10703 Deep Reinforcement Learning and Control

Russ Salakhutdinov

Machine Learning Department
rsalakhu@cs.cmu.edu

Deep Learning I

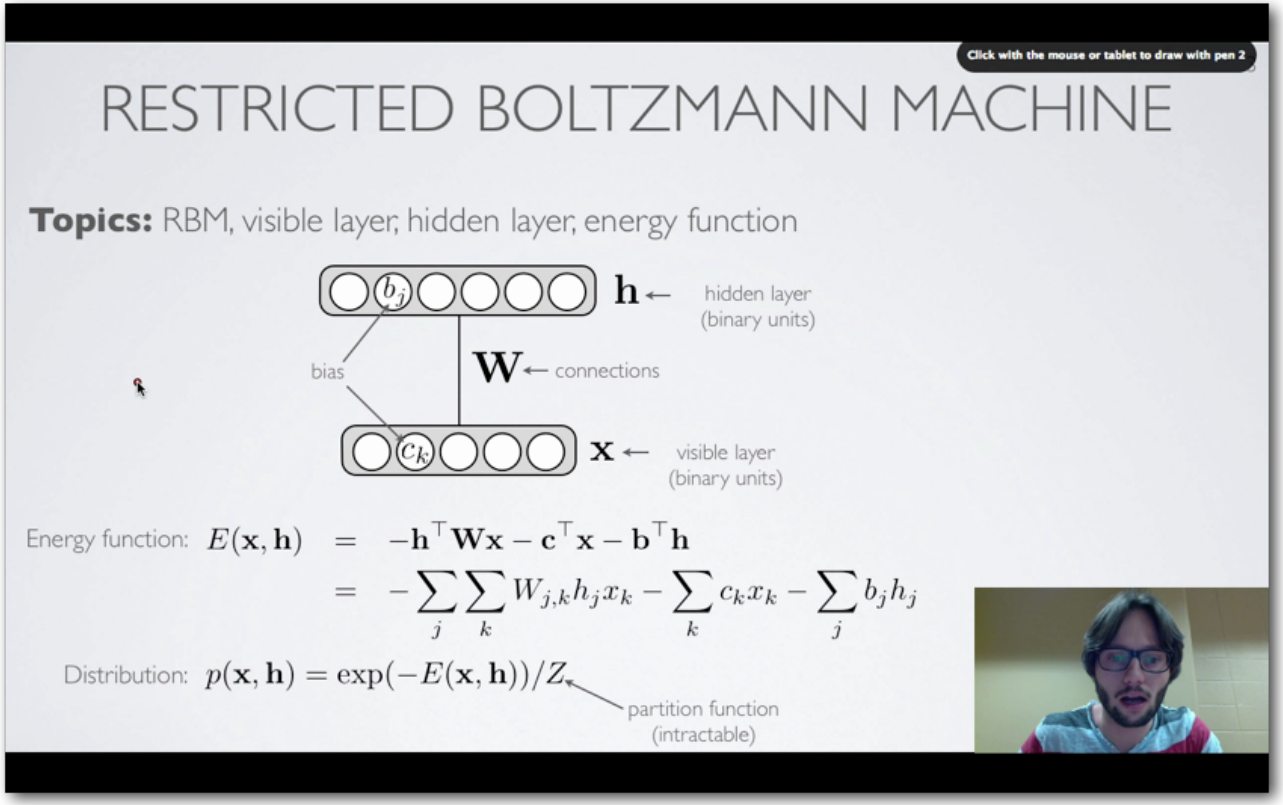
Neural Networks Online Course

• **Disclaimer**: Some of the material and slides for this lecture were borrowed from Hugo Larochelle's class on Neural Networks: https://sites.google.com/site/deeplearningsummerschool2016/

• Hugo's class covers many other topics: convolutional networks, neural language model, Boltzmann machines, autoencoders, sparse coding, etc.

• We will use his material for some of the other lectures.

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



Large-Scale Reinforcement Learning

- Reinforcement learning can be used to solve large problems, e.g.
  - Backgammon: 10^{20} states
  - Computer Go: 10^{170} states
  - Helicopter: continuous state space

- How can we scale up the model-free methods for prediction and control?

Value Function Approximation (VFA)

- So far we have represented value function by a lookup table
  - Every state s has an entry V(s), or
  - Every state-action pair (s,a) has an entry Q(s,a)
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with function approximation

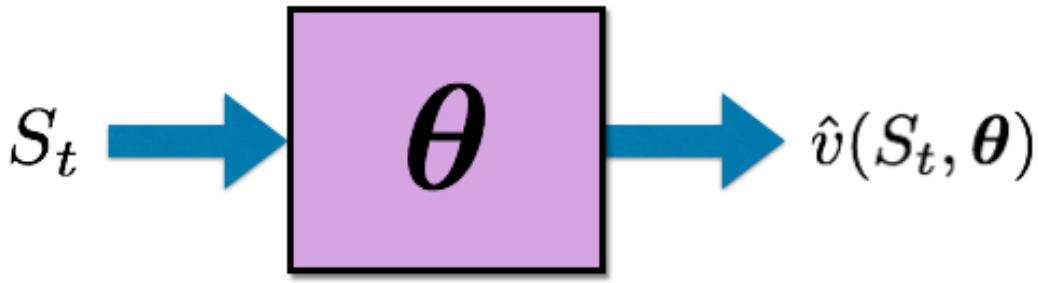
$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

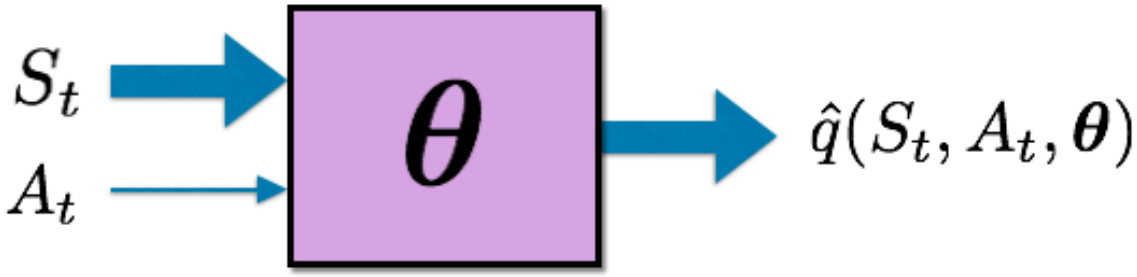
$$\text{or } \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

  - Generalize from seen states to unseen states

Value Function Approximation (VFA)

- Value function approximation (VFA) replaces the table with a general parameterized form:



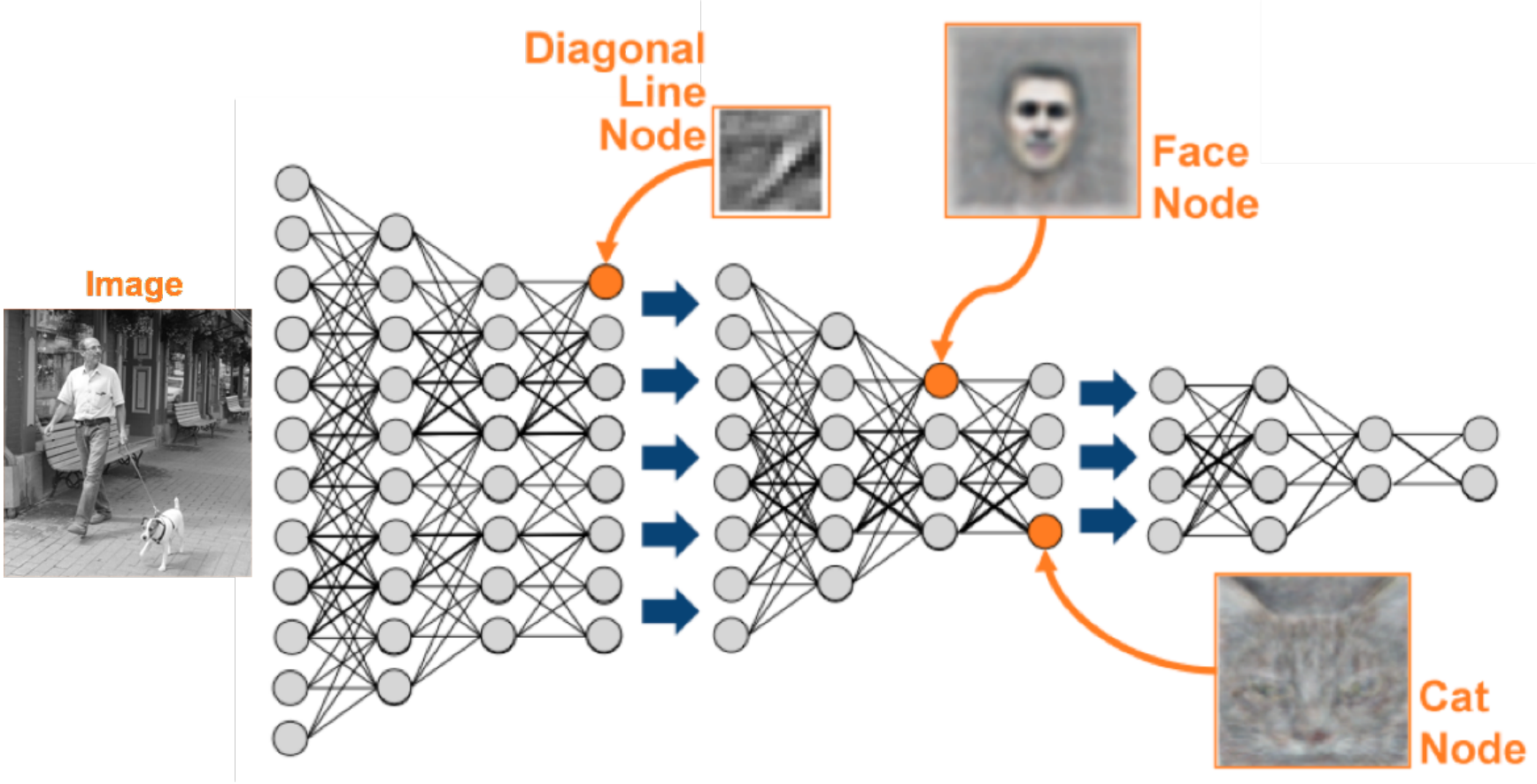


Which Function Approximation?

- There are many function approximators, e.g.
  - Linear combinations of features
  - Neural networks
  - Decision tree
  - Nearest neighbour
  - Fourier / wavelet bases
  - …

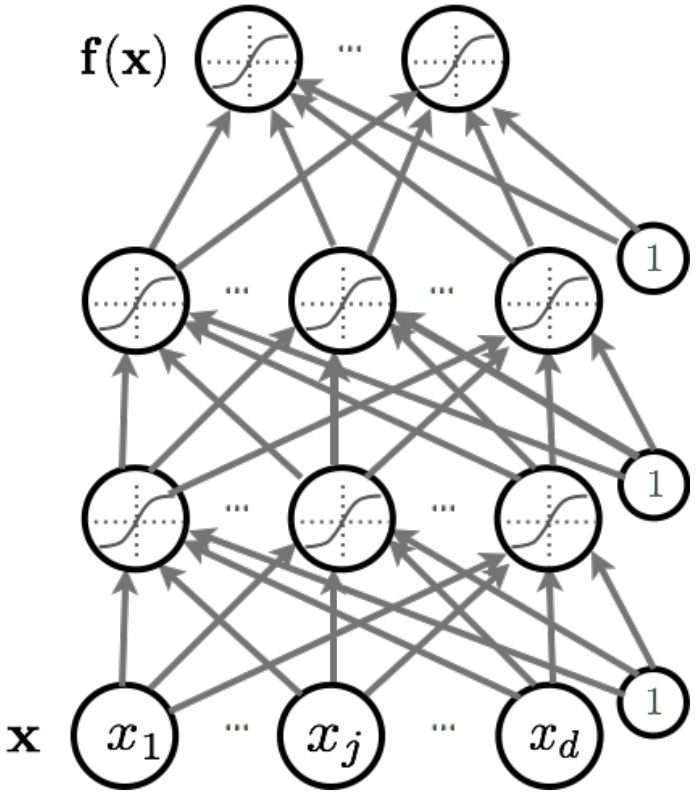
- We consider differentiable function approximators, e.g.
  - Linear combinations of features
  - **Neural networks**

Deep Learning



Feedforward Neural Networks

- Definition of Neural Networks
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- How to train neural nets:
  - Loss function
  - Backpropagation with gradient descent
- More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



Artificial Neuron

- 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:

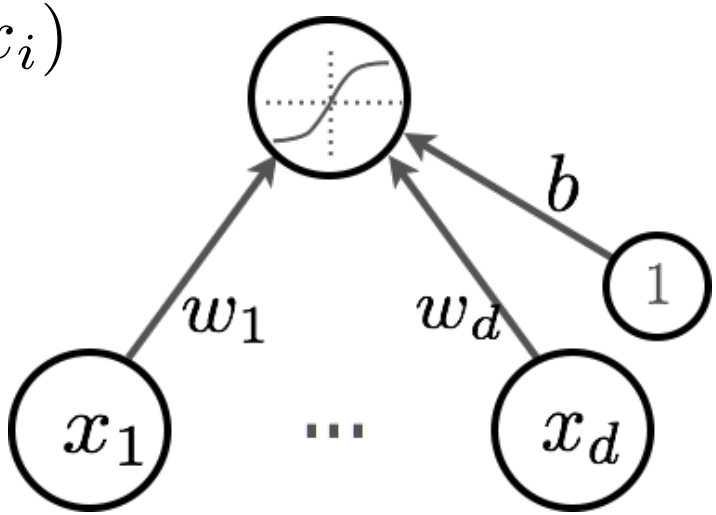
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

where

$\mathbf{w}$ are the weights (parameters)

b is the bias term

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

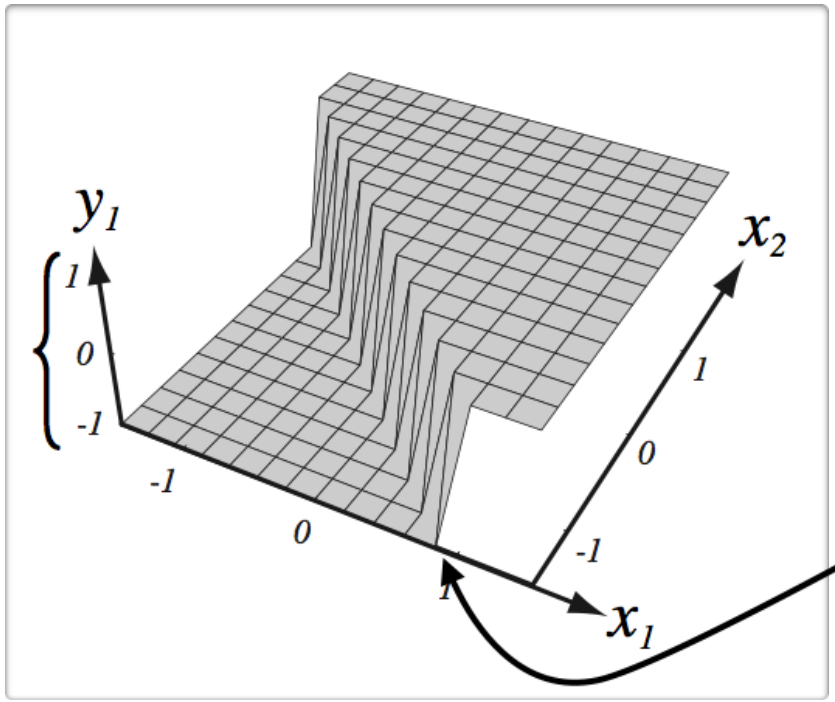


Artificial Neuron

- Output activation of the neuron:

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

Range is determined by $g(\cdot)$



(from Pascal Vincent's slides)

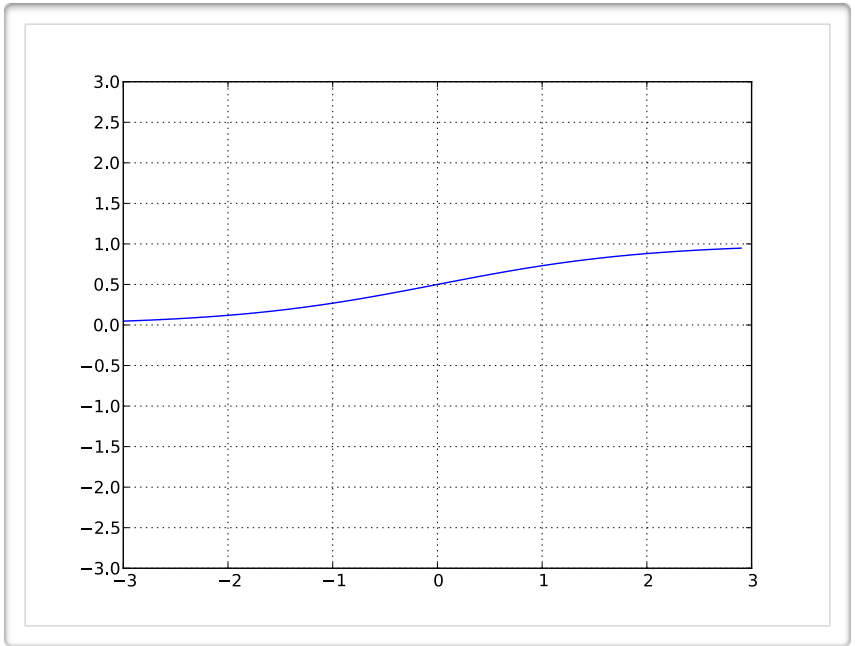
Bias only changes the position of the riff

Activation Function

- Sigmoid activation function:

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

  - Squashes the neuron's output between 0 and 1
  - Always positive
  - Bounded
  - Strictly Increasing

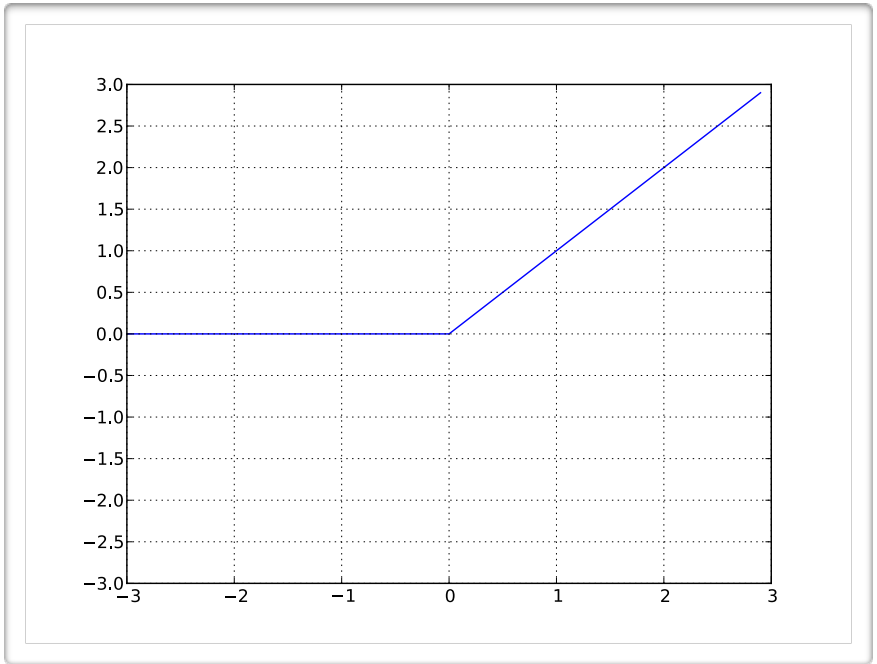


Activation Function

- Rectified linear (ReLU) activation function:

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

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



Single Hidden Layer Neural Net

- Hidden layer pre-activation:

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

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

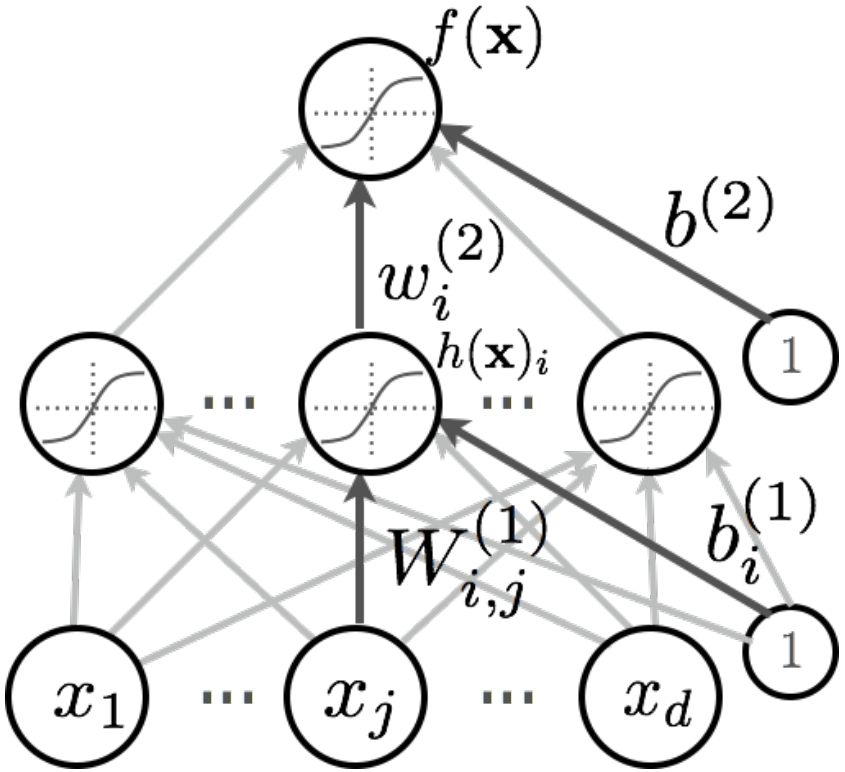
- Hidden layer activation:

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

- Output layer activation:

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

Output activation function



Multilayer Neural Net

- Consider a network with L hidden layers.
  - layer pre-activation for k>0

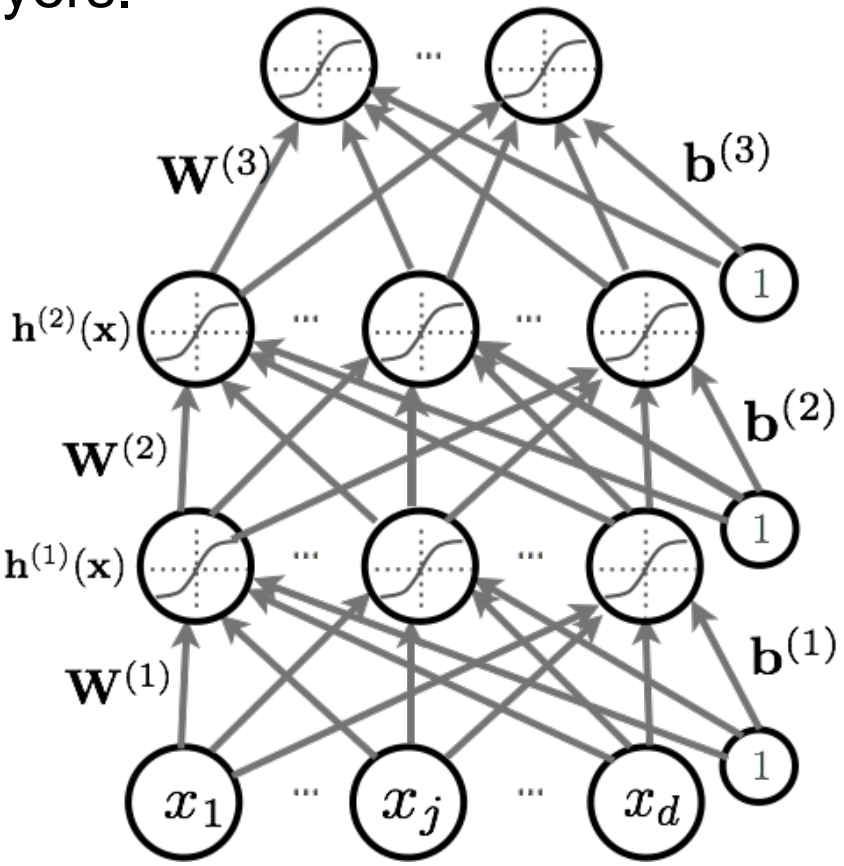
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation from 1 to L:

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

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

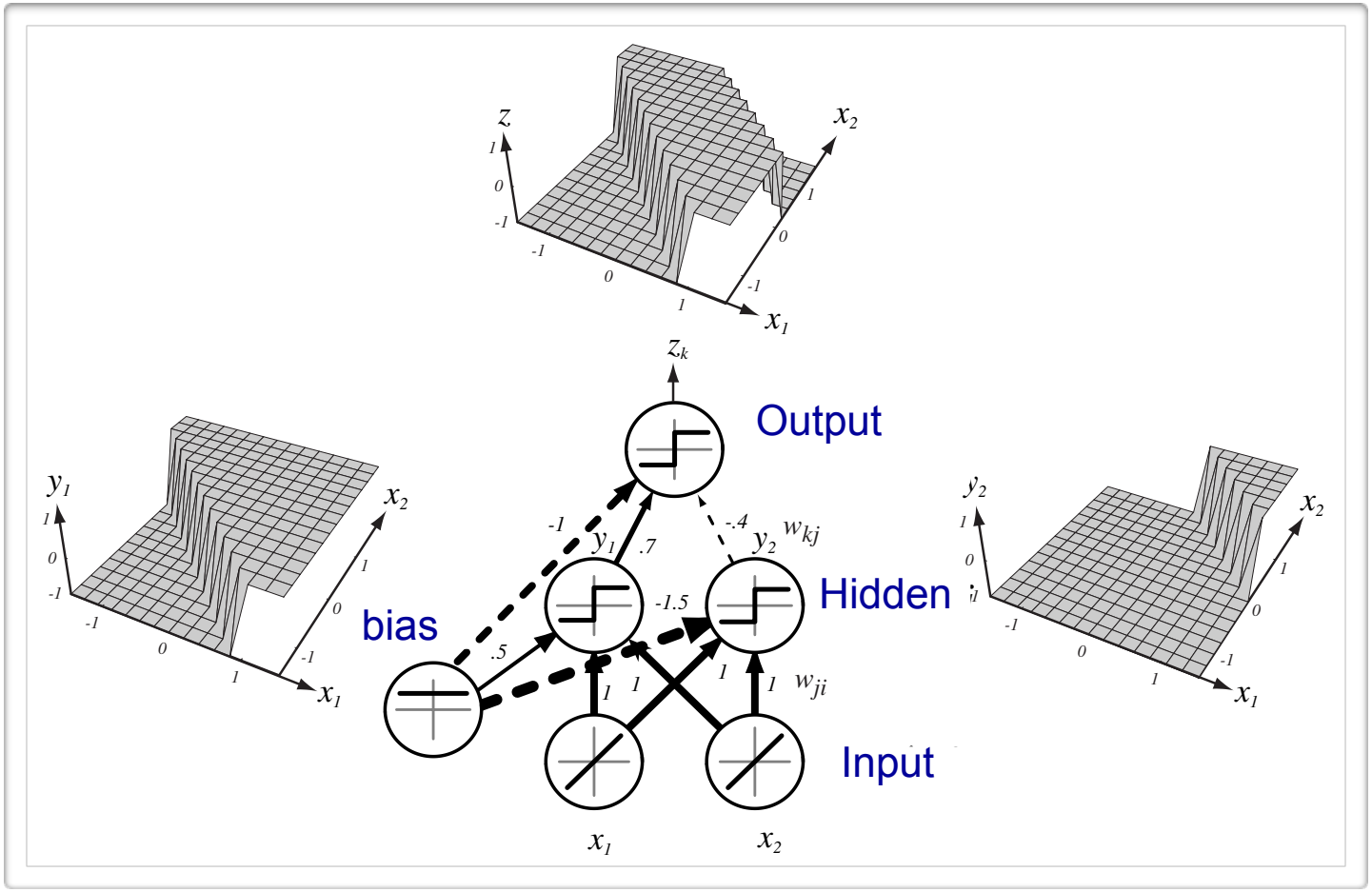
$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



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

Capacity of Neural Nets

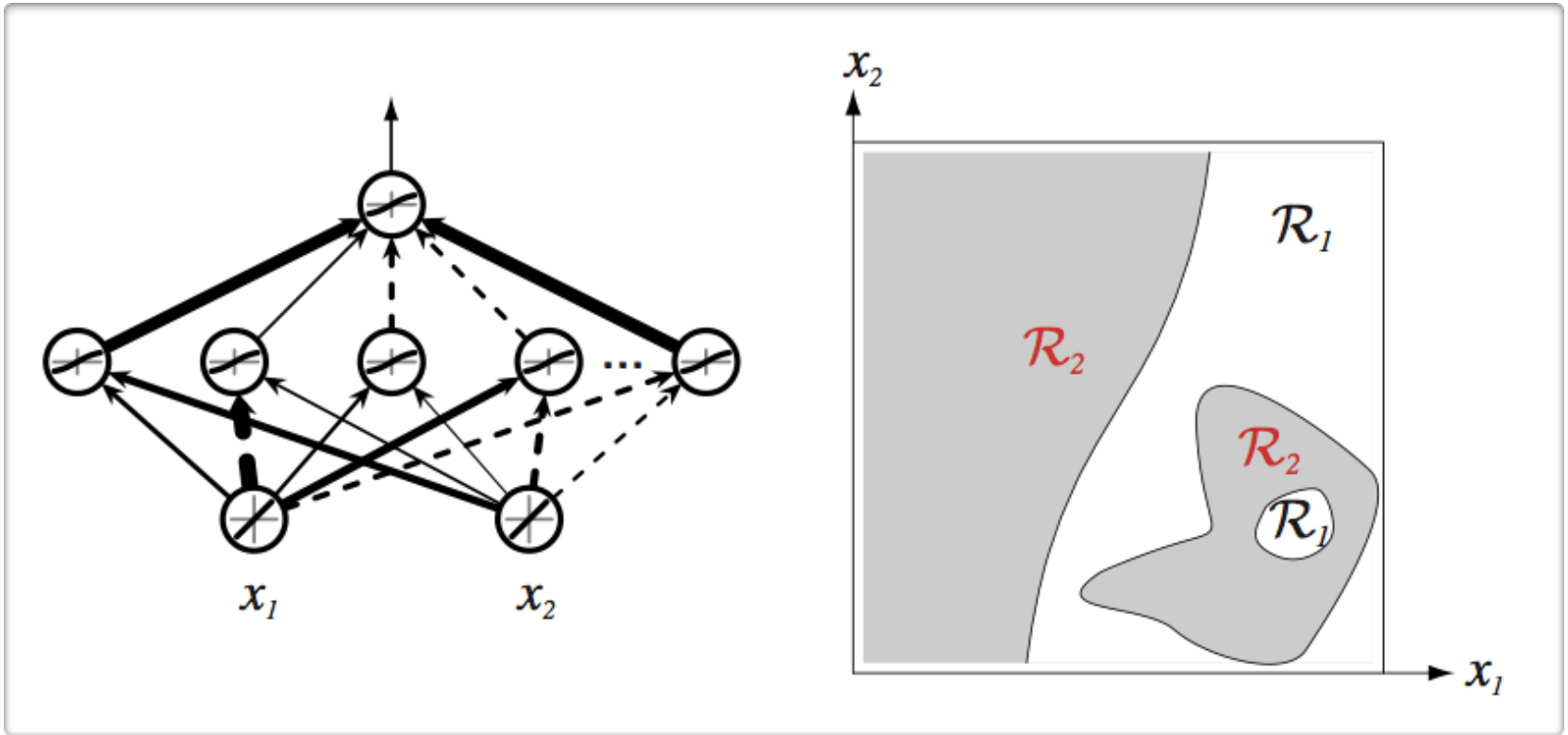
- Consider a single layer neural network



(from Pascal Vincent's slides)

Capacity of Neural Nets

- Consider a single layer neural network



(from Pascal Vincent's slides)

Universal Approximation

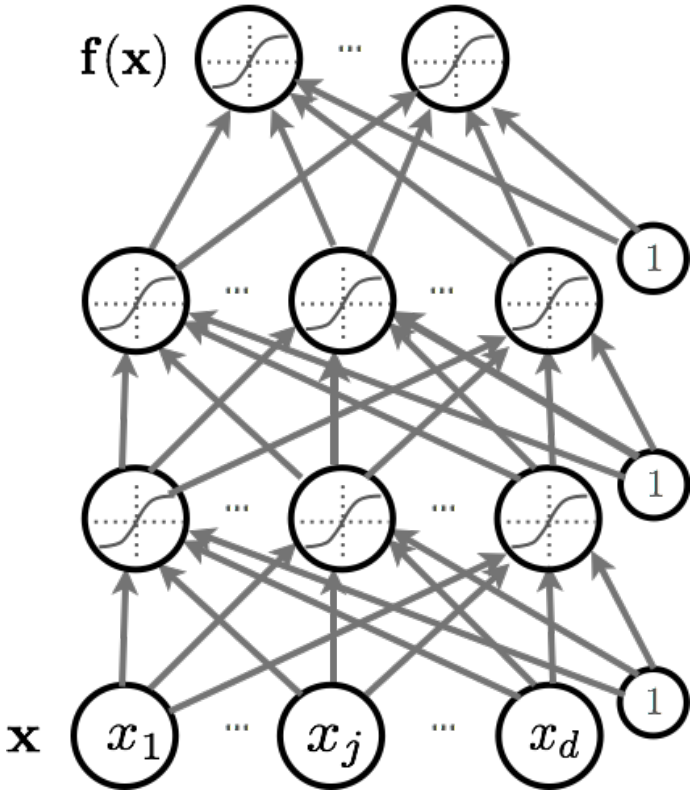
- Universal Approximation Theorem (Hornik, 1991):
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"

- This applies for sigmoid, tanh and many other activation functions.

- However, this does not mean that there is learning algorithm that can find the necessary parameter values.

Feedforward Neural Networks

- How neural networks predict f(x) given an input x:
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- How to train neural nets:
  - Loss function
  - Backpropagation with gradient descent
- More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



Training

- Empirical Risk Minimization:

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

- Learning is cast as optimization.
  - For classification problems, we would like to minimize classification error.
  - Loss function can sometimes be viewed as a surrogate for what we want to optimize (e.g. upper bound)

Stochastic Gradient Descend

- Perform updates after seeing each example:
  - Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
  - For t=1:T
    - 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 of all examples

- To train a neural net, we need:
  - Loss function: $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - A procedure to compute gradients: $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - Regularizer and its gradient: $\Omega(\boldsymbol{\theta}), \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$

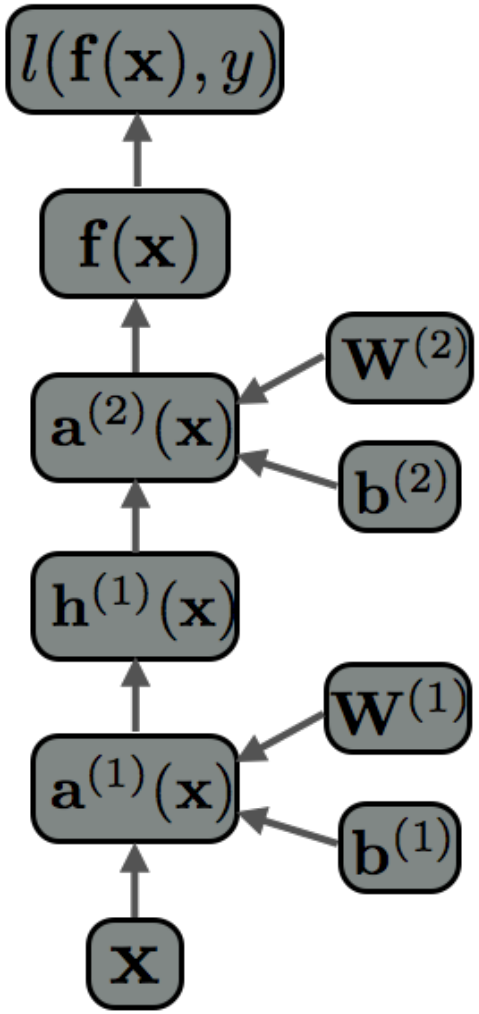
Computational Flow Graph

• Forward propagation can be represented as an acyclic flow graph

• Forward propagation can be implemented in a modular way:

- Each box can 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 yields forward propagation

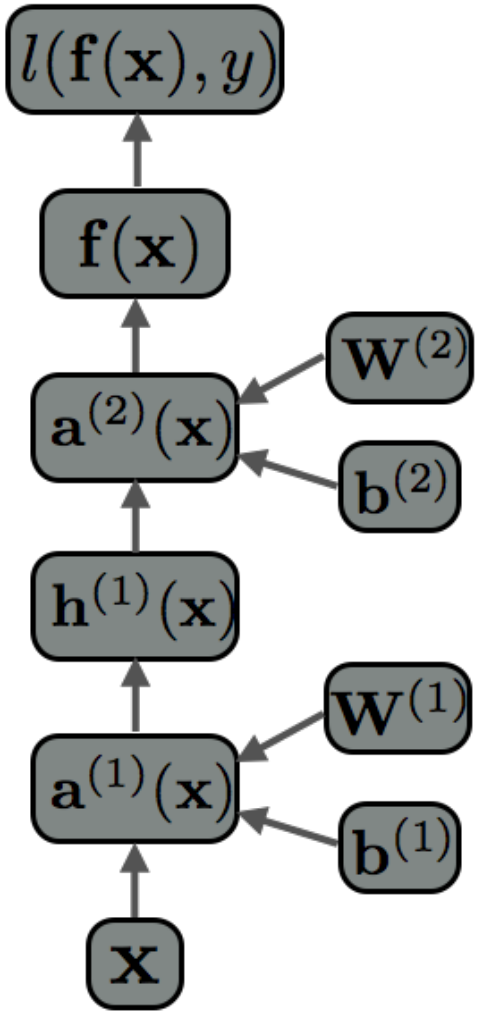


Computational Flow Graph

- Each object also has a bprop method

  – it computes the gradient of the loss with respect to each child box.

- By calling bprop in the reverse order, we obtain backpropagation



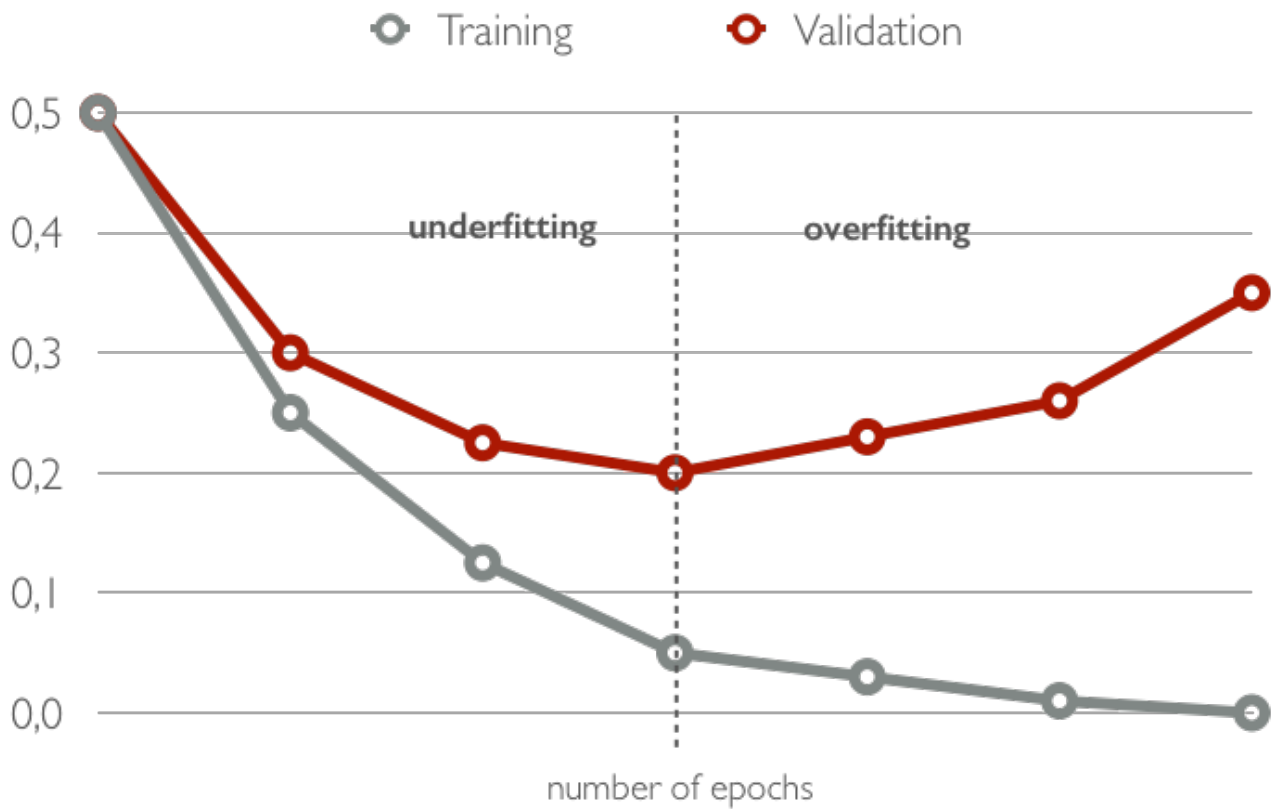
Model Selection

- Training Protocol:
  - Train your model on the Training Set $\mathcal{D}^{\text{train}}$
  - For model selection, use Validation Set $\mathcal{D}^{\text{valid}}$
    - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.
  - Estimate generalization performance using the Test Set $\mathcal{D}^{\text{test}}$

- Generalization is the behavior of the model on **unseen examples**.

Early Stopping

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



Mini-batch, Momentum

• Make updates based on a mini-batch of examples (instead of a single example):

- the gradient is the average regularized loss for that mini-batch
- can give a more accurate estimate of the gradient
- can leverage matrix/matrix operations, which are more efficient

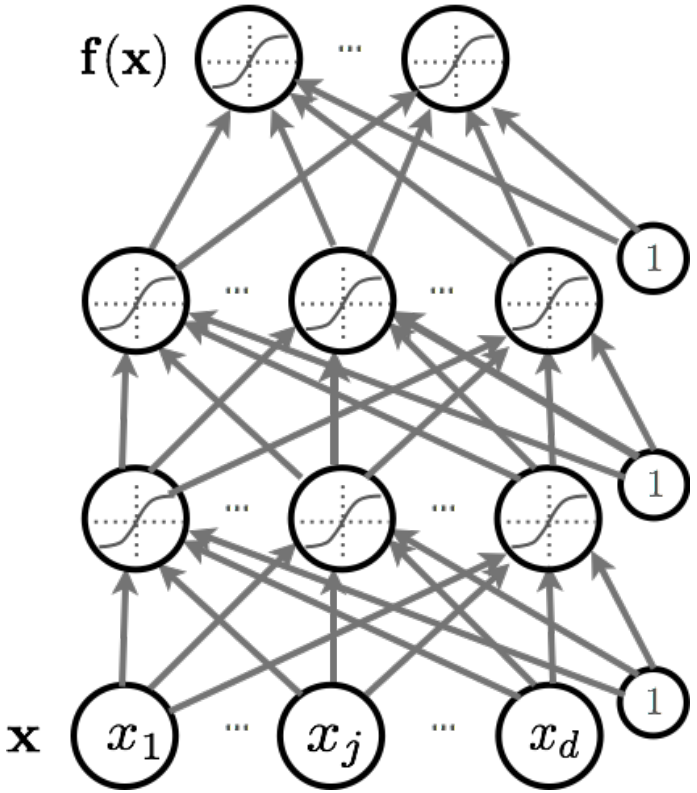
• Momentum: Can use an exponential average of previous gradients:

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

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

Feedforward Neural Networks

- How neural networks predict f(x) given an input x:
  - Forward propagation
  - Types of units
  - Capacity of neural networks
- How to train neural nets:
  - Loss function
  - Backpropagation with gradient descent
- More recent techniques:
  - Dropout
  - Batch normalization
  - Unsupervised Pre-training



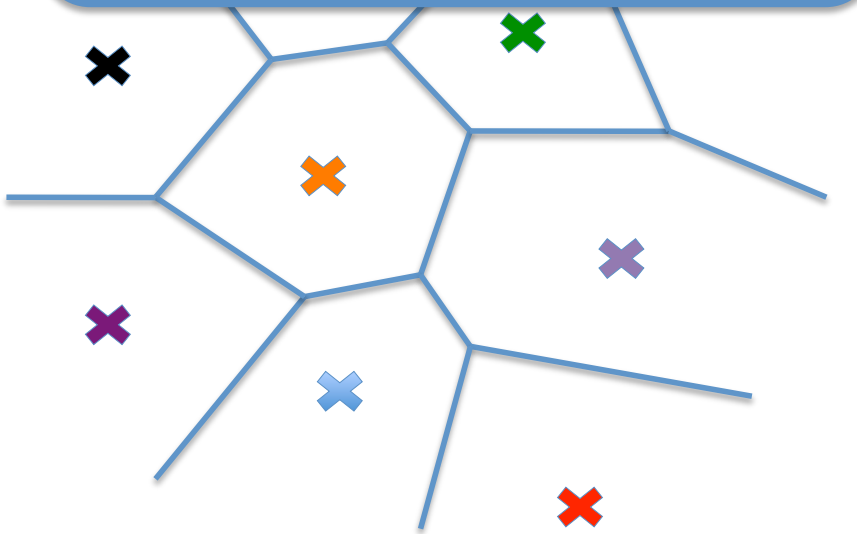
Learning Distributed Representations

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

Local vs. Distributed Representations

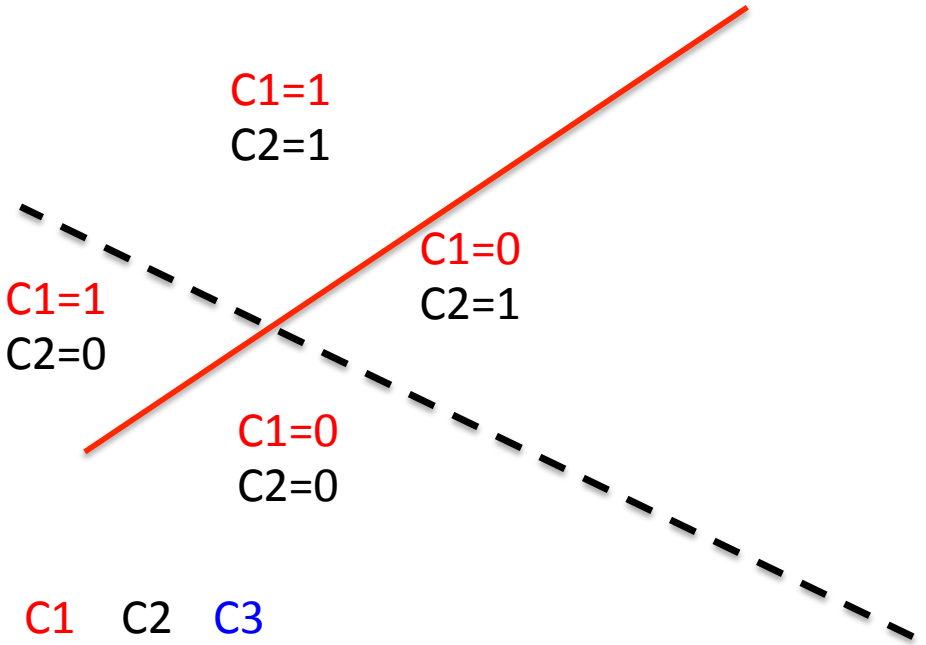
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

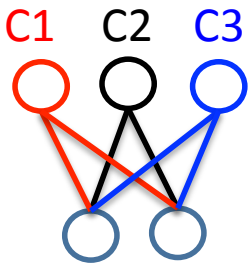
- Parameters for each region.
- # of regions is linear with # of parameters.



Learned prototypes

- RBMs, Factor models, PCA, Sparse Coding, Deep models



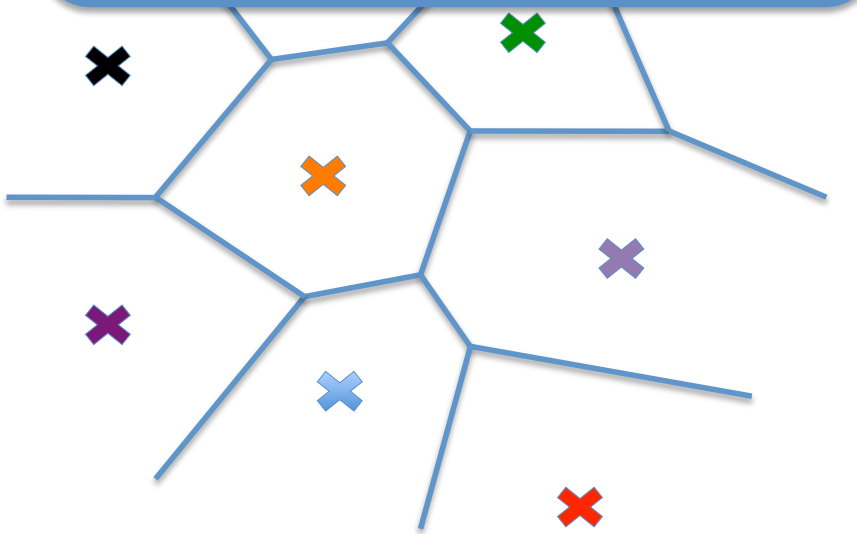


(Bengio, 2009, Foundations and Trends in Machine Learning)

Local vs. Distributed Representations

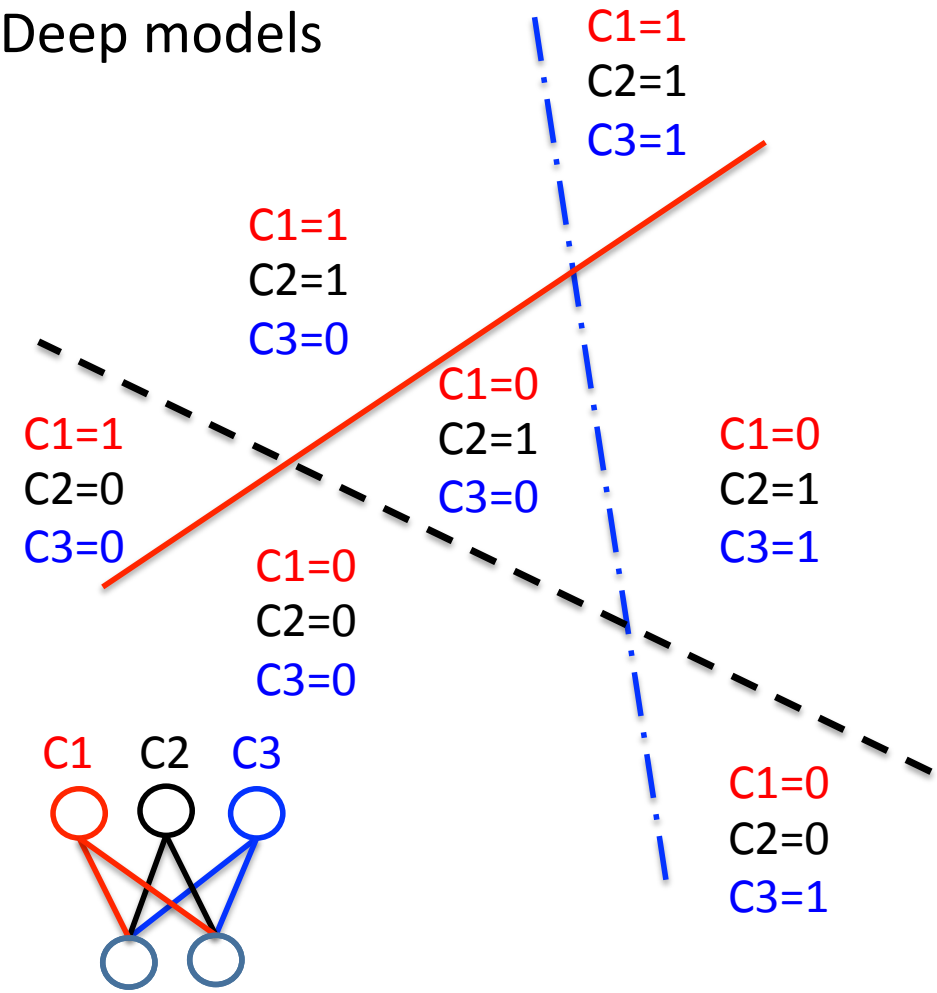
• Clustering, Nearest Neighbors, RBF SVM, local density estimators

• Parameters for each region.
• # of regions is linear with # of parameters.



Learned prototypes

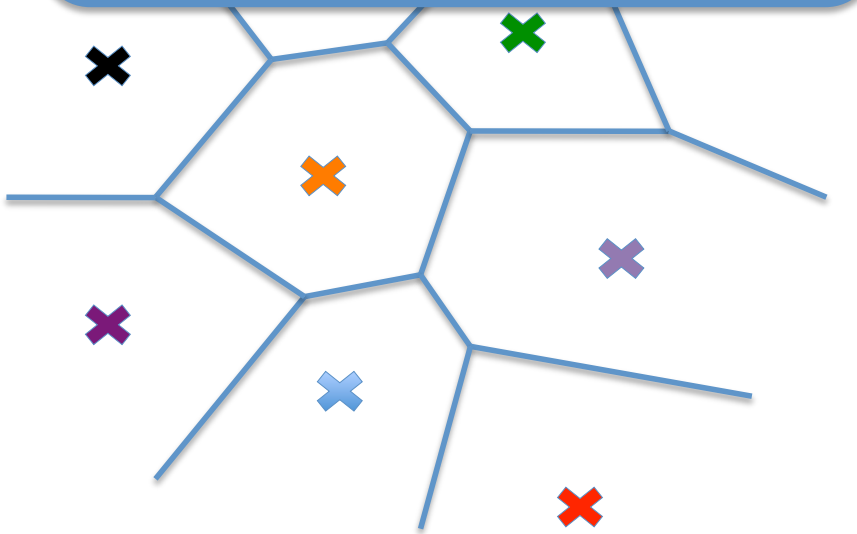
• RBMs, Factor models, PCA, Sparse Coding, Deep models



Local vs. Distributed Representations

• Clustering, Nearest Neighbors, RBF SVM, local density estimators

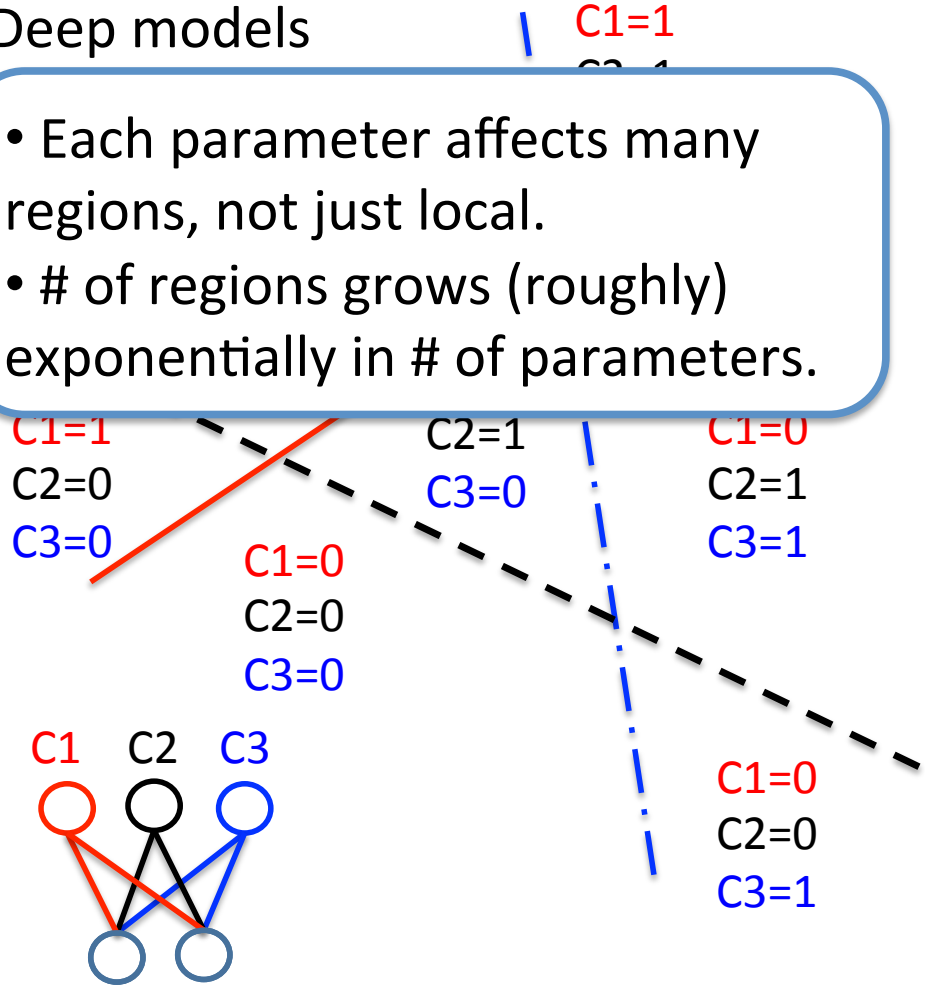
• Parameters for each region.
• # of regions is linear with # of parameters.



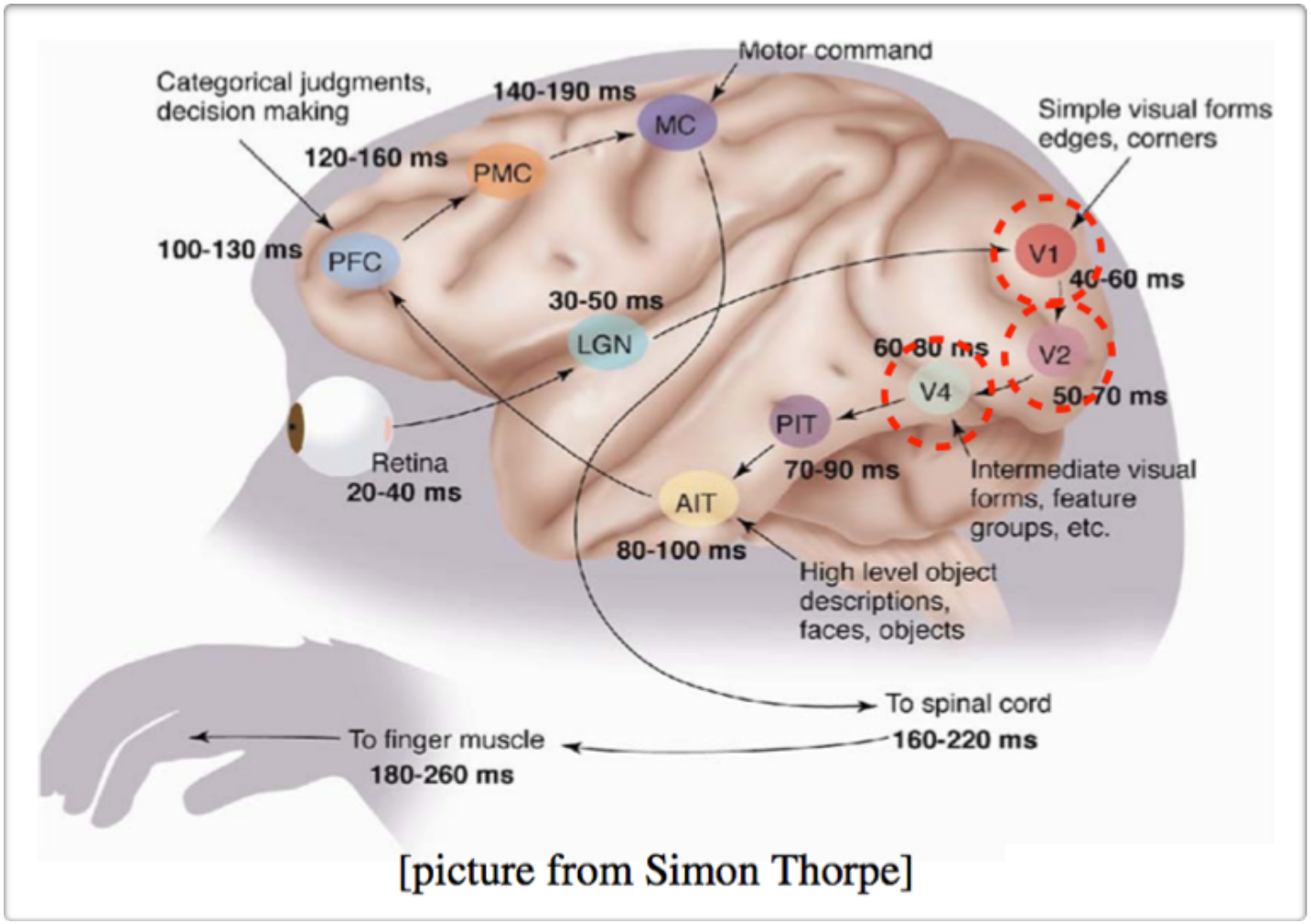
Learned prototypes

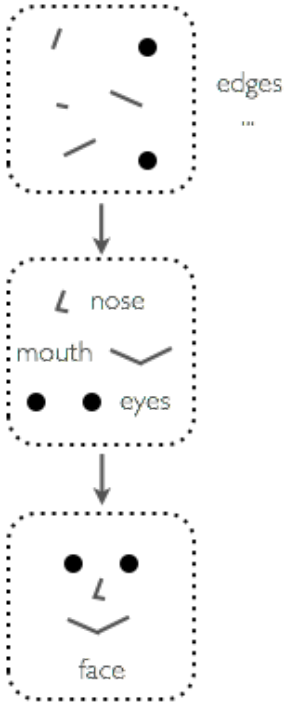
• RBMs, Factor models, PCA, Sparse Coding, Deep models

• Each parameter affects many regions, not just local.
• # of regions grows (roughly) exponentially in # of parameters.



Inspiration from Visual Cortex

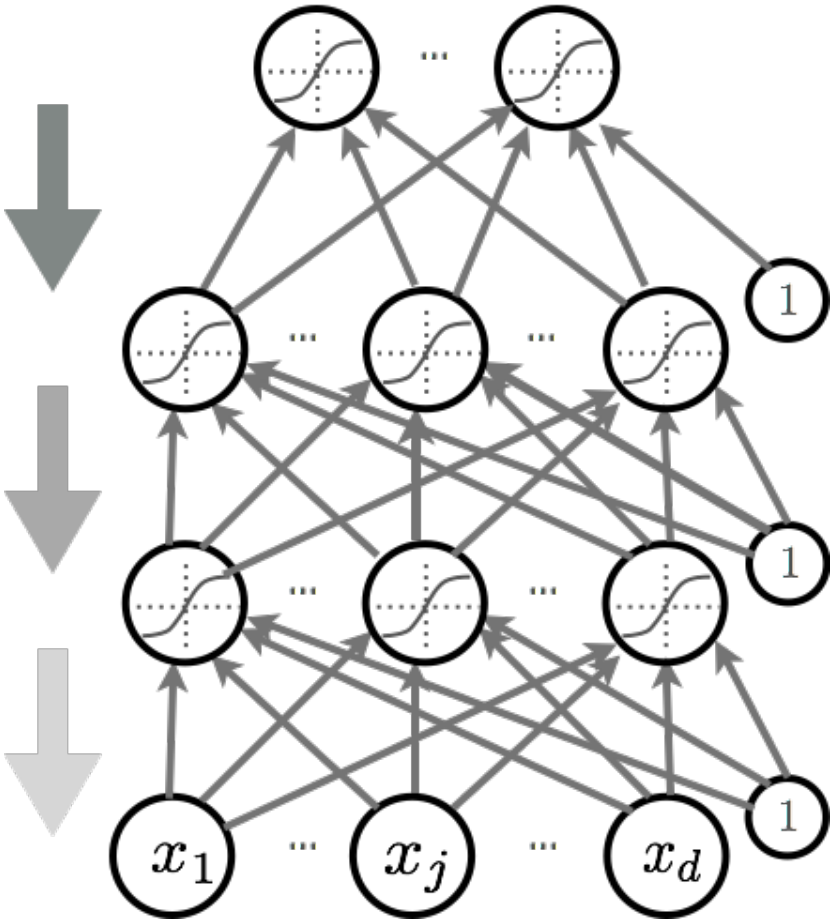




Why Training is Hard

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

- This is a well known problem in recurrent neural networks

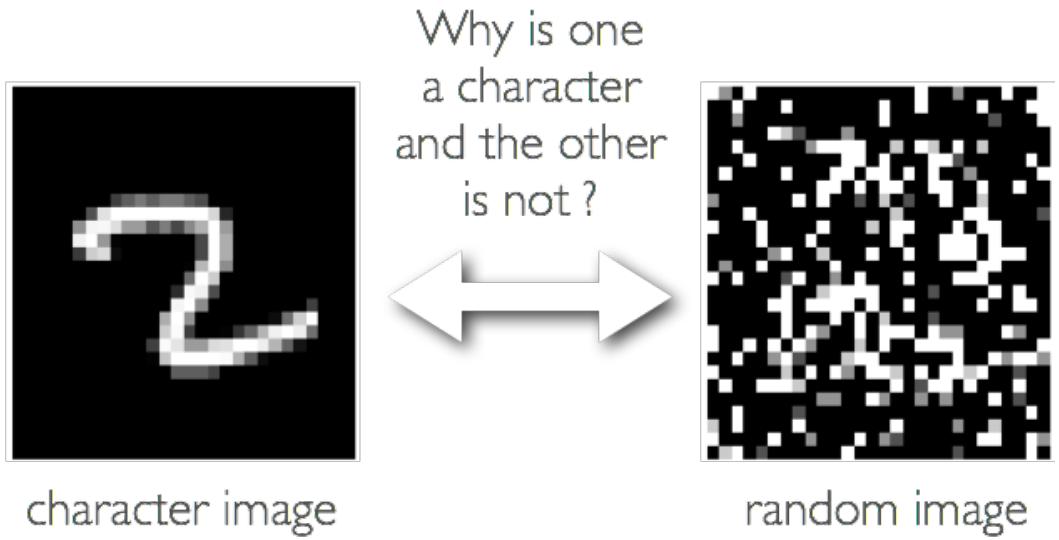


Why Training is Hard

- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

Unsupervised Pre-training

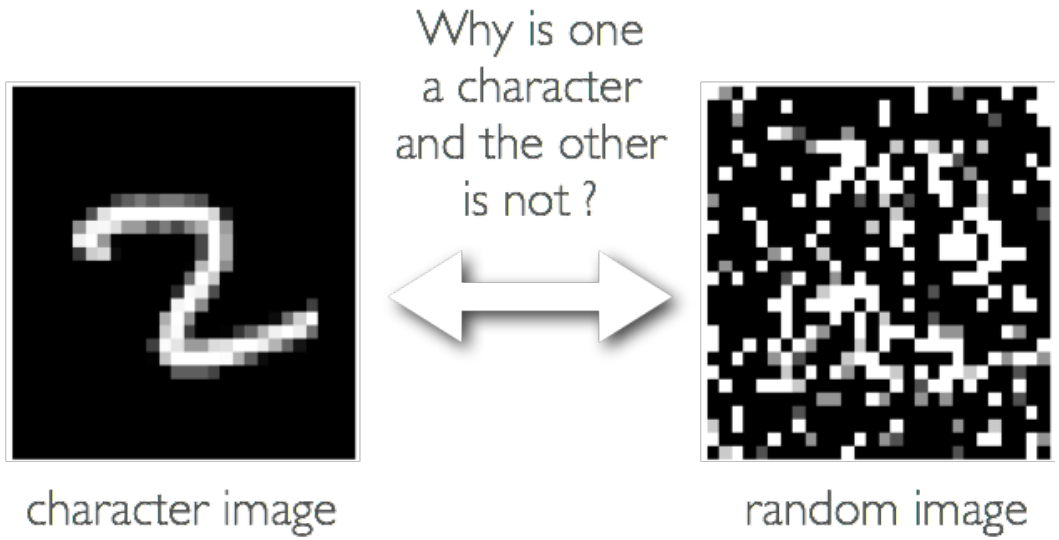
- Initialize hidden layers using unsupervised learning
  - Force network to represent latent structure of input distribution



  - Encourage hidden layers to encode that structure

Unsupervised Pre-training

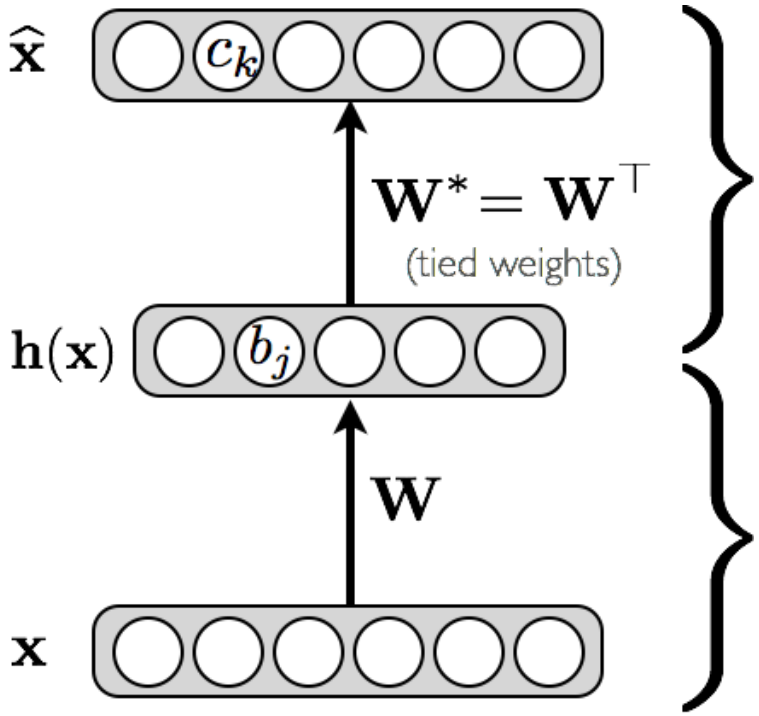
- Initialize hidden layers using unsupervised learning
  - This is a harder task than supervised learning (classification)



  - Hence we expect less overfitting

Autoencoders: Preview

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



Decoder

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

For binary units

Encoder

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

Autoencoders: Preview

- Loss function 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 error function $f(\mathbf{x}) \equiv \widehat{\mathbf{x}}$

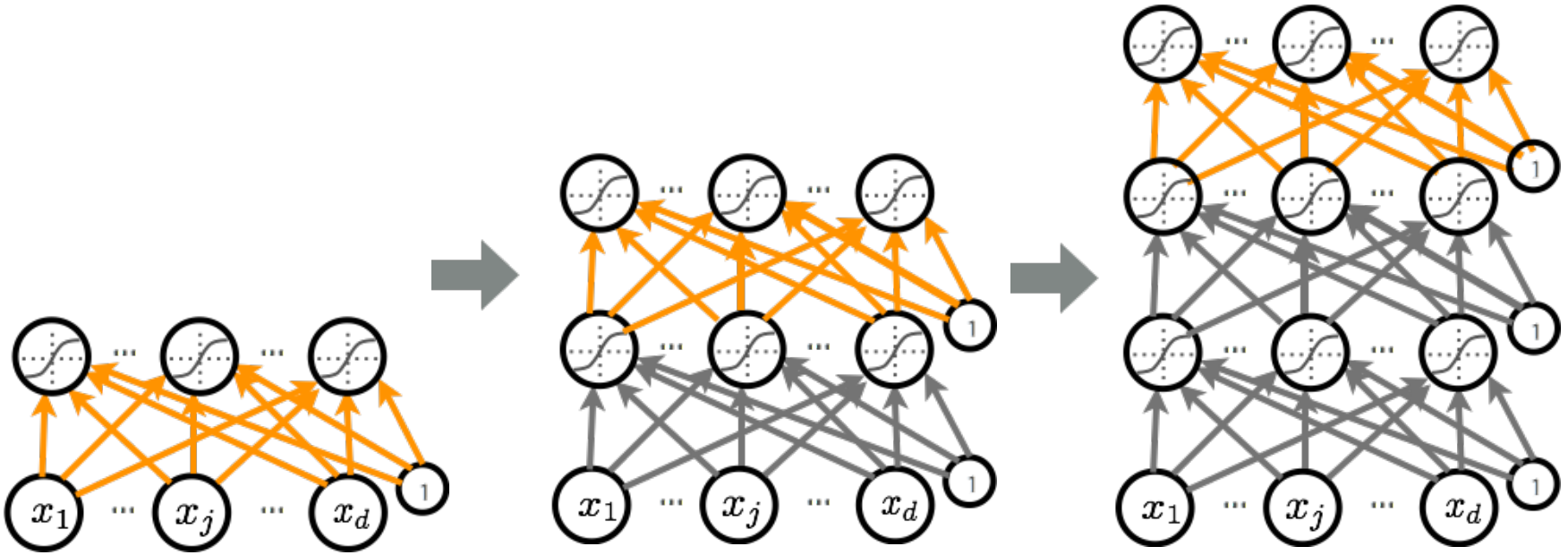
- Loss function for real-valued inputs

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

  - sum of squared differences
  - we use a linear activation function at the output

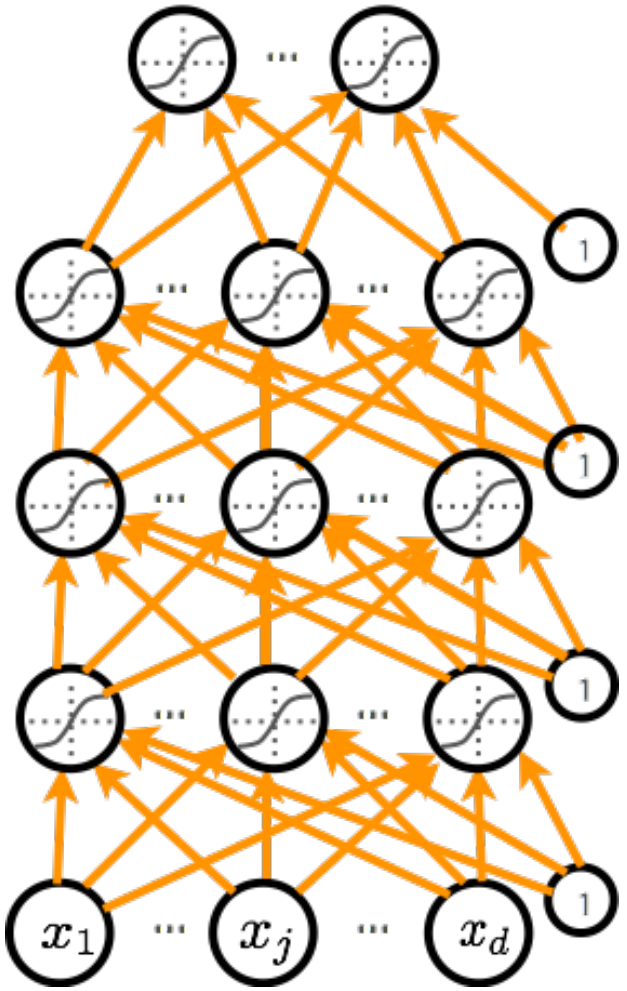
Pre-training

- We will use a greedy, layer-wise procedure
  - Train one layer at a time with unsupervised criterion
  - Fix the parameters of previous hidden layers
  - Previous layers can be viewed as feature extraction



Fine-tuning

- Once all layers are pre-trained
  - add output layer
  - train the whole network using supervised learning
- We call this last phase fine-tuning
  - all parameters are "tuned" for the supervised task at hand
  - representation is adjusted to be more discriminative



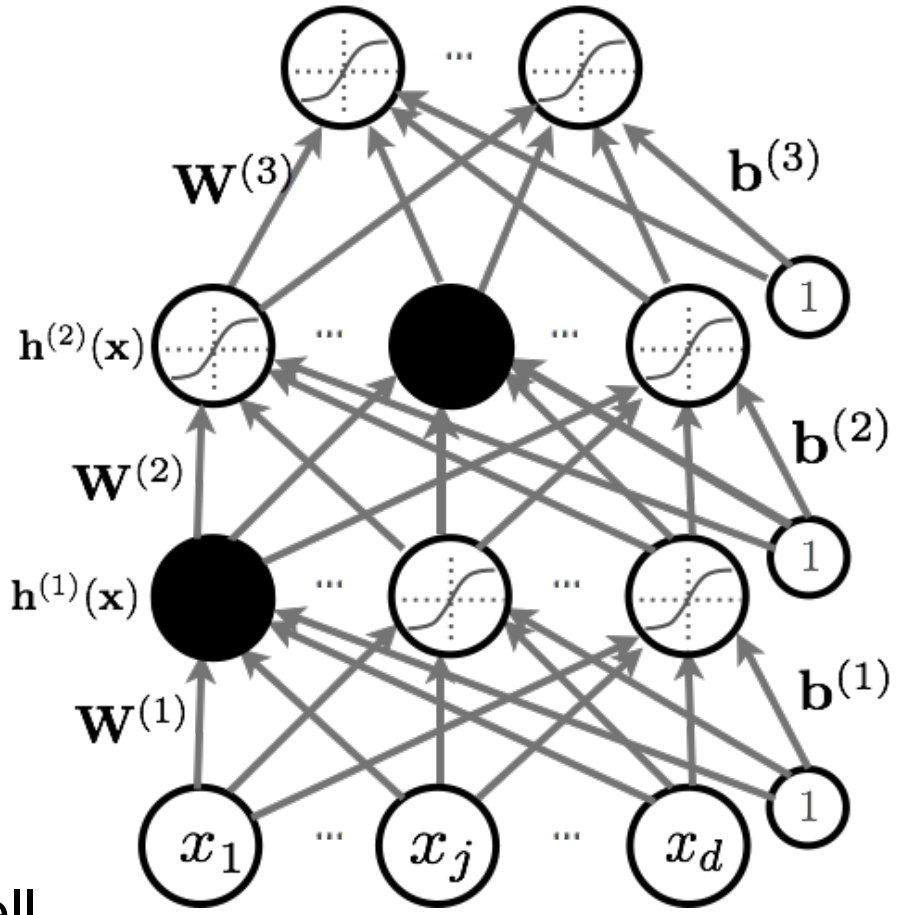
Why Training is Hard

- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

Dropout

• Key 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

- Use random binary masks $m^{(k)}$
  - layer pre-activation for k>0

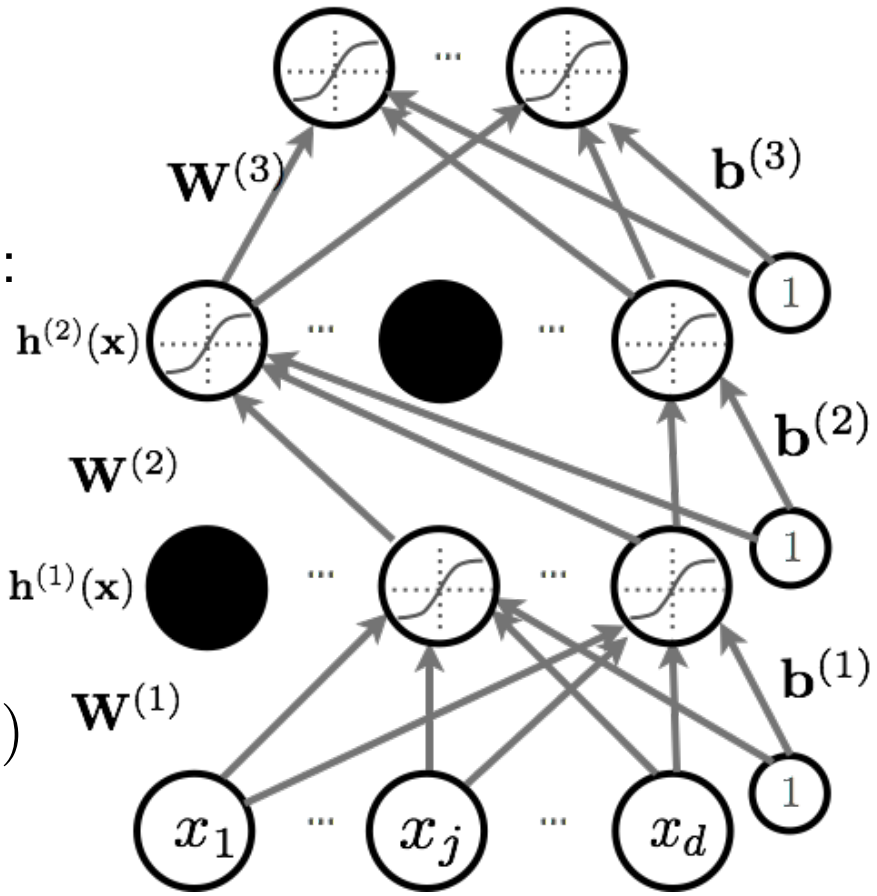
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

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

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

  - Output activation (k=L+1)

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



Dropout at Test Time

- 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: 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
- Ensemble: Can be viewed as a geometric average of exponential number of networks.

Why Training is Hard

- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- For many large-scale practical problems, you will need to use both: better optimization and better regularization!

Batch Normalization

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - could normalization 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

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

- 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)

- Use the global mean and 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
    - propagate all training data to that layer
    - compute and store the global mean and stddev of each unit
  - for early stopping, could use a running average