

# Training Deep Neural Networks

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

# 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](http://info.usherbrooke.ca/hlarochelle/neural_networks)

Click with the mouse or tablet to draw with pen 2

## 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 bias unit  $b_j$  is associated with the hidden layer, and a bias unit  $c_k$  is associated with the visible layer. The layers are connected by a weight matrix  $\mathbf{W}$ . A cursor icon is visible on the left side of the diagram area.

hidden layer  
(binary units)

$\mathbf{h} \leftarrow$

bias

$\mathbf{W} \leftarrow$  connections

visible layer  
(binary units)

$\mathbf{x} \leftarrow$

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

**Topics:** multilayer neural network

- Could have  $L$  hidden layers:

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

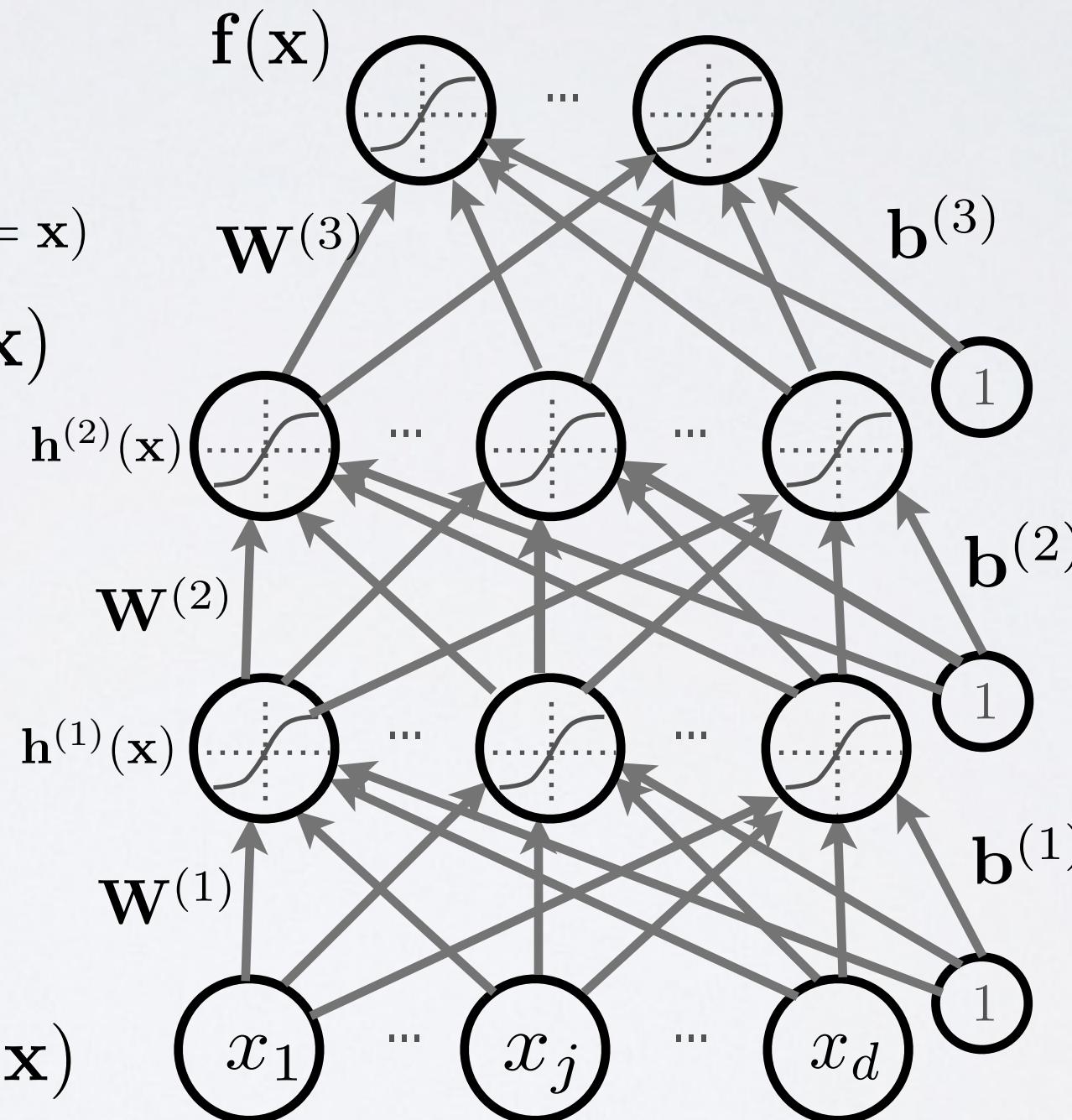
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- 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  $\boldsymbol{\theta}$  ( $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$ )
- for N iterations

$$\left. \begin{array}{l} \text{- for each training example } (\mathbf{x}^{(t)}, y^{(t)}) \\ \quad \checkmark \Delta = -\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) \\ \quad \checkmark \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \Delta \end{array} \right\} \begin{array}{l} \text{training epoch} \\ = \\ \text{iteration over \textbf{all} examples} \end{array}$$

- 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  $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$ )

# 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)

A diagram showing a downward-pointing arrow from the word "natural log (ln)" to the term  $\log f(\mathbf{x})_c$  in the equation above.

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

# BACKPROPAGATION

**Topics:** backpropagation algorithm

- This assumes a forward propagation has been made before

- ▶ compute output gradient (before activation)

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

- ▶ 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

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

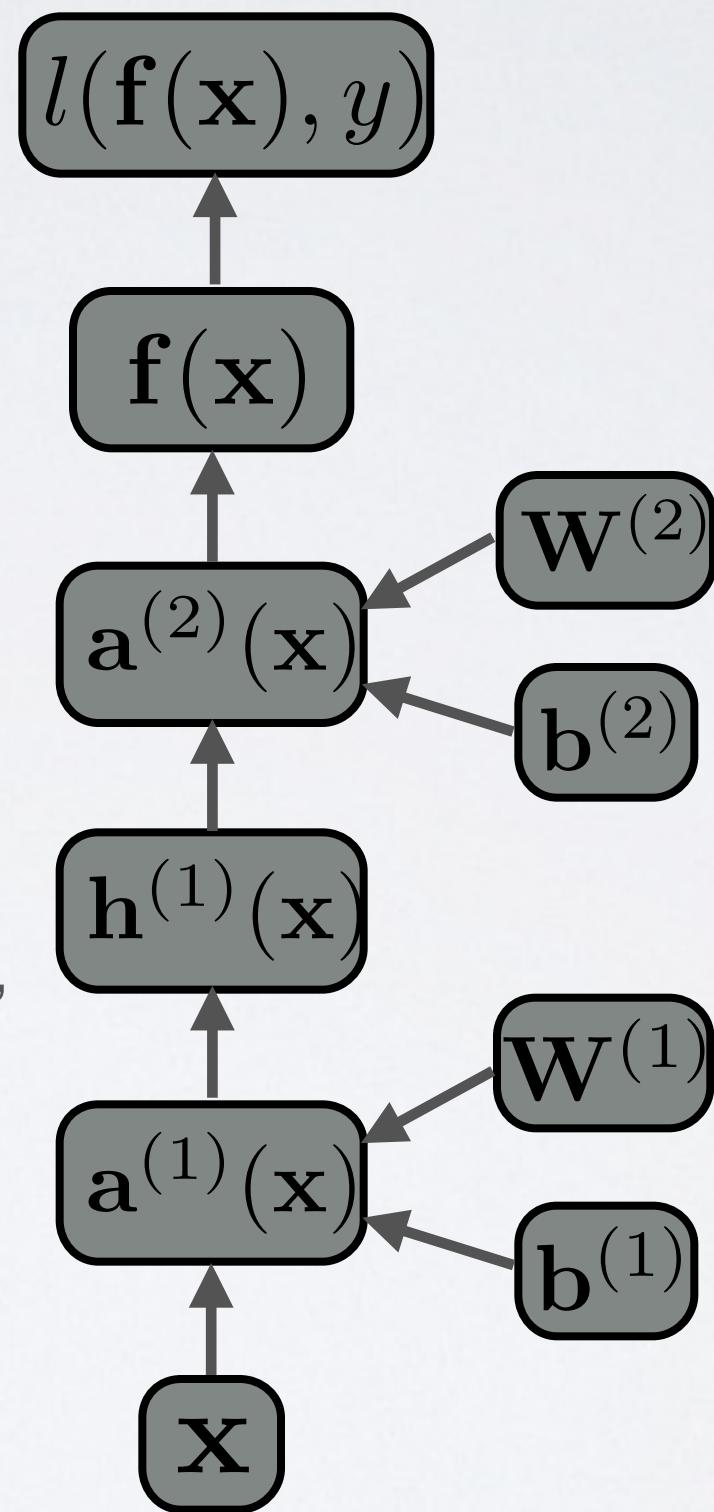
- compute gradient of hidden layer below (before activation)

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

# FLOW GRAPH

## Topics: flow graph

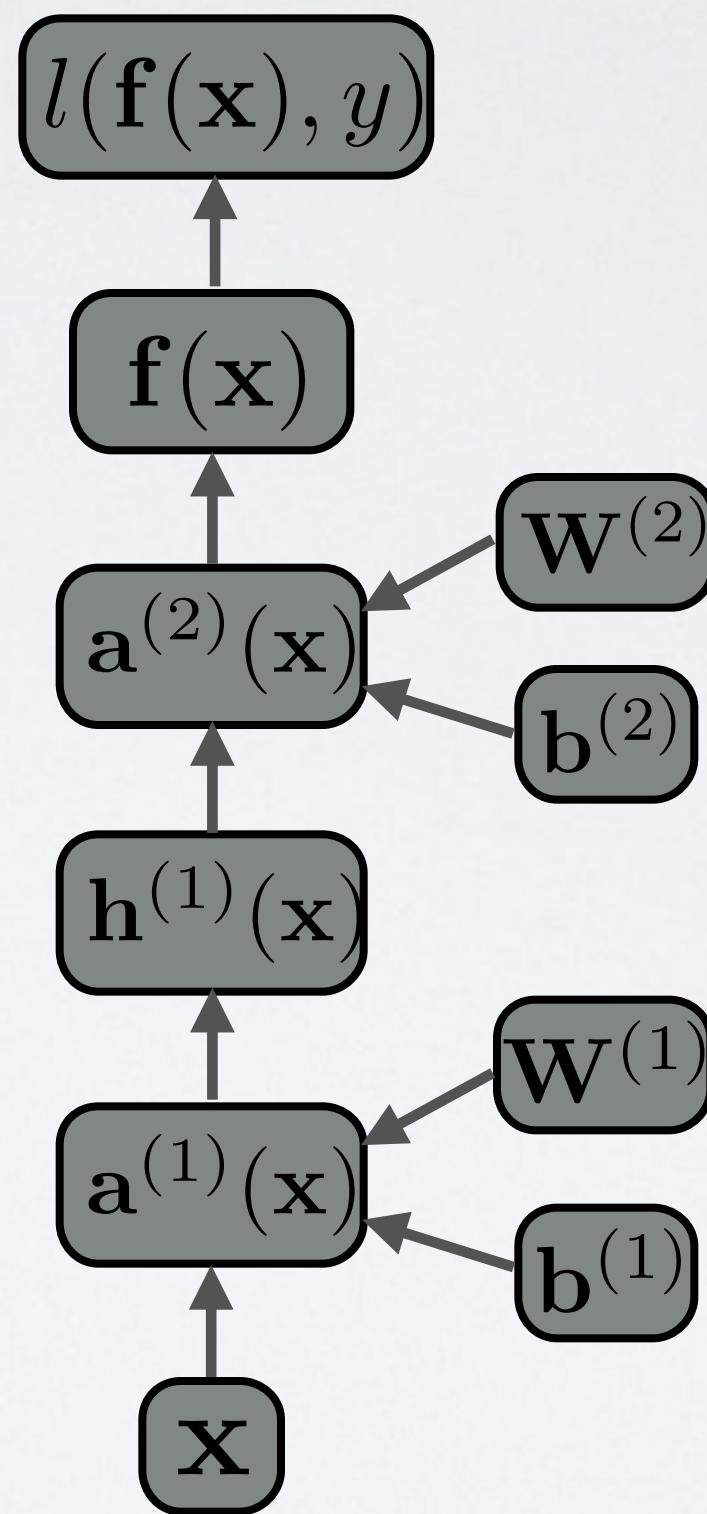
- Forward propagation can be represented as an acyclic flow graph
- It's a nice way of implementing forward propagation in a modular way
  - ▶ each box could be an object with an fprop method, that computes the value of the box given its children
  - ▶ calling the fprop method of each box in the right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



# 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

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

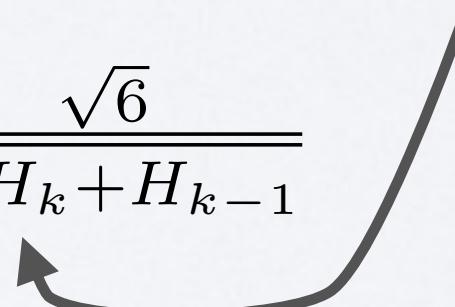
- Gradient:  $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$ 
  - ▶ where  $\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$
- Also only applied on weights
- Unlike L2, L1 will push certain weights to be exactly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

# 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)

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



# 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

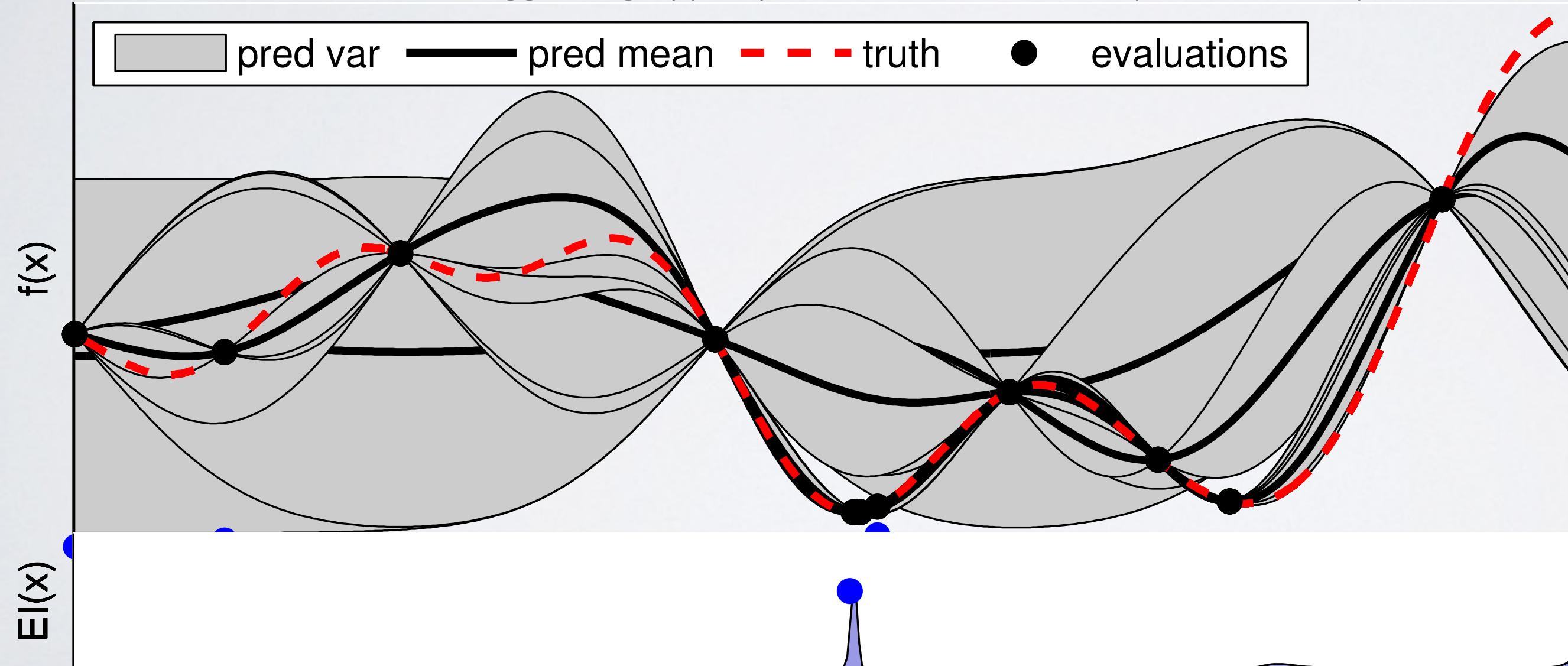
**Topics:** grid search, random search

- 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 a configuration, and repeat as many times as wanted
- Use a validation set performance to select the best configuration
- You can go back and refine the grid/distributions if needed

# MODEL SELECTION

**Topics:** bayesian optimization, sequential model-based optimization

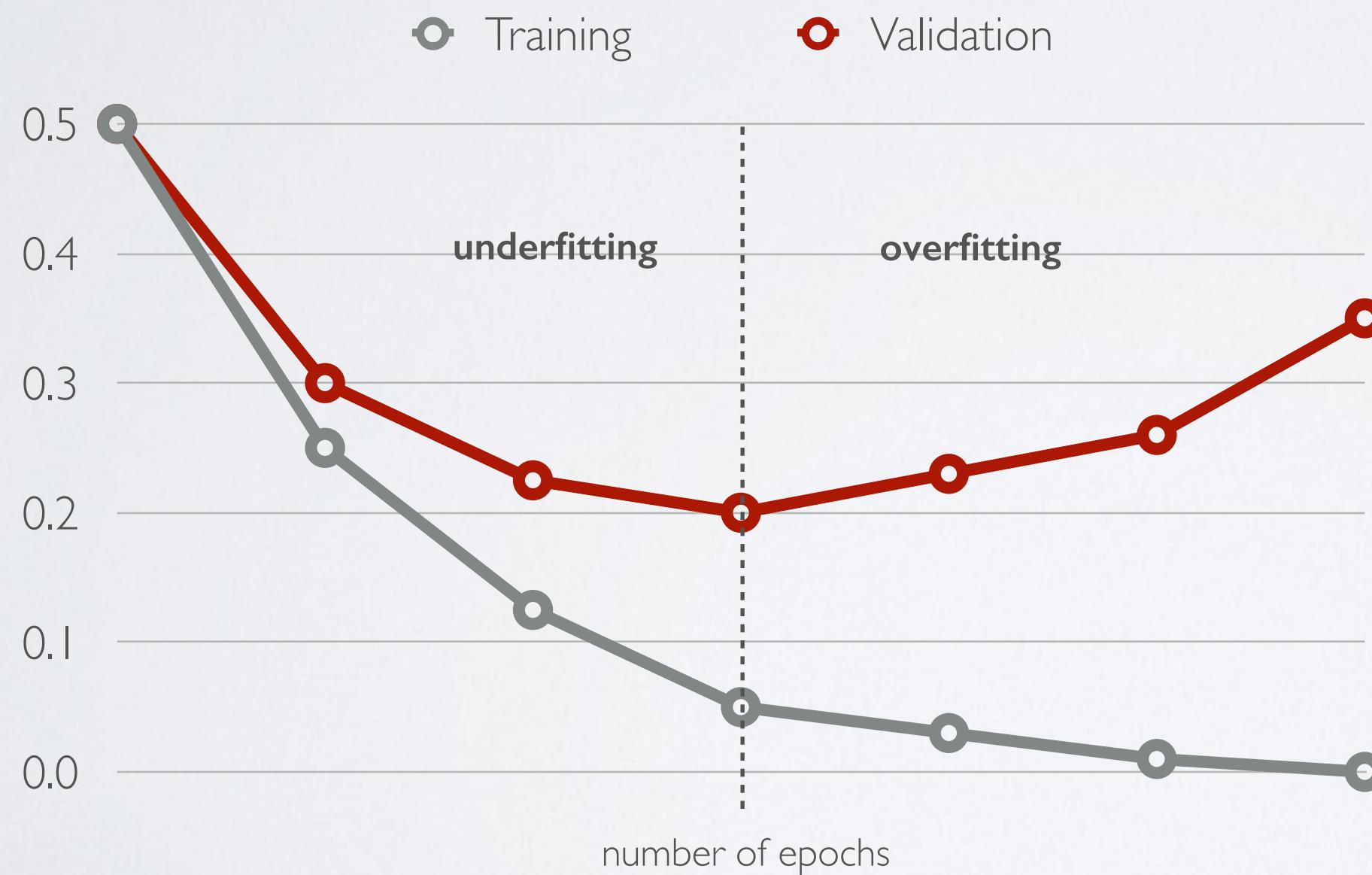
- Use machine learning to predict performance on validation set
  - ▶ model must provide a predictive mean and variance
  - ▶ alternate between suggesting hyper-parameters and train/predict valid performance



# 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:

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

- ▶ can get through plateaus more quickly, by “gaining momentum”

# 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

# Neural networks

Deep learning - difficulty of training

# NEURAL NETWORK

REMINDER

**Topics:** multilayer neural network

- Could have  $L$  hidden layers:

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

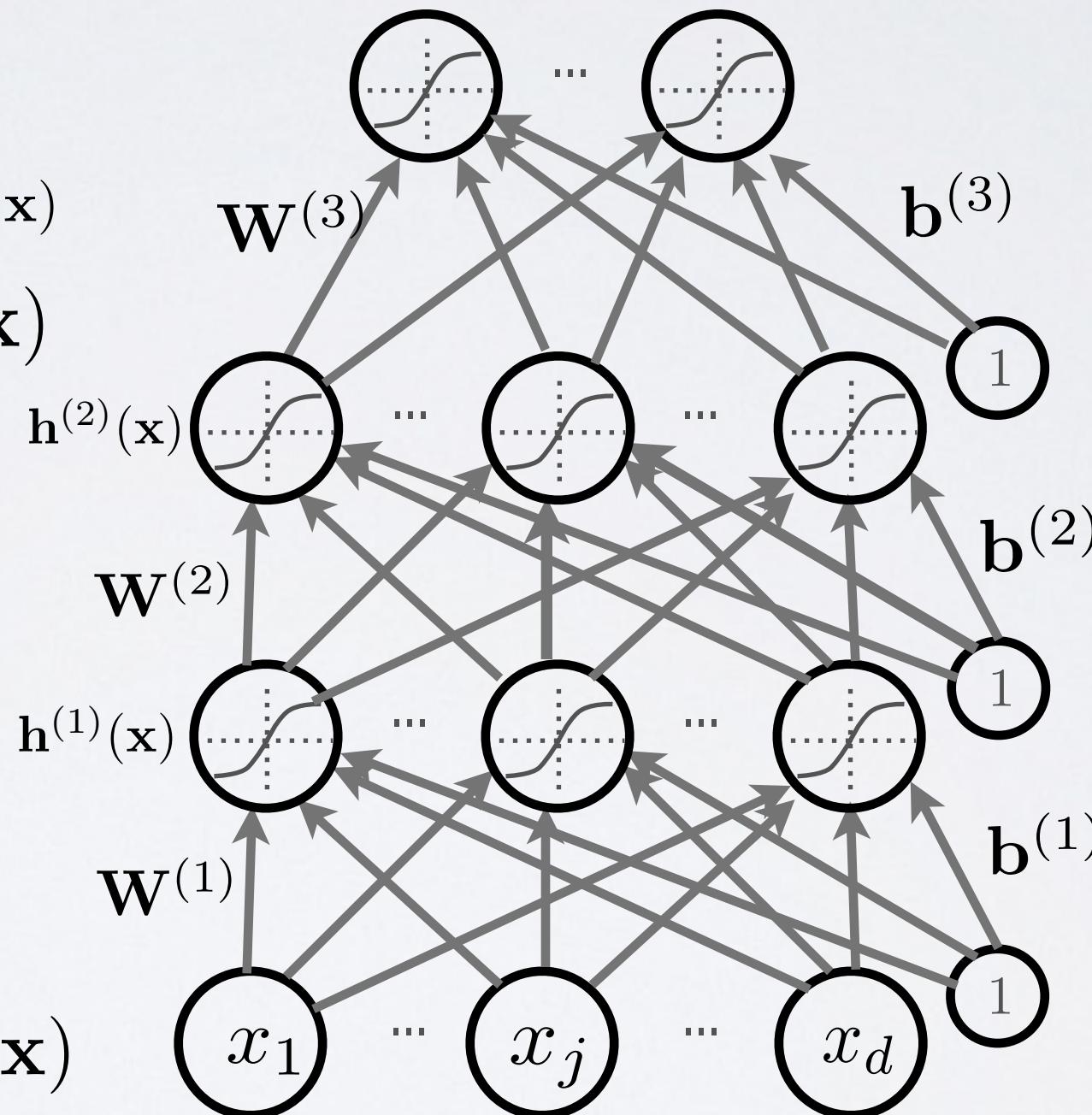
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

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

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

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

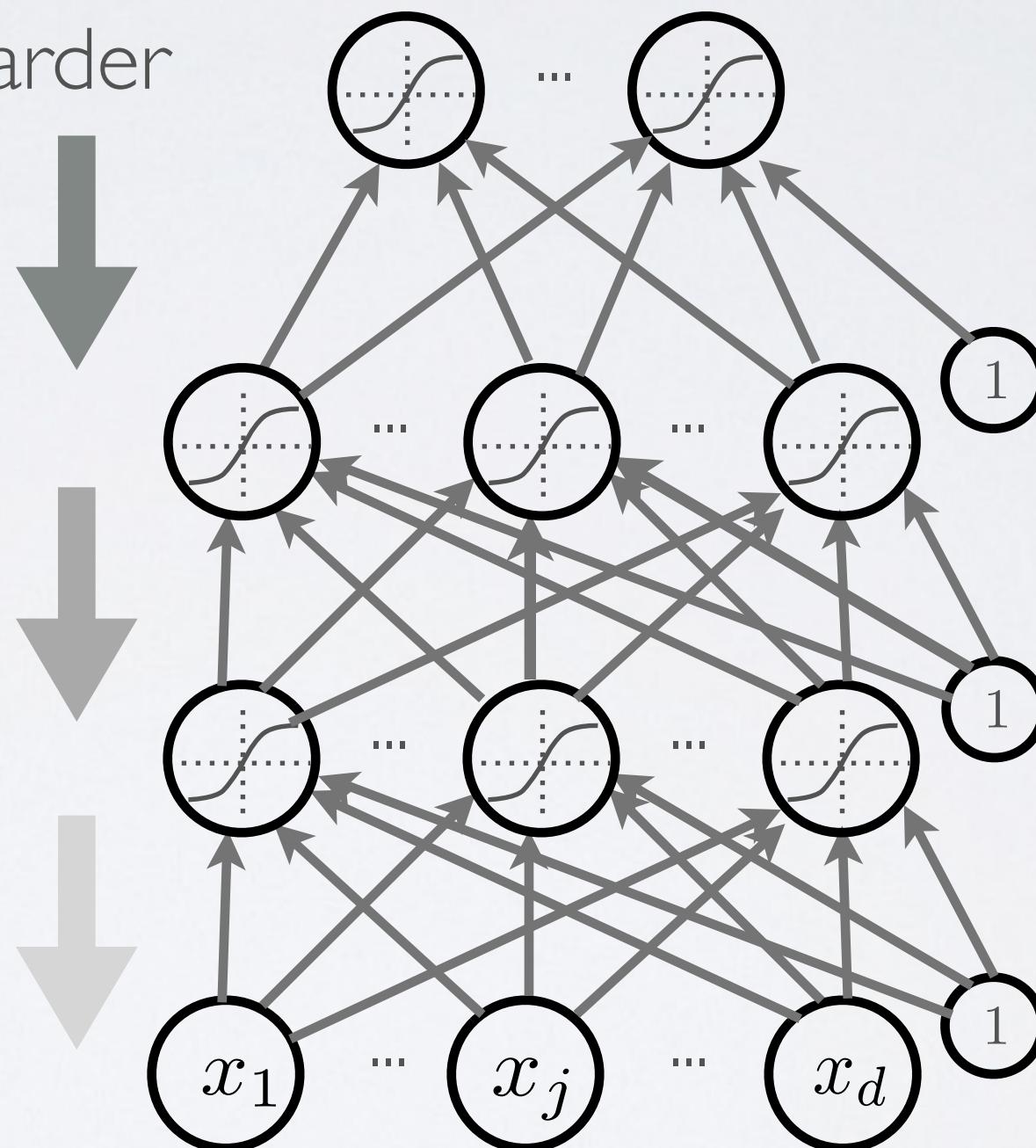
$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



# DEEP LEARNING

## Topics: why training is hard

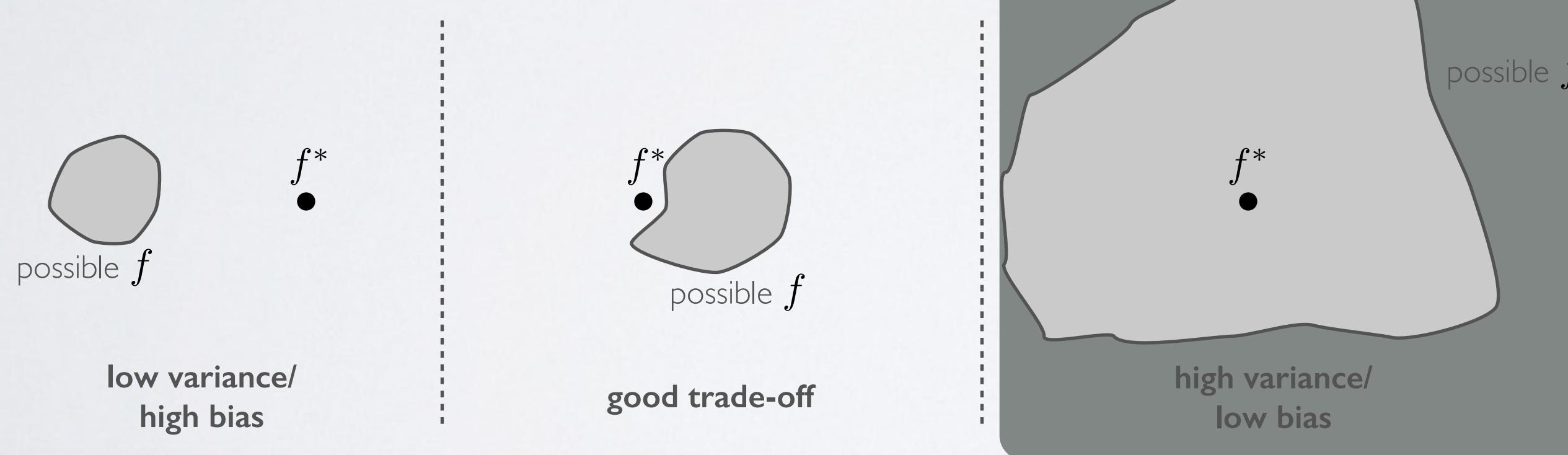
- First hypothesis: optimization is harder (underfitting)
  - ▶ vanishing gradient problem
  - ▶ saturated units block gradient propagation
- This is a well known problem in recurrent neural networks



# 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

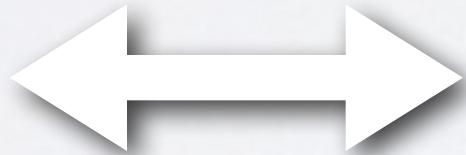
**Topics:** unsupervised pre-training

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



character image

Why is one  
a character  
and the other  
is not ?



random image

- ▶ encourage hidden layers to encode that structure

# UNSUPERVISED PRE-TRAINING

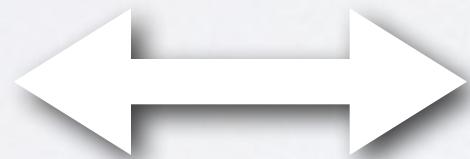
**Topics:** unsupervised pre-training

- 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 ?



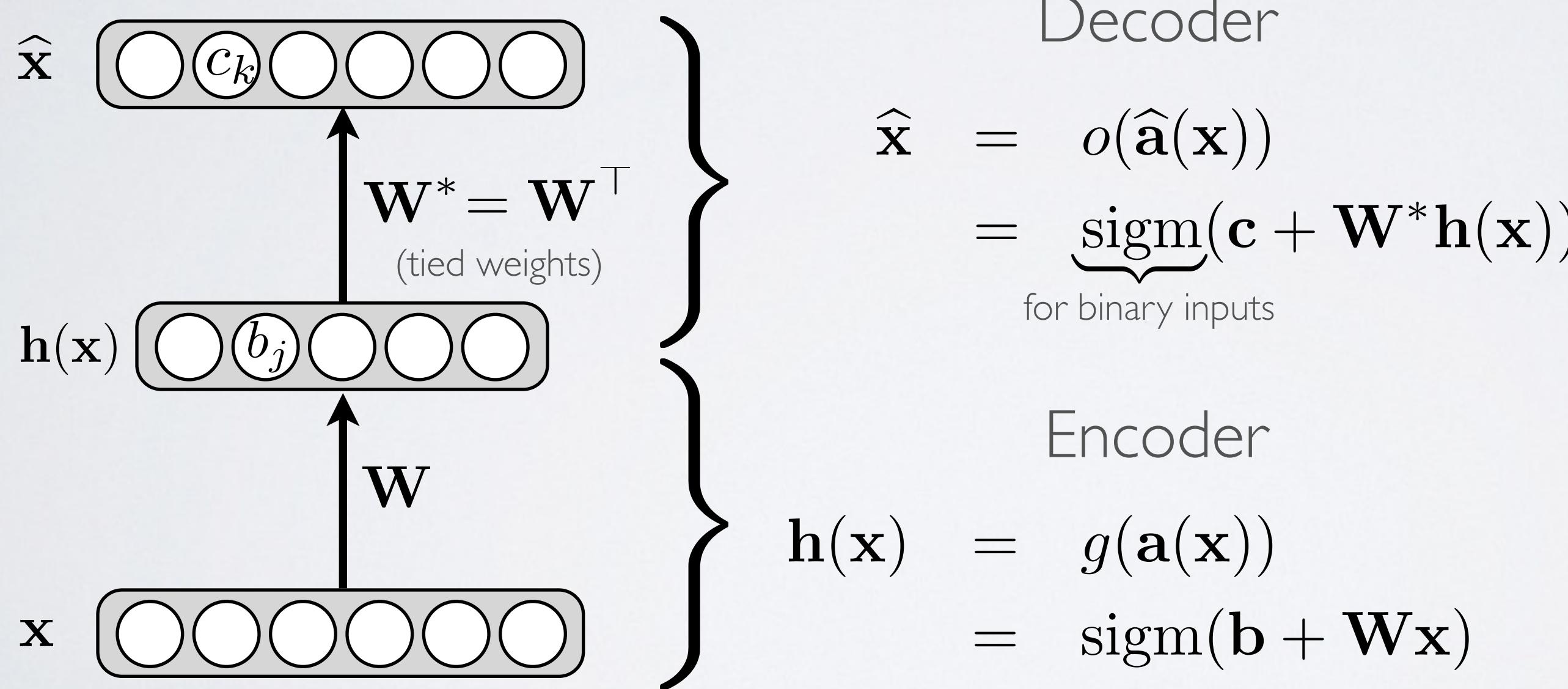
random image

- ▶ 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

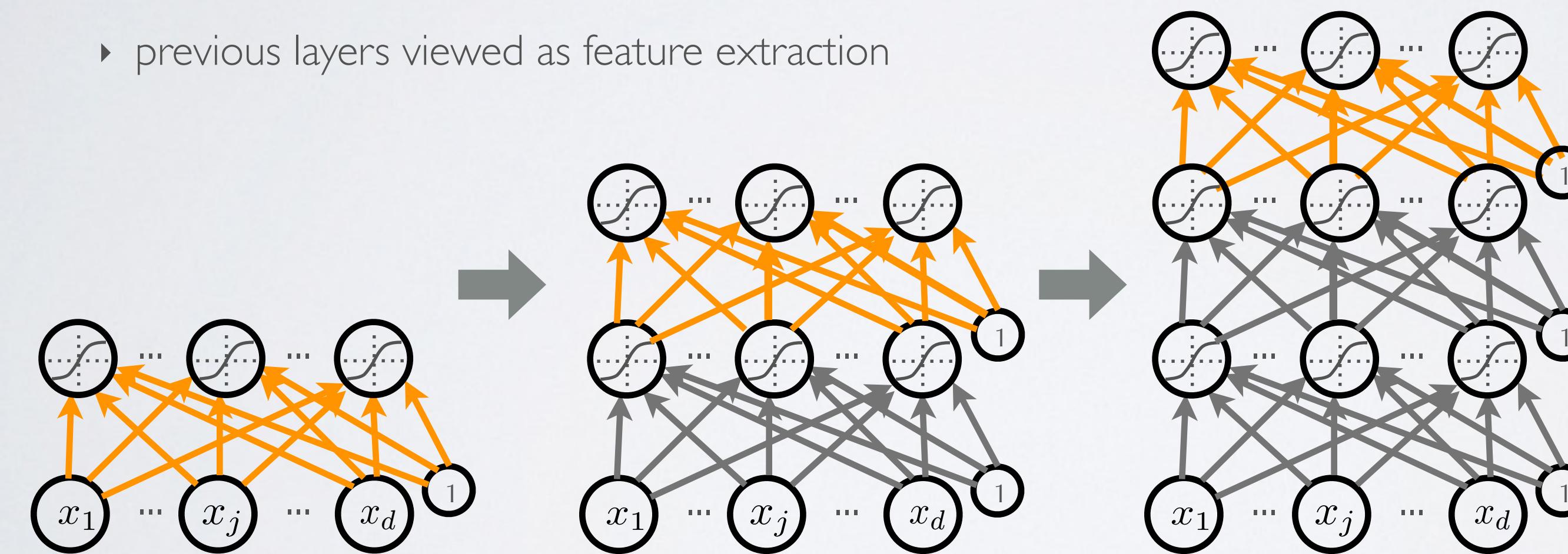


# UNSUPERVISED PRE-TRAINING

**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



# UNSUPERVISED PRE-TRAINING

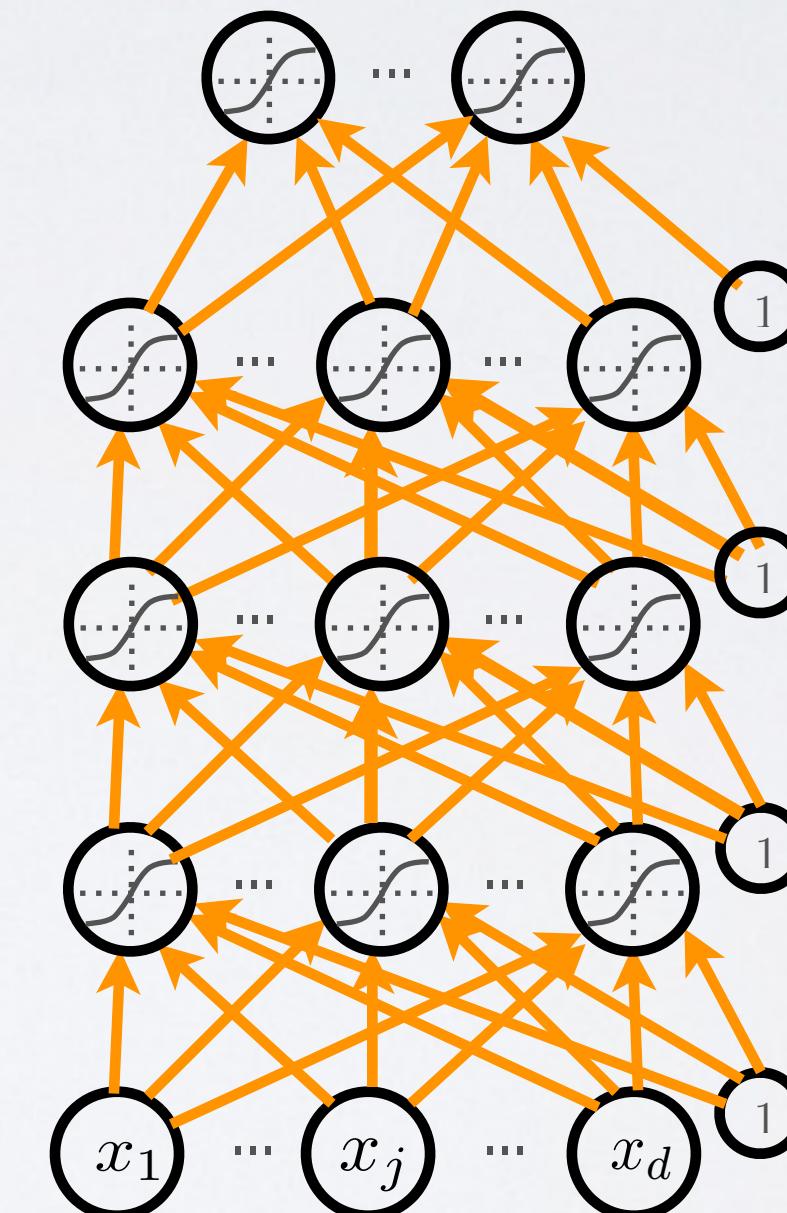
**Topics:** unsupervised pre-training

- 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



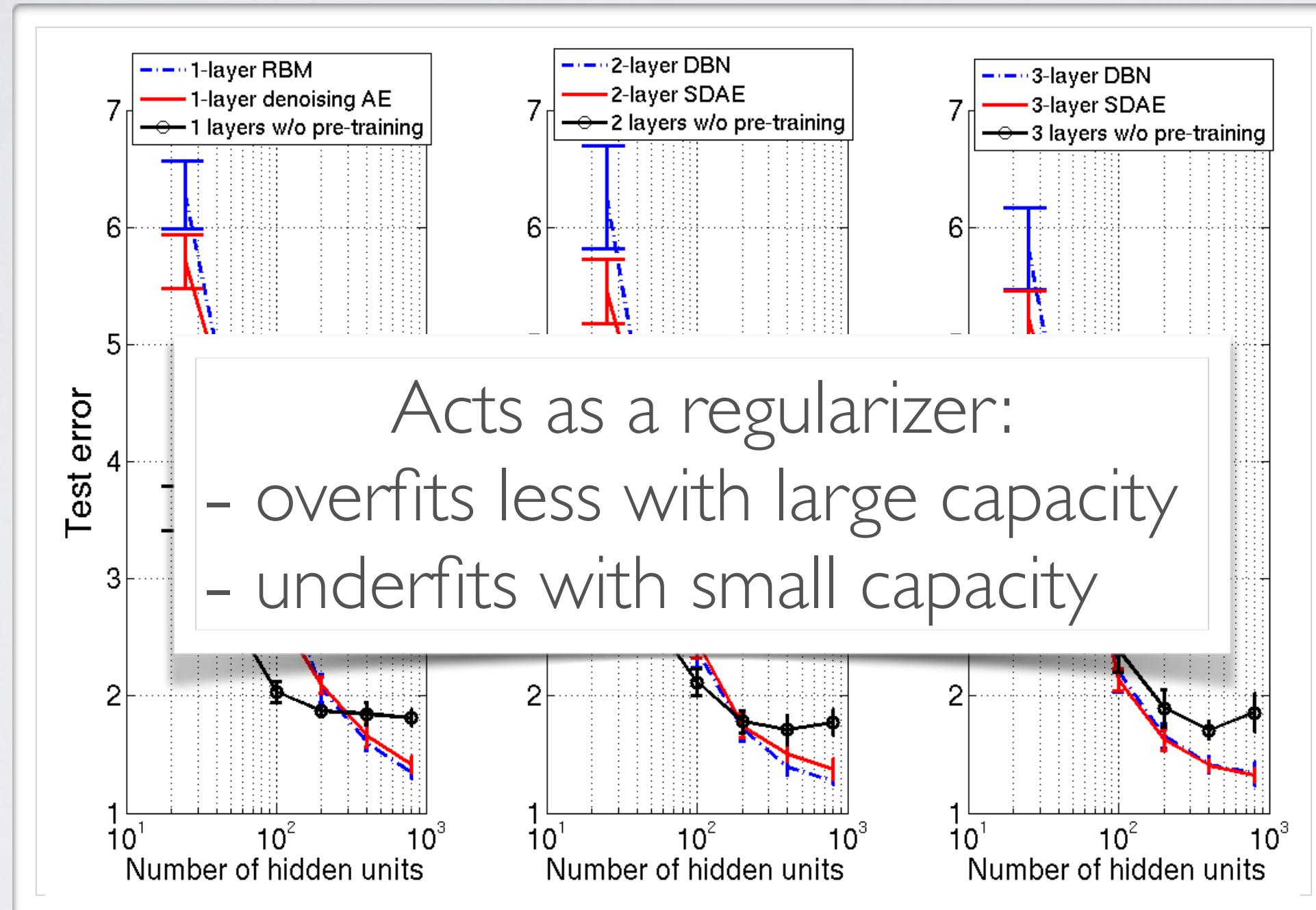
# DEEP LEARNING

## 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  $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$
  - 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**
- pre-training**

# 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

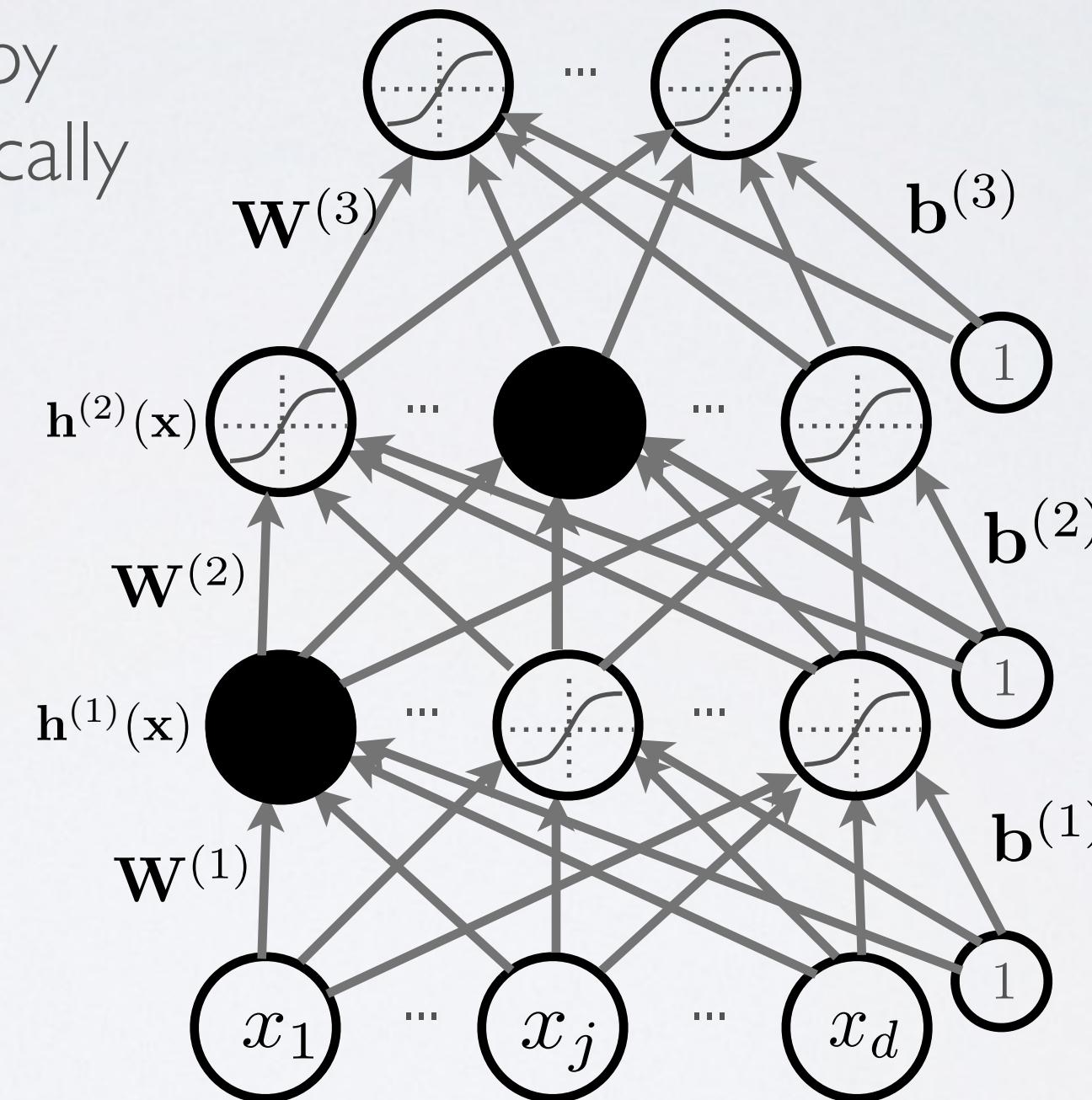
**Topics:** why training is hard

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

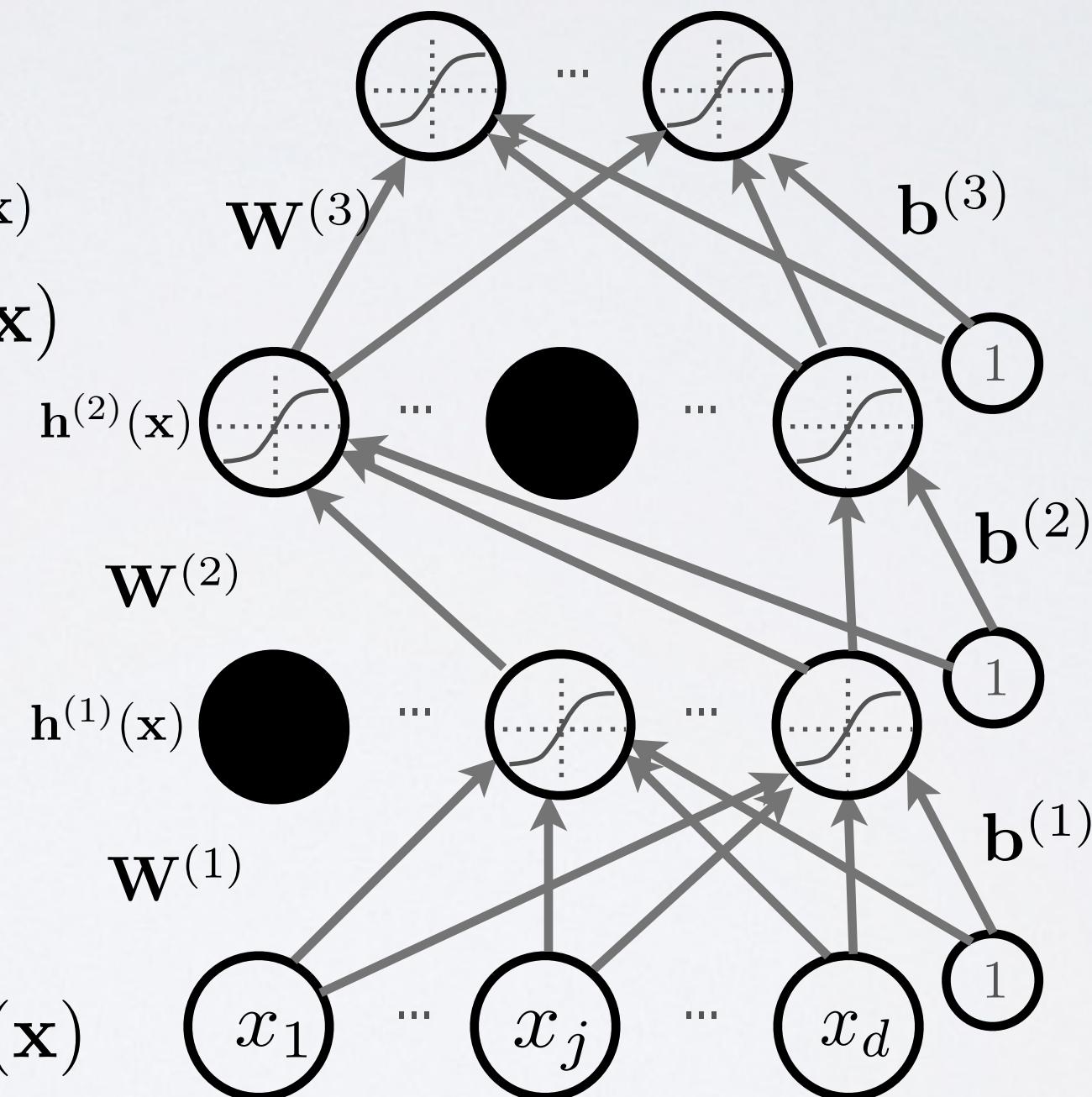
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

► 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)

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

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

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**)

# 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  $y_i \leftarrow \gamma \hat{x}_i + \beta$  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  $\gamma$  and  $\beta$
- 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 networks

Training neural networks - types of learning

# SUPERVISED LEARNING

**Topics:** supervised learning

- Training time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

- Test time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

- Example

- ▶ classification
  - ▶ regression

# UNSUPERVISED LEARNING

**Topics:** unsupervised learning

- Training time

- ▶ data :

$$\{\mathbf{x}^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)} \sim p(\mathbf{x})$$

- Test time

- ▶ data :

$$\{\mathbf{x}^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)} \sim p(\mathbf{x})$$

- Example

- ▶ distribution estimation
  - ▶ dimensionality reduction

# SEMI-SUPERVISED LEARNING

**Topics:** semi-supervised learning

- Training time

- data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

$$\{\mathbf{x}^{(t)}\}$$

- setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

$$\mathbf{x}^{(t)} \sim p(\mathbf{x})$$

- Test time

- data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

- setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

# MULTITASK LEARNING

## Topics: multitask learning

- Training time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)} \sim$$

$$p(\mathbf{x}, y_1, \dots, y_M)$$

- Test time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)} \sim$$

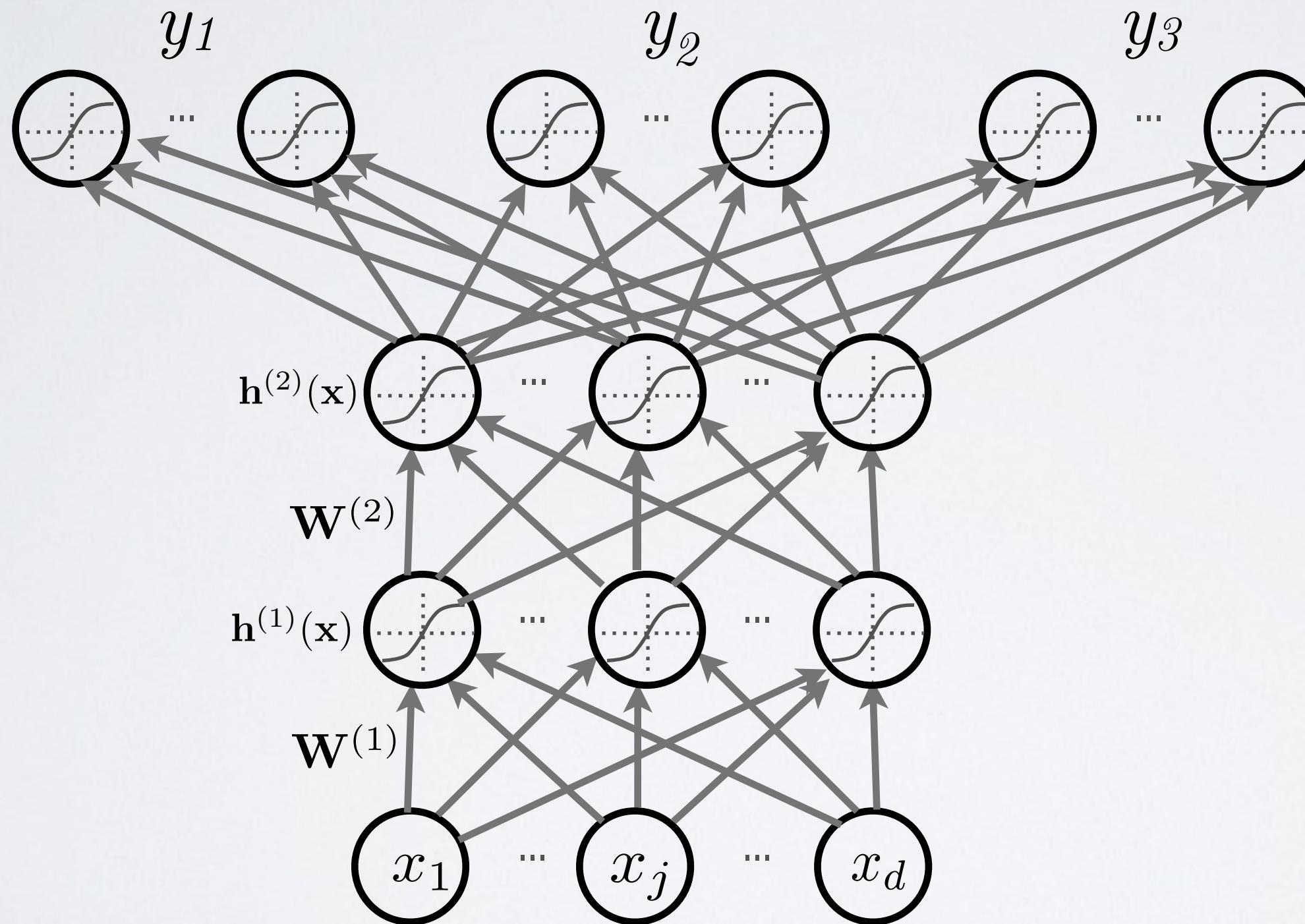
$$p(\mathbf{x}, y_1, \dots, y_M)$$

- Example

- ▶ object recognition in images with multiple objects

# MULTITASK LEARNING

**Topics:** multitask learning



# TRANSFER LEARNING

**Topics:** transfer learning

- Training time

- data :

$$\{\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)}\}$$

- setting :

$$\mathbf{x}^{(t)}, y_1^{(t)}, \dots, y_M^{(t)} \sim p(\mathbf{x}, y_1, \dots, y_M)$$

- Test time

- data :

$$\{\mathbf{x}^{(t)}, y_1^{(t)}\}$$

- setting :

$$\mathbf{x}^{(t)}, y_1^{(t)} \sim p(\mathbf{x}, y_1)$$

# STRUCTURED OUTPUT PREDICTION

**Topics:** structured output prediction

- Training time

- ▶ data :

$$\{\mathbf{x}^{(t)}, \mathbf{y}^{(t)}\}$$

of arbitrary structure  
(vector, sequence, graph)

- ▶ setting :

$$\mathbf{x}^{(t)}, \mathbf{y}^{(t)} \sim p(\mathbf{x}, \mathbf{y})$$

- Test time

- ▶ data :

$$\{\mathbf{x}^{(t)}, \mathbf{y}^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, \mathbf{y}^{(t)} \sim p(\mathbf{x}, \mathbf{y})$$

- Example

- ▶ image caption generation
  - ▶ machine translation

# DOMAIN ADAPTATION

**Topics:** domain adaptation, covariate shift

- Training time

- ▶ data :

$$\begin{aligned}\{\mathbf{x}^{(t)}, y^{(t)}\} \\ \{\bar{\mathbf{x}}^{(t')}\}\end{aligned}$$

- ▶ setting :

$$\mathbf{x}^{(t)} \sim p(\mathbf{x})$$

$$y^{(t)} \sim p(y|\mathbf{x}^{(t)})$$

$$\bar{\mathbf{x}}^{(t)} \sim q(\mathbf{x}) \approx p(\mathbf{x})$$

- Test time

- ▶ data :

$$\{\bar{\mathbf{x}}^{(t)}, y^{(t)}\}$$

- ▶ setting :

$$\bar{\mathbf{x}}^{(t)} \sim q(\mathbf{x})$$

$$y^{(t)} \sim p(y|\bar{\mathbf{x}}^{(t)})$$

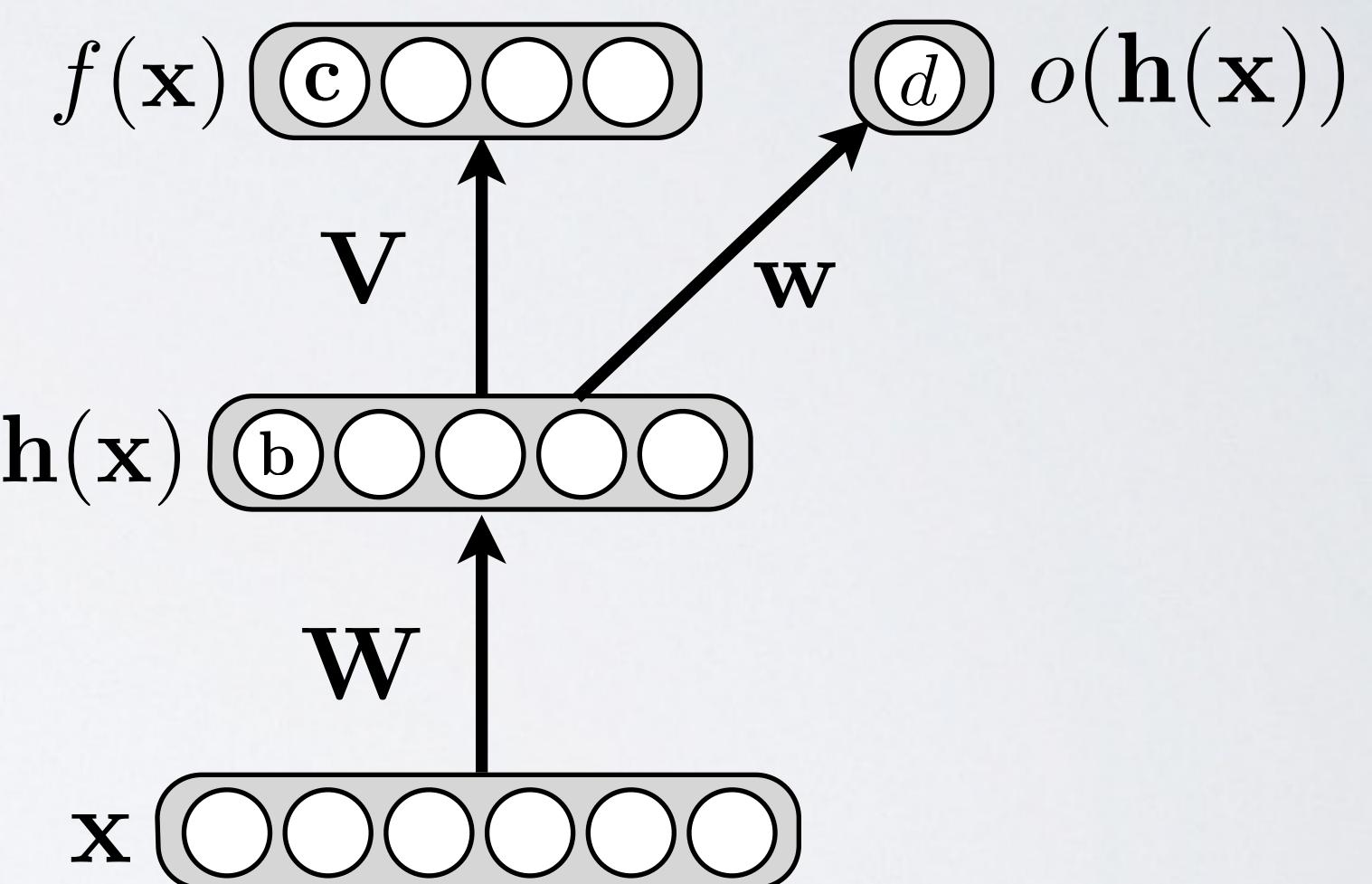
- Example

- ▶ classify sentiment in reviews of different products

# DOMAIN ADAPTATION

**Topics:** domain adaptation, covariate shift

- Domain-adverarial networks (Ganin et al. 2015)  
train hidden layer representation to be
  1. **predictive** of the target class
  2. **indiscriminate** of the domain
- Trained by stochastic gradient descent
  - ▶ for each random pair  $\mathbf{x}^{(t)}, \bar{\mathbf{x}}^{(t')}$ 
    1. update  $\mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}$  in opposite direction of gradient
    2. update  $\mathbf{w}, d$  in direction of gradient



# ONE-SHOT LEARNING

## Topics: one-shot learning

- Training time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

subject to  $y^{(t)} \in \{1, \dots, C\}$

- Test time

- ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

- ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

subject to  $y^{(t)} \in \{C + 1, \dots, C + M\}$

- ▶ additional data :

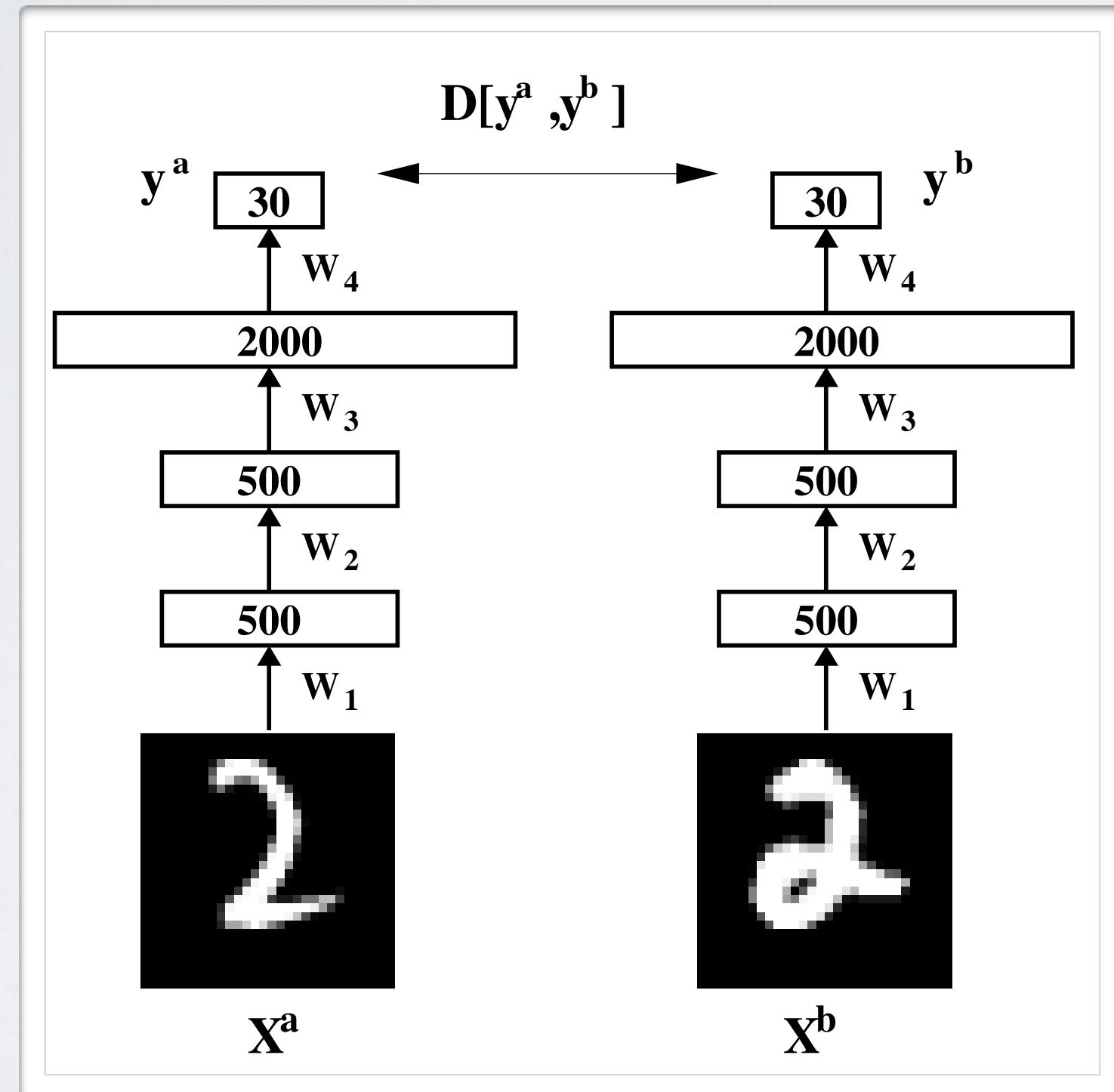
- a single labeled example from each of the  $M$  new classes

- Example

- ▶ recognizing a person based on a single picture of him/her

# ONE-SHOT LEARNING

**Topics:** one-shot learning



Siamese architecture  
(figure taken from Salakhutdinov  
and Hinton, 2007)

# ZERO-SHOT LEARNING

**Topics:** zero-shot learning, zero-data learning

- Training time

  - ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

  - ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

subject to  $y^{(t)} \in \{1, \dots, C\}$

  - ▶ additional data :

    - description vector  $\mathbf{z}_c$  of each of the  $C$  classes

- Test time

  - ▶ data :

$$\{\mathbf{x}^{(t)}, y^{(t)}\}$$

  - ▶ setting :

$$\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$$

subject to  $y^{(t)} \in \{C + 1, \dots, C + M\}$

  - ▶ additional data :

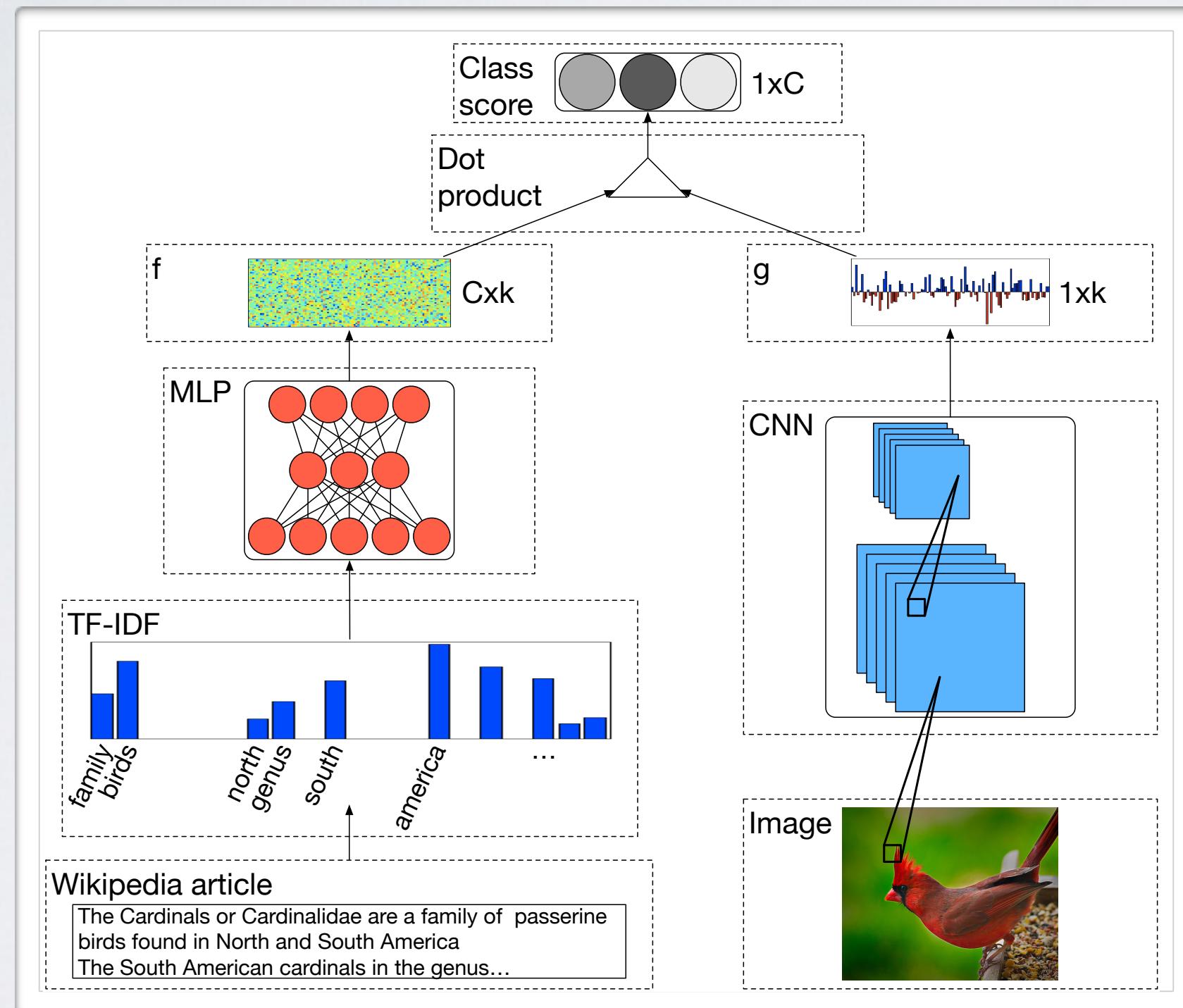
    - description vector  $\mathbf{z}_c$  of each of the new  $M$  classes

- Example

  - ▶ recognizing an object based on a worded description of it

# ZERO-SHOT LEARNING

**Topics:** zero-shot learning, zero-data learning



Ba, Swersky, Fidler, Salakhutdinov  
arxiv 2015

# 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](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 bias unit  $b_j$  is associated with the hidden layer, and a bias unit  $c_k$  is associated with the visible layer. The layers are connected by a weight matrix  $\mathbf{W}$ . A cursor icon is visible on the left side of the diagram.

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!!