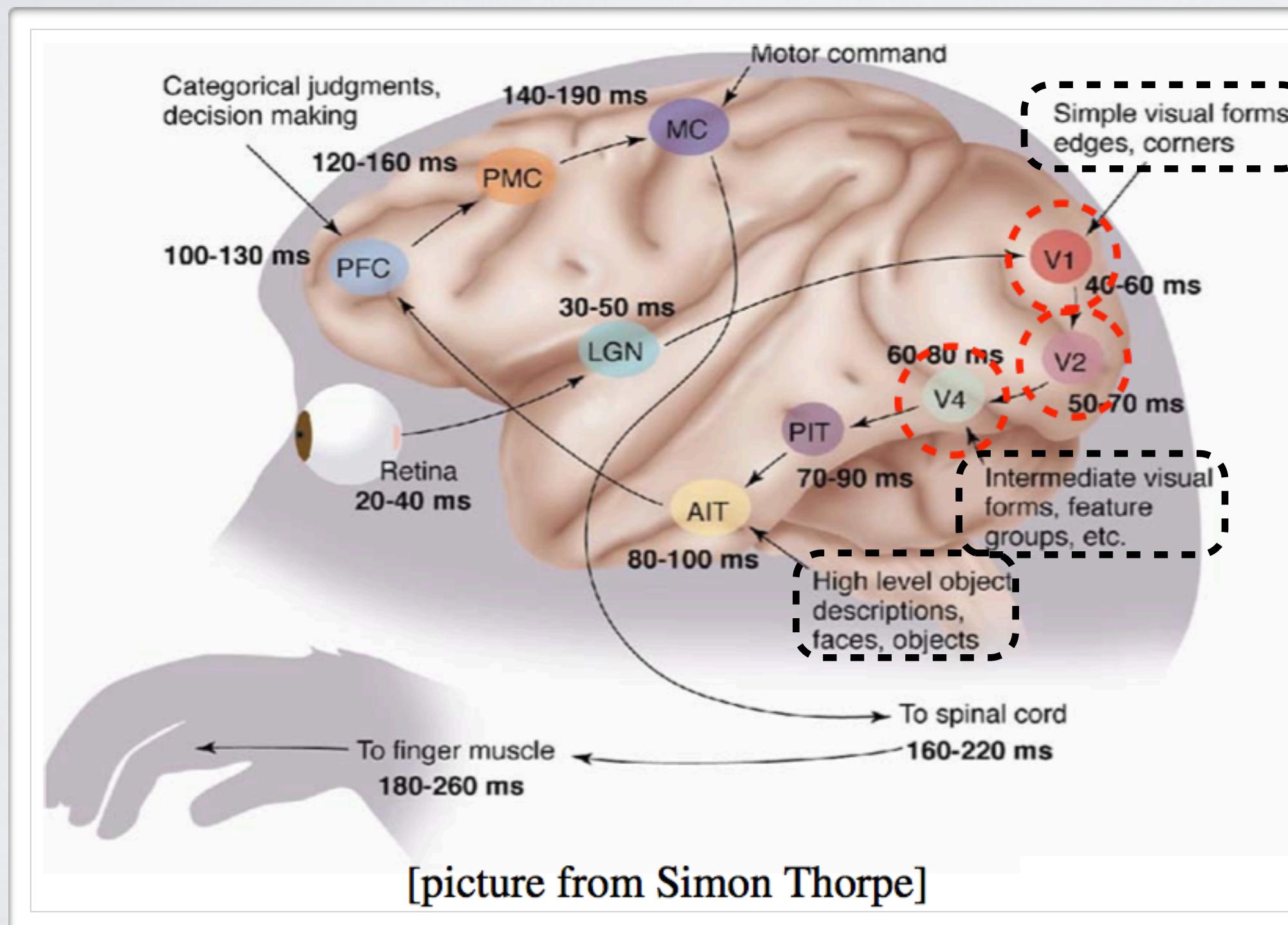
# DEEP LEARNING

**Topics:** inspiration from visual cortex



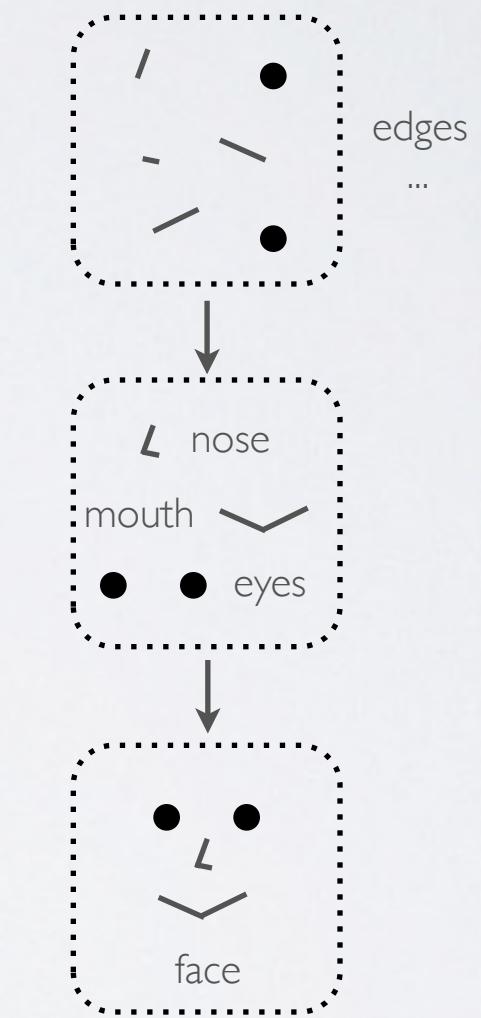
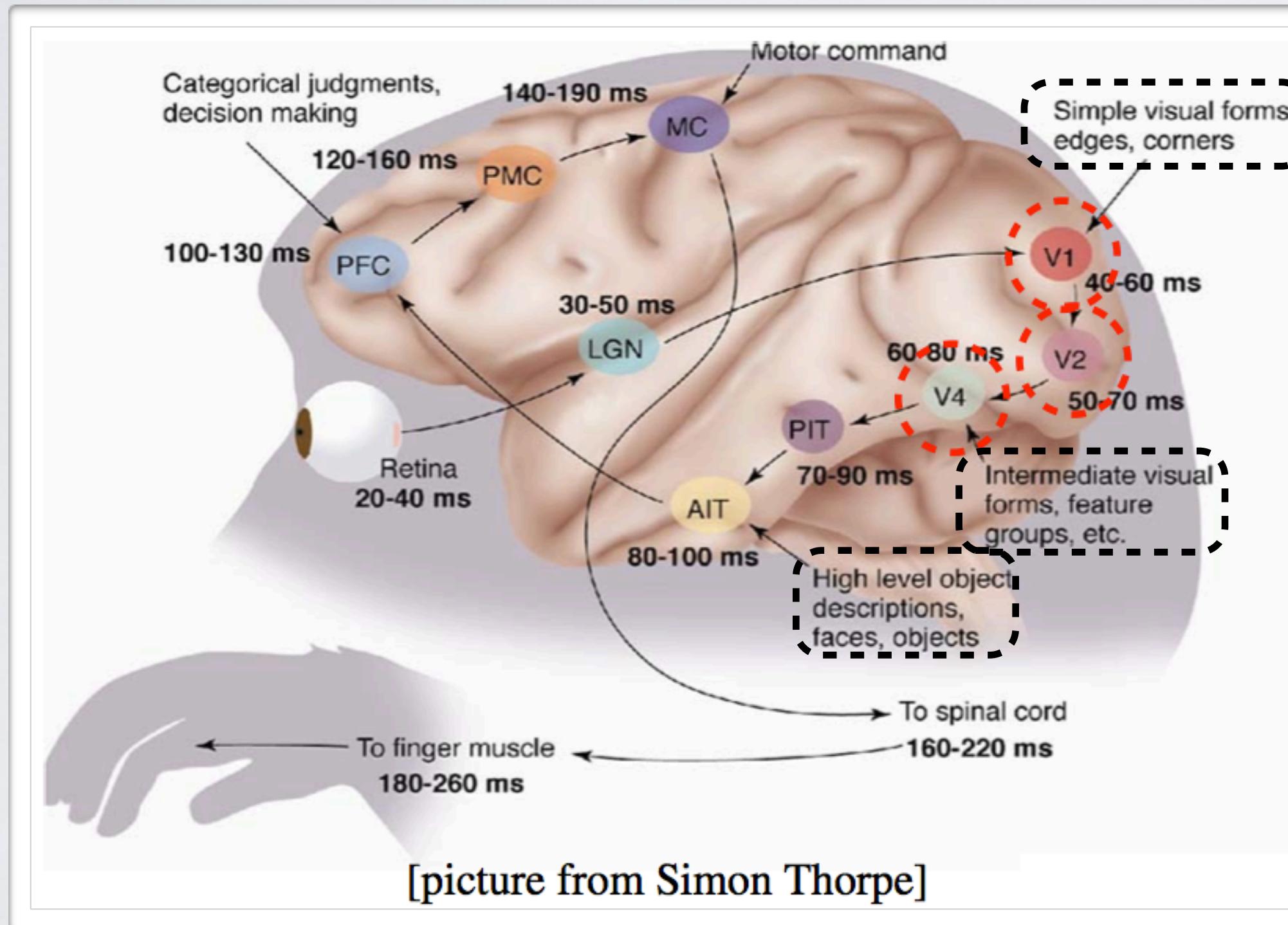
# DEEP LEARNING

**Topics:** inspiration from visual cortex



# 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

The screenshot shows the Microsoft Research website. The header features the Microsoft Research logo and a search bar. The main navigation menu includes Home, Our Research, Connections, Careers, Hub, About Us, News, Media Resources, Events, and Community. Below the menu, a breadcrumb trail indicates the article is under News > Speech Recognition Leaps Forward. The main title of the article is "Speech Recognition Leaps Forward". It is written by Janie Chang and published on August 29, 2011, at 12:01 AM PT. The article text discusses research at Interspeech 2011, mentioning Dong Yu and Frank Seide's work on artificial neural networks for speech recognition. A section titled "The Holy Grail of Speech Recognition" is partially visible at the bottom.

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

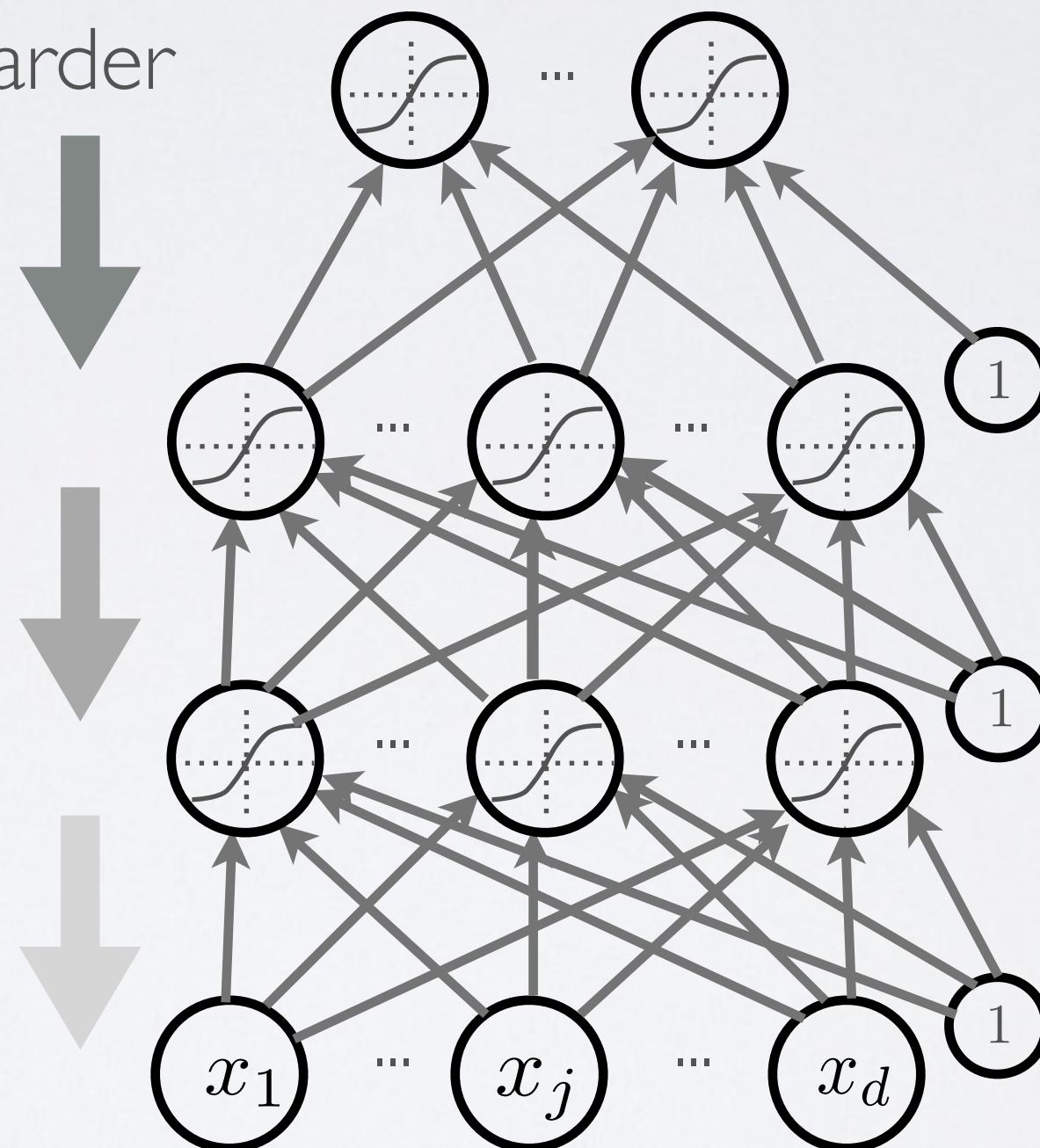
**Topics:** success story: computer vision



# DEEP LEARNING

**Topics:** why training is hard

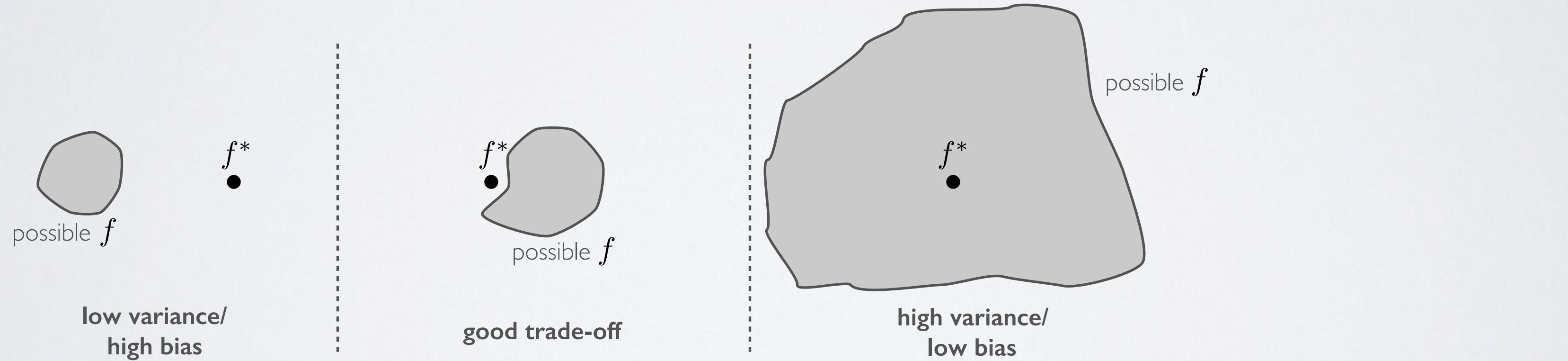
- First hypothesis: optimization is harder (underfitting)
  - ▶ vanishing gradient problem
  - ▶ saturated units block gradient propagation
- This is a well known problem in recurrent neural networks



# DEEP LEARNING

**Topics:** why training is hard

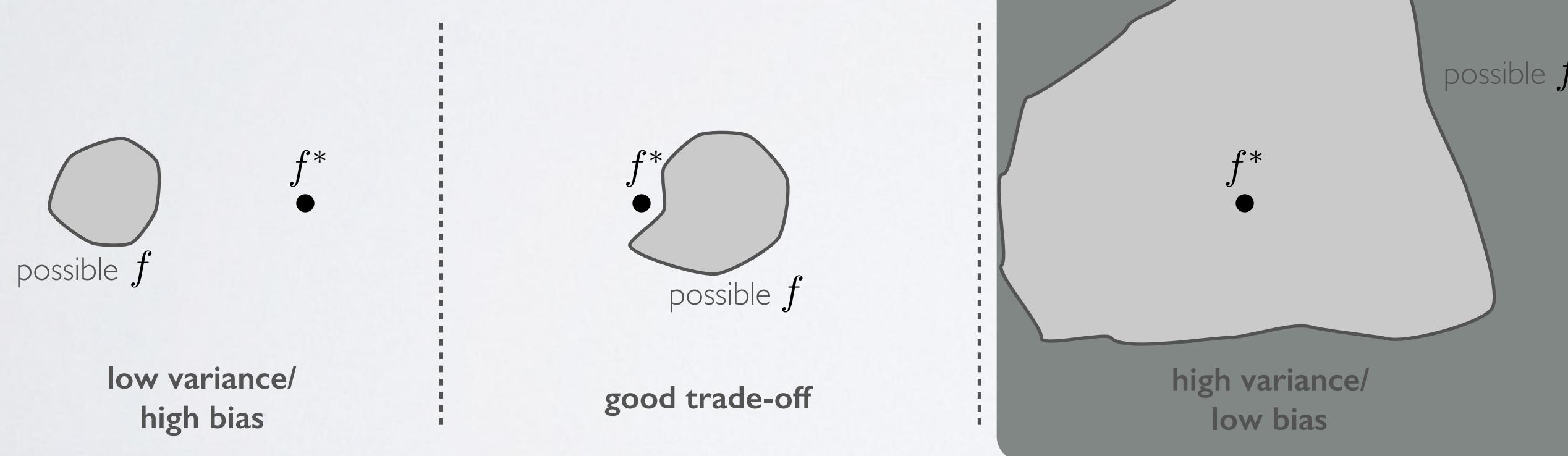
- Second hypothesis: overfitting
  - we are exploring a space of complex functions
  - deep nets usually have lots of parameters
- Might be in a high variance / low bias situation



# DEEP LEARNING

**Topics:** why training is hard

- 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 pre-training
  - ▶ stochastic «dropout» training

# DEEP LEARNING

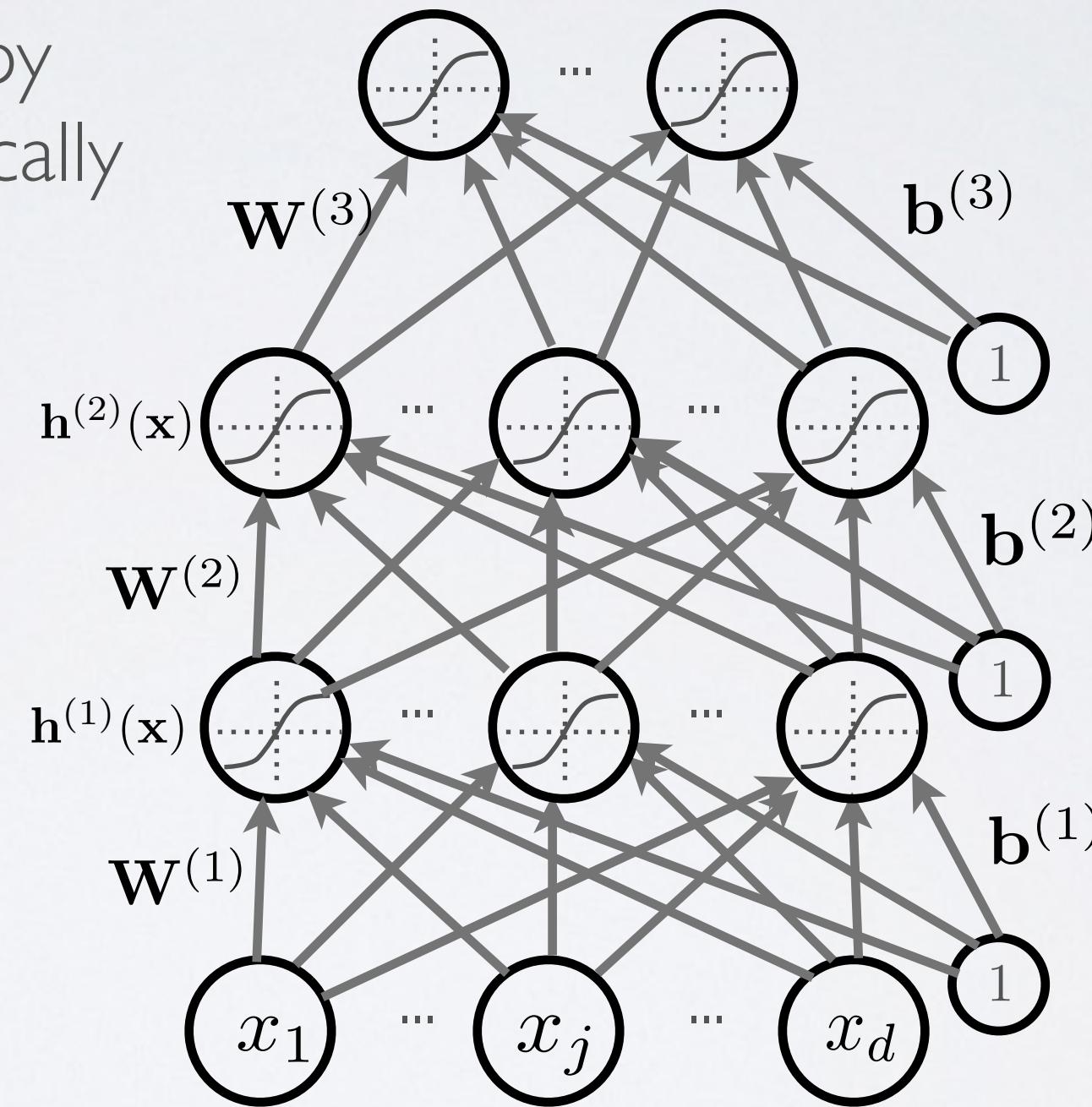
**Topics:** why training is hard

- 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 pre-training
  - ▶ **stochastic «dropout» training**

# DROPOUT

## Topics: dropout

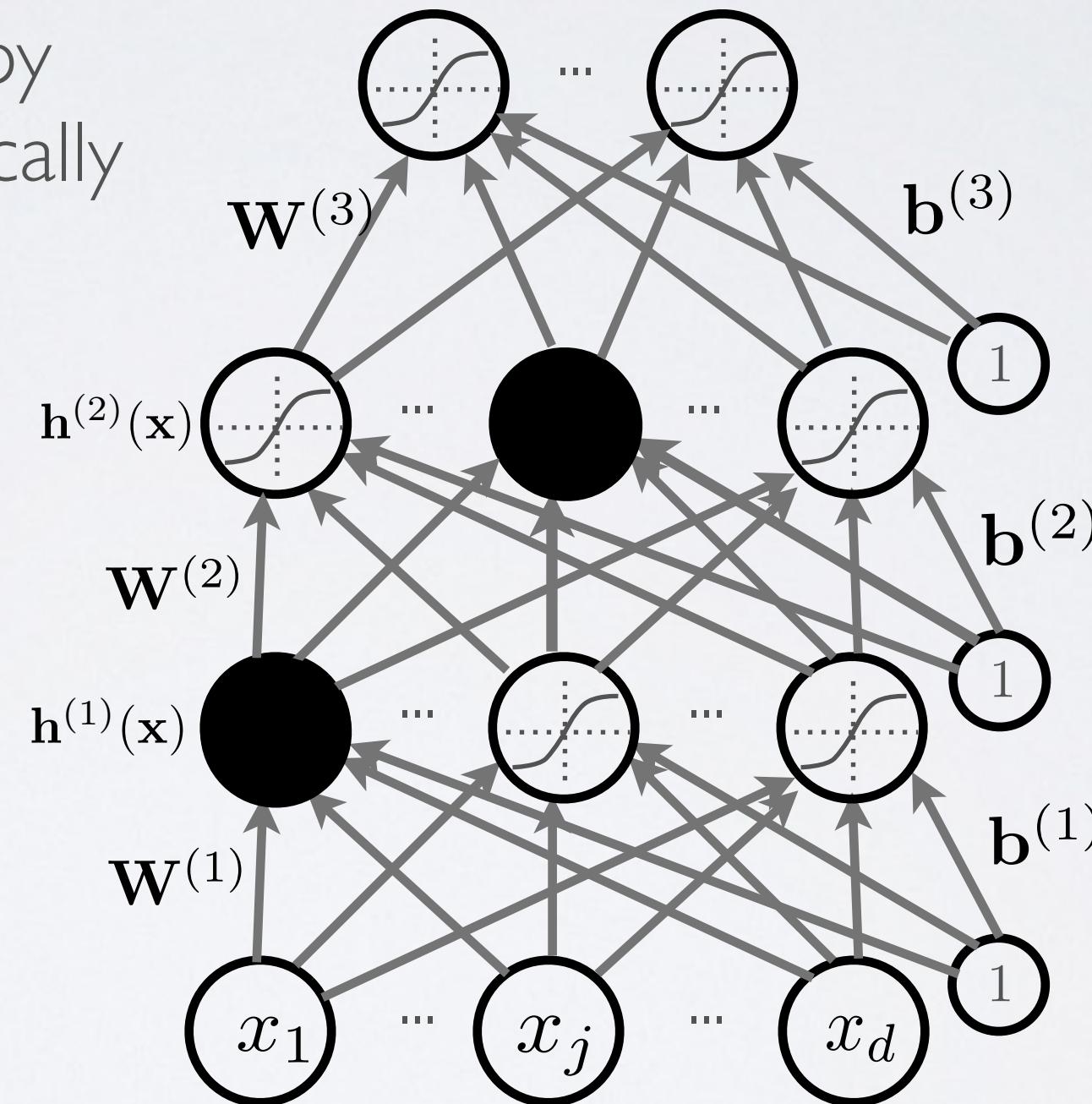
- Idea: «cripple» neural network by removing hidden units stochastically
  - ▶ each hidden unit is set to 0 with probability 0.5
  - ▶ hidden units cannot co-adapt to other units
  - ▶ hidden units must be more generally useful
- Could use a different dropout probability, but 0.5 usually works well



# DROPOUT

## Topics: dropout

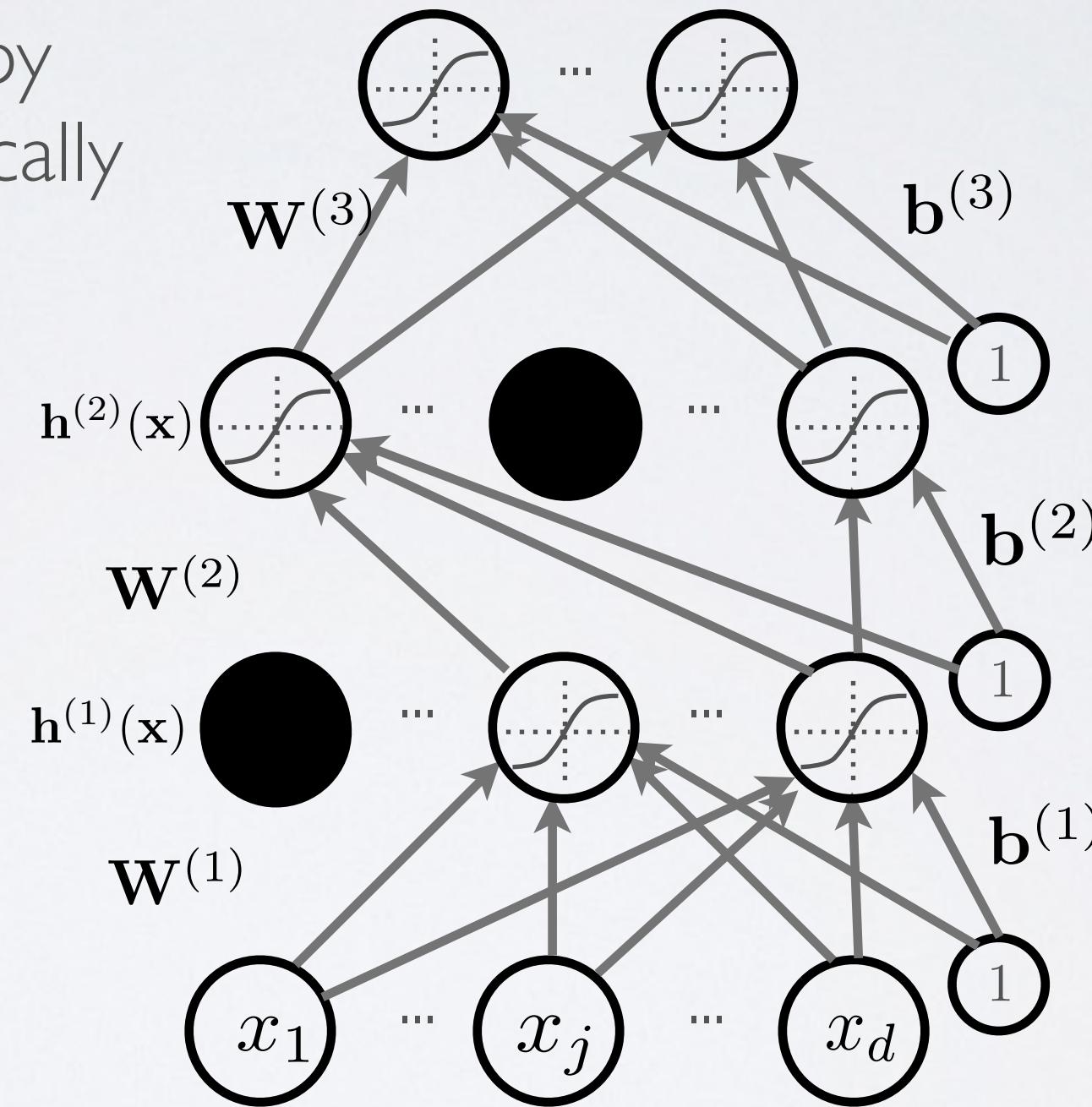
- Idea: «cripple» neural network by removing hidden units stochastically
  - ▶ each hidden unit is set to 0 with probability 0.5
  - ▶ hidden units cannot co-adapt to other units
  - ▶ hidden units must be more generally useful
- Could use a different dropout probability, but 0.5 usually works well



# DROPOUT

## Topics: dropout

- 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  $k > 0$  ( $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$ )

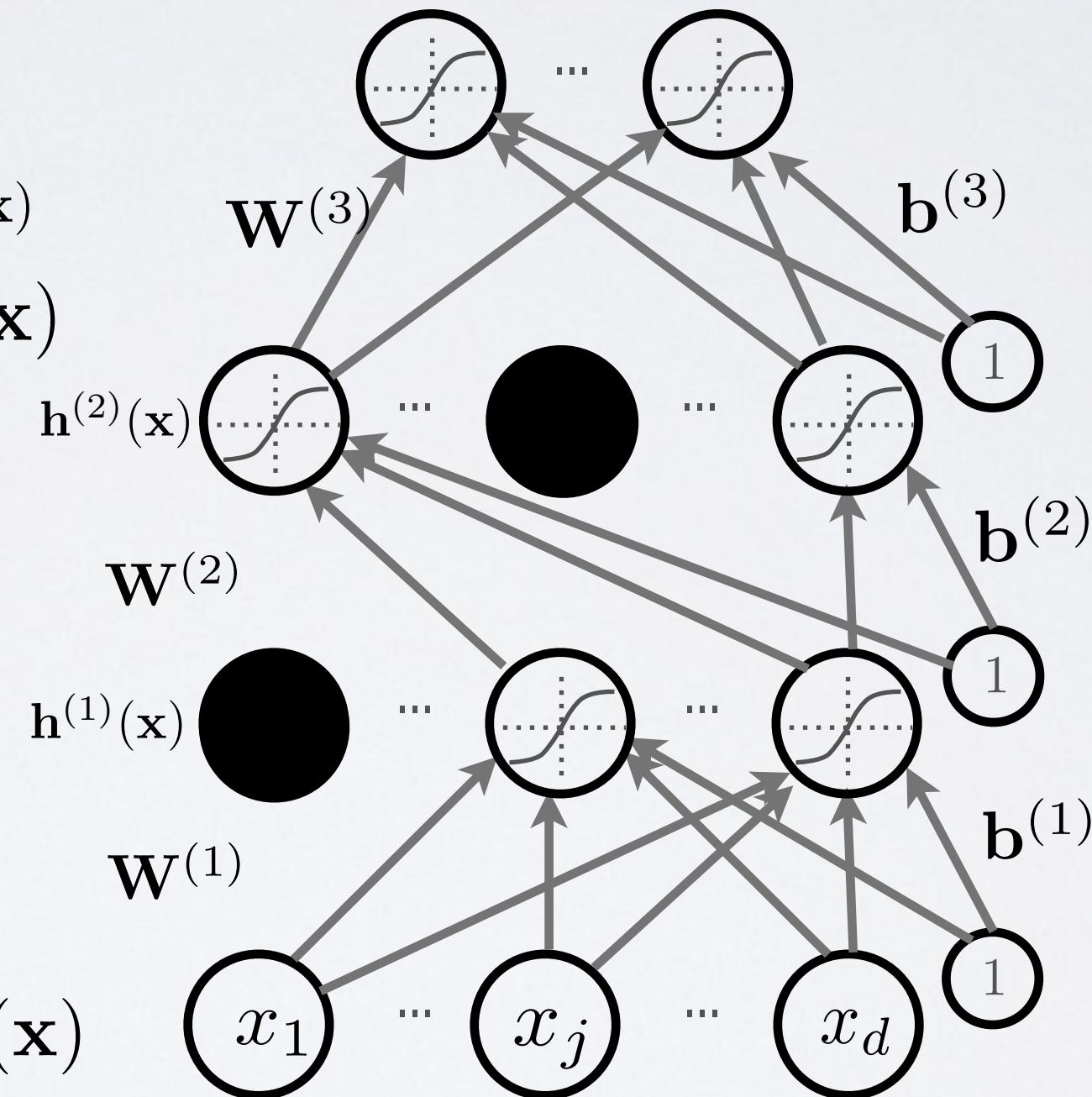
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

► hidden layer activation ( $k$  from 1 to  $L$ ):

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

► 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

- Use random binary masks  $\mathbf{m}^{(k)}$

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

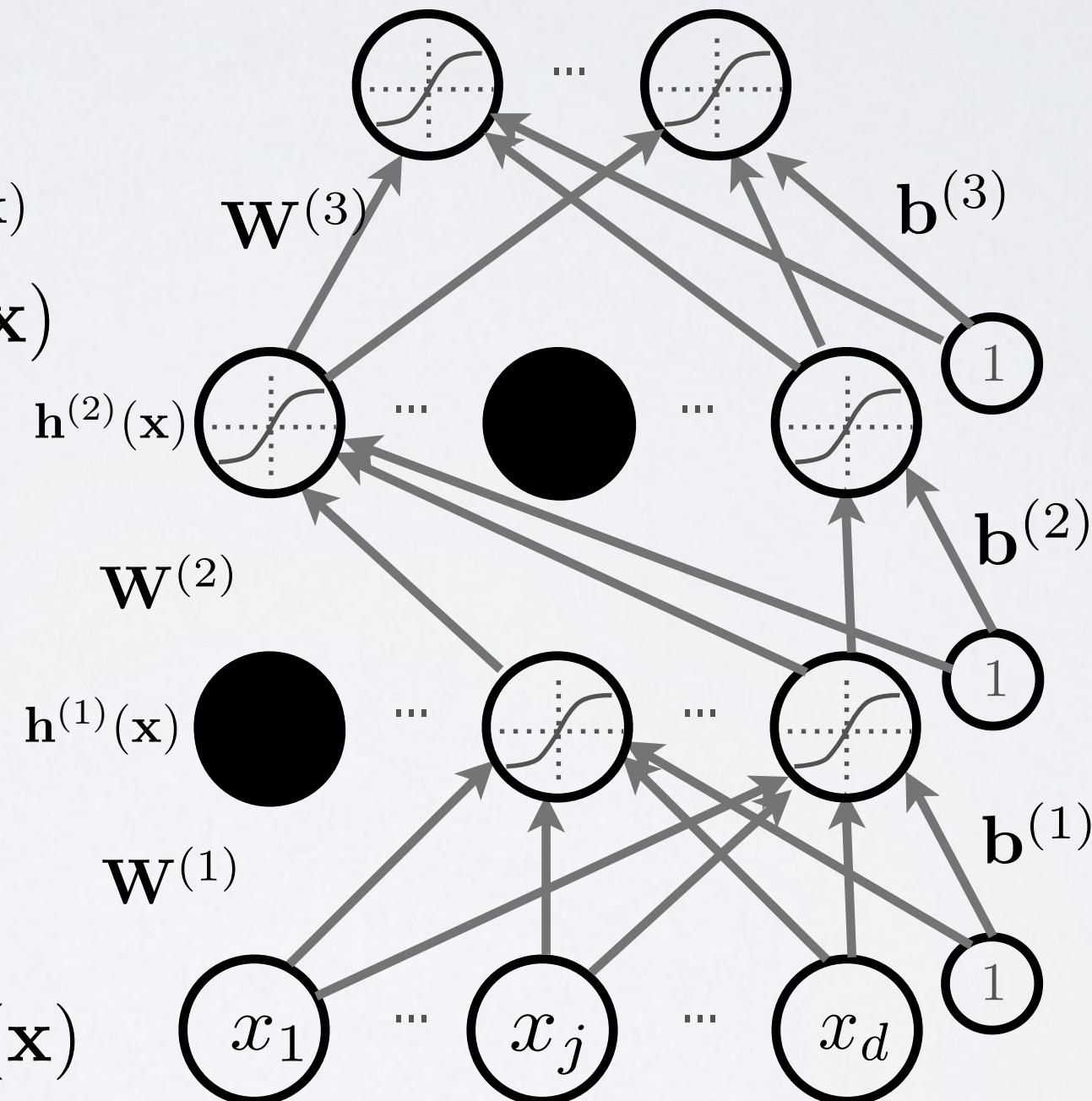
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

► 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

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

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

- compute gradient of hidden layer below

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

- compute gradient of hidden layer below (before activation)

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

# DROPOUT

## Topics: dropout backpropagation

- This assumes a forward propagation has been made before

- compute output gradient (before activation)

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

- for  $k$  from  $L+1$  to 1

- compute gradients of hidden layer parameter

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

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

- compute gradient of hidden layer below

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

- compute gradient of hidden layer below (before activation)

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

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
- Beats regular backpropagation on many datasets, but is slower ( $\sim 2x$ )
  - ▶ Improving neural networks by preventing co-adaptation of feature detectors.  
Hinton, Srivastava, Krizhevsky, Sutskever and Salakhutdinov, 2012.

# DEEP LEARNING

**Topics:** why training is hard

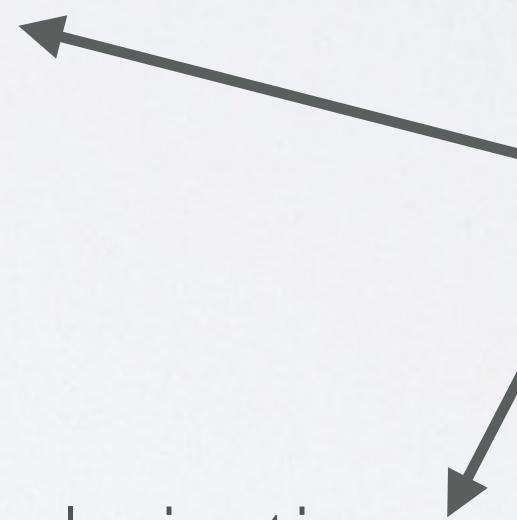
- Depending on the problem, one or the other situation will tend to dominate
- If first hypothesis (underfitting): better optimize
  - ▶ use better optimization methods
  - ▶ use GPUs
- If second hypothesis (overfitting): use better regularization
  - ▶ unsupervised pre-training
  - ▶ stochastic «dropout» training

# 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 pre-training
  - ▶ stochastic «dropout» training

**Batch normalization**



# 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:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

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

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

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

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

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

# BATCH NORMALIZATION

**Topics:** batch normalization

- **Batch normalization**

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

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

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

// mini-batch mean

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$$

// mini-batch variance

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

// normalize

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$$

// scale and shift

Learned linear transformation  
to adapt to non-linear activation  
function  
( $\gamma$  and  $\beta$  are **trained**)

# 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 pre-training**
  - ▶ stochastic «dropout» training

# UNSUPERVISED PRE-TRAINING

**Topics:** unsupervised pre-training

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



character image



random image

- ▶ encourage hidden layers to encode that structure

# UNSUPERVISED PRE-TRAINING

**Topics:** unsupervised pre-training

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



character image

Why is one  
a character  
and the other  
is not ?



random image

- ▶ 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

Why is one  
a character  
and the other  
is not ?



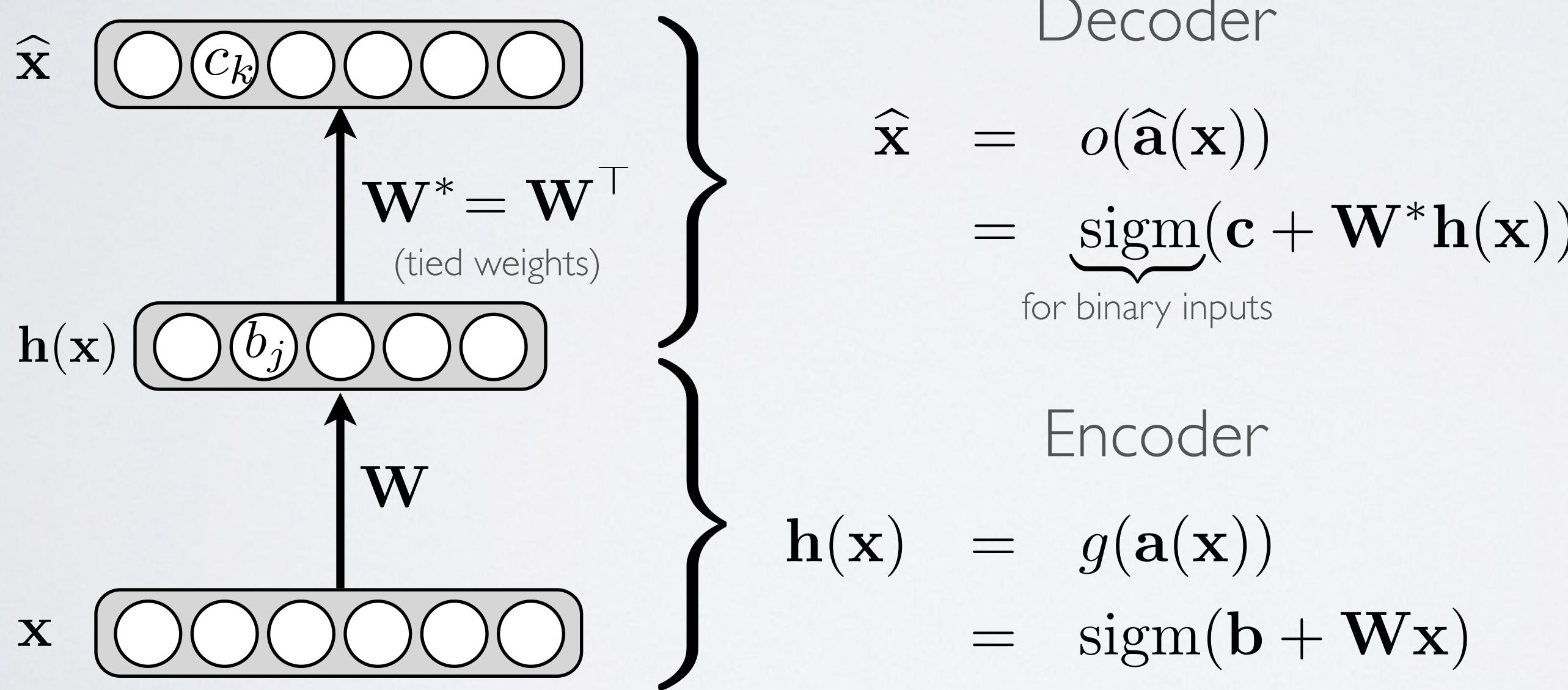
random image

- ▶ 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

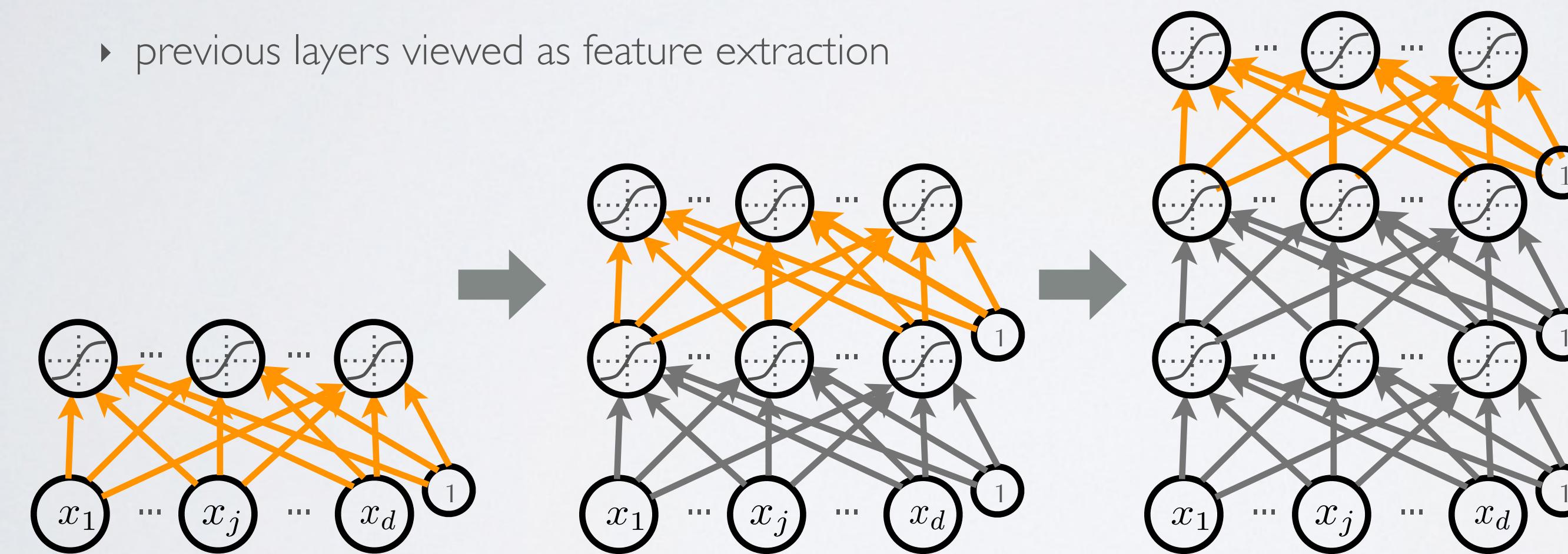


# UNSUPERVISED PRE-TRAINING

**Topics:** unsupervised pre-training

- We will use a greedy, layer-wise procedure

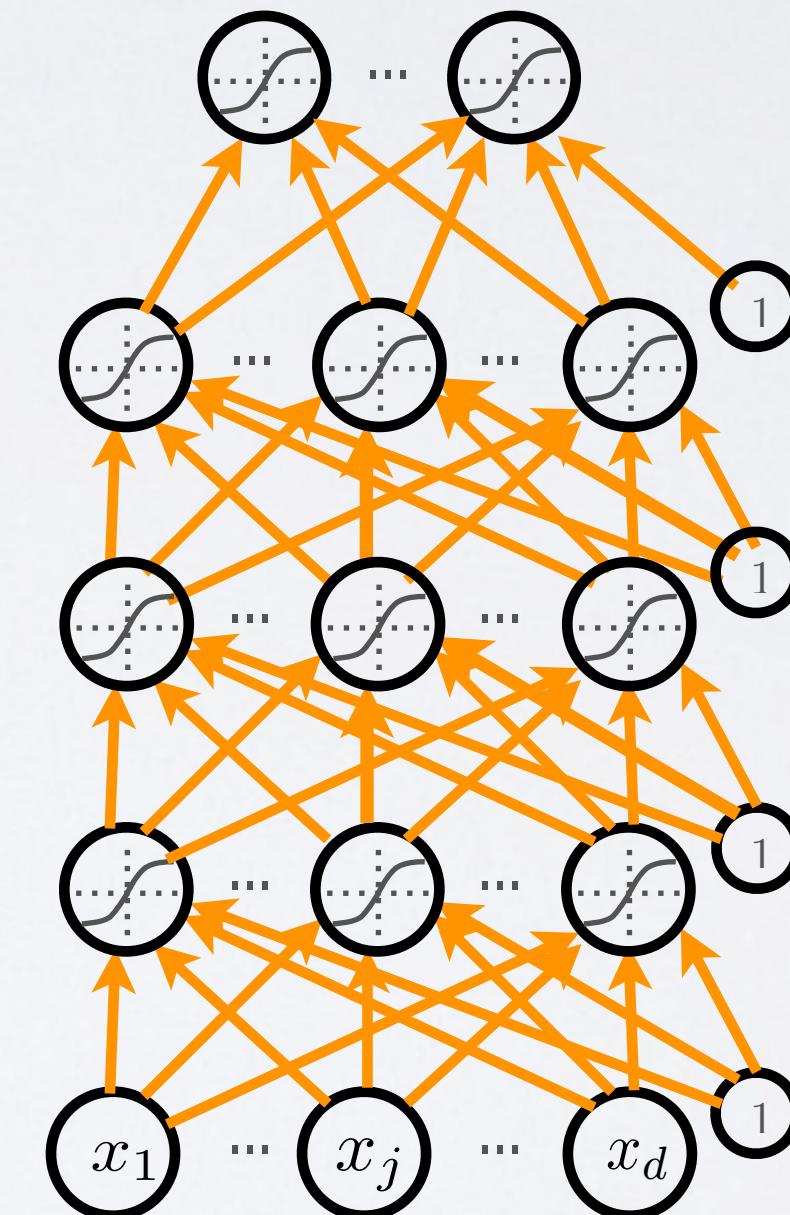
- ▶ train one layer at a time, from first to last, with unsupervised criterion
- ▶ fix the parameters of previous hidden layers
- ▶ previous layers viewed as feature extraction



# 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



# NEURAL NETWORK ONLINE COURSE

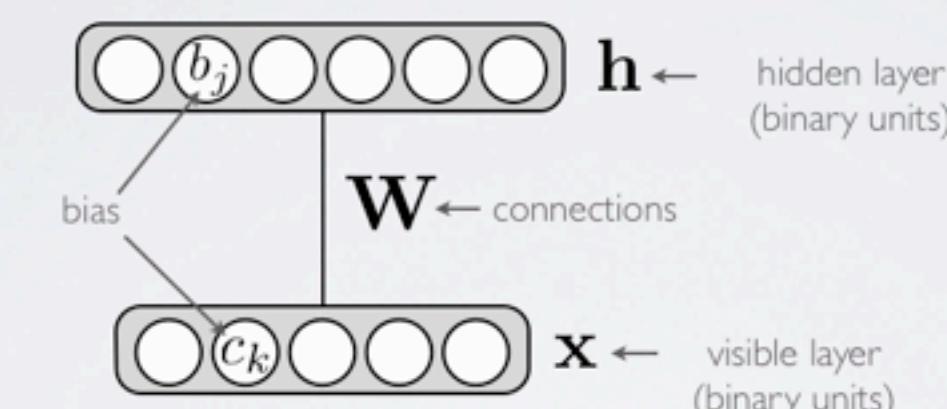
**Topics:** online videos

- ▶ for a more detailed description of neural networks...
- ▶ ... and much more!

[http://info.usherbrooke.ca/hlarochelle/neural\\_networks](http://info.usherbrooke.ca/hlarochelle/neural_networks)

RESTRICTED BOLTZMANN MACHINE

**Topics:** RBM, visible layer, hidden layer, energy function



Energy function: 
$$\begin{aligned} E(\mathbf{x}, \mathbf{h}) &= -\mathbf{h}^\top \mathbf{W} \mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h} \\ &= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j \end{aligned}$$

Distribution:  $p(\mathbf{x}, \mathbf{h}) = \exp(-E(\mathbf{x}, \mathbf{h}))/Z$

partition function  
(intractable)

Click with the mouse or tablet to draw with pen 2



# NEURAL NETWORK ONLINE COURSE

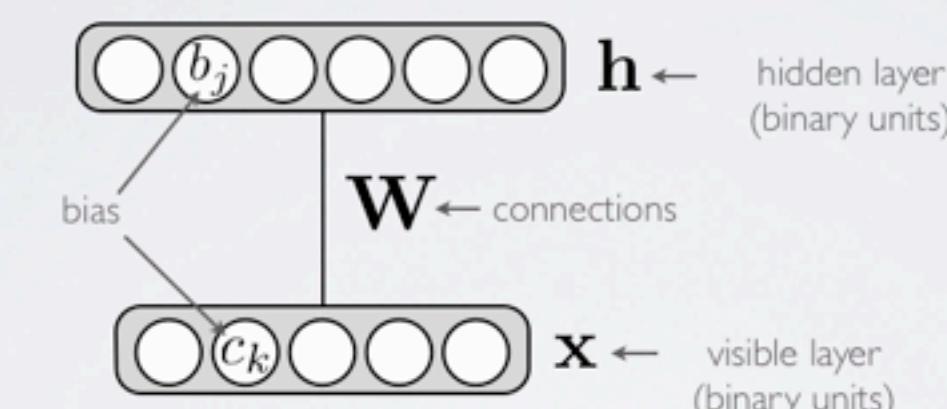
**Topics:** online videos

- ▶ for a more detailed description of neural networks...
- ▶ ... and much more!

[http://info.usherbrooke.ca/hlarochelle/neural\\_networks](http://info.usherbrooke.ca/hlarochelle/neural_networks)

RESTRICTED BOLTZMANN MACHINE

**Topics:** RBM, visible layer, hidden layer, energy function



Energy function: 
$$\begin{aligned} E(\mathbf{x}, \mathbf{h}) &= -\mathbf{h}^\top \mathbf{W} \mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h} \\ &= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j \end{aligned}$$

Distribution:  $p(\mathbf{x}, \mathbf{h}) = \exp(-E(\mathbf{x}, \mathbf{h}))/Z$

partition function  
(intractable)

**Click with the mouse or tablet to draw with pen 2**



MERCI!!