10417/10617
Intermediate Deep Learning: Fall2019

Russ Salakhutdinov

Machine Learning Department
rsalakhu@cs.cmu.edu

Neural Networks II

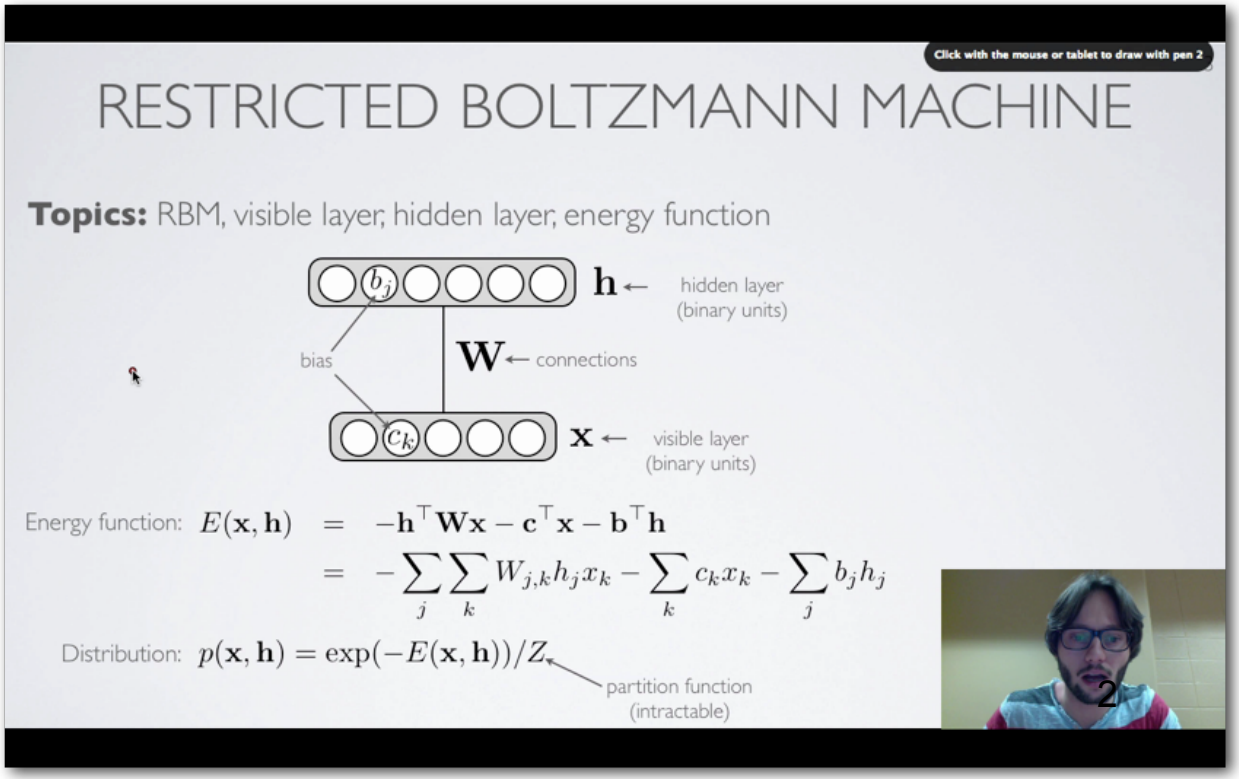
Neural Networks Online Course

• **Disclaimer**: Much 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.

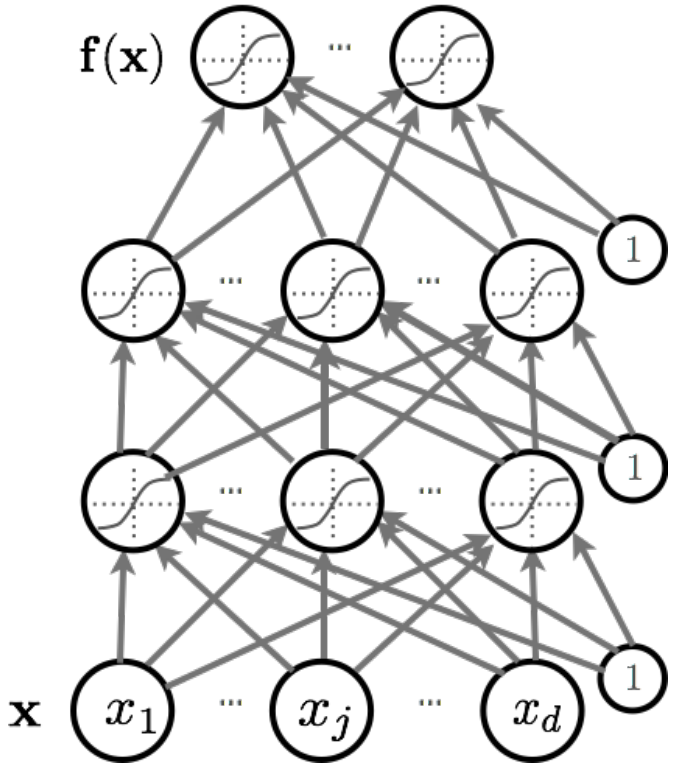
• We will use his material for some of the other lectures.

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



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



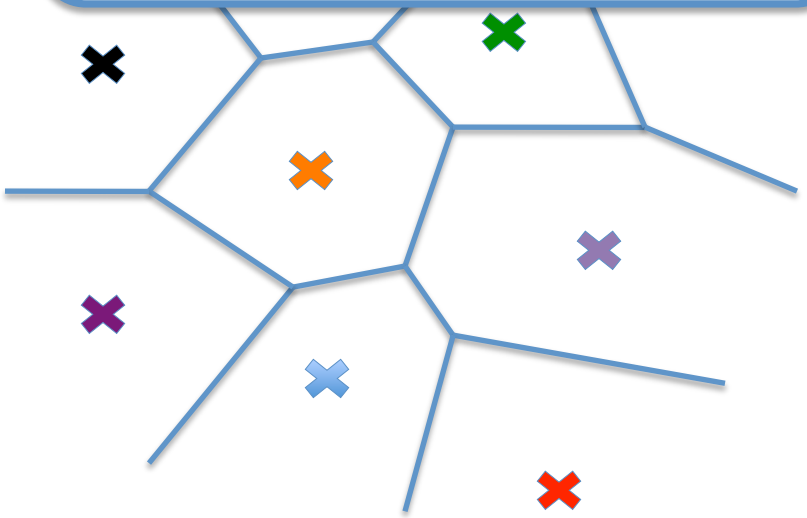
Learning Distributed Representations

- 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

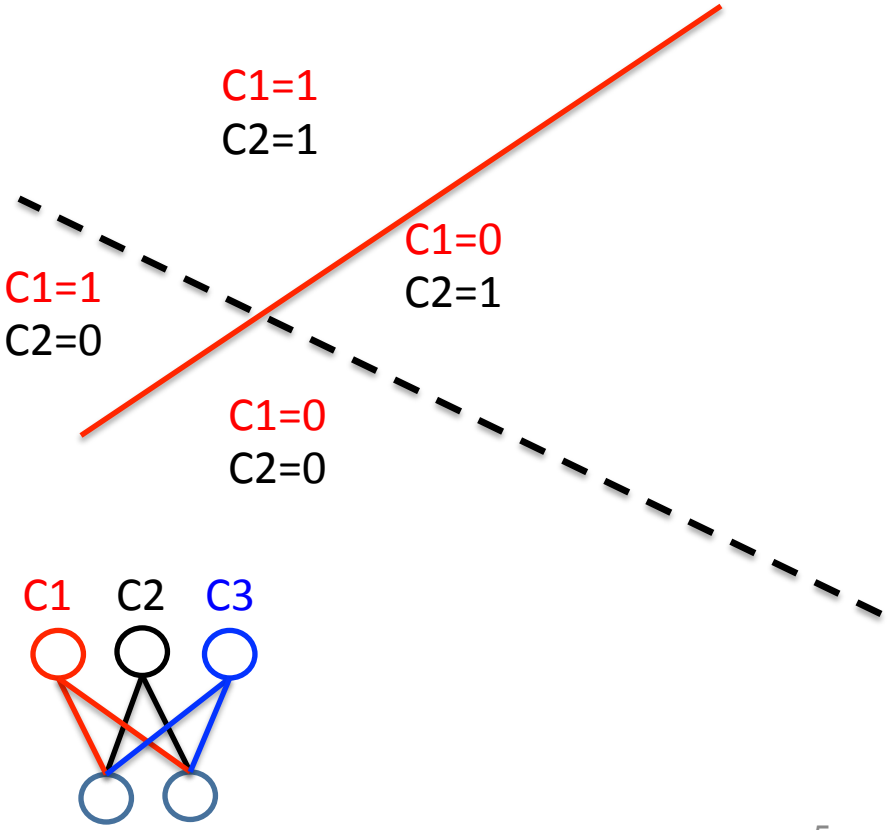
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

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



Learned prototypes

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

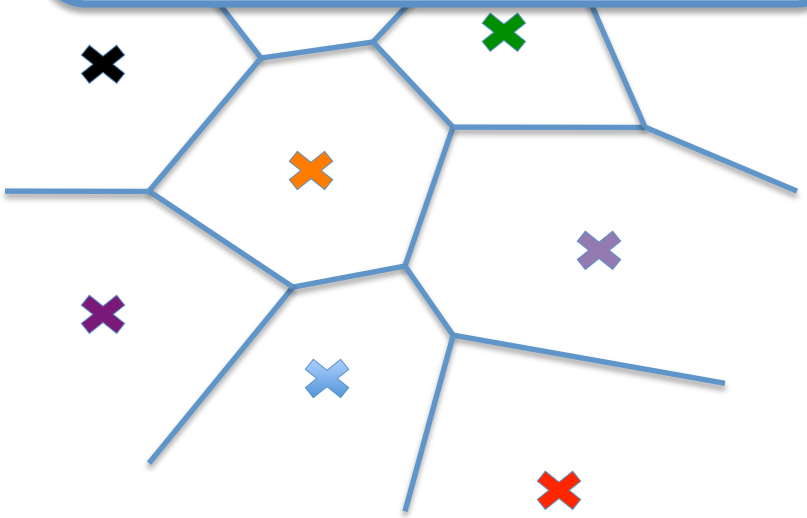


Bengio, 2009, Foundations and Trends in Machine Learning

Local vs. Distributed Representations

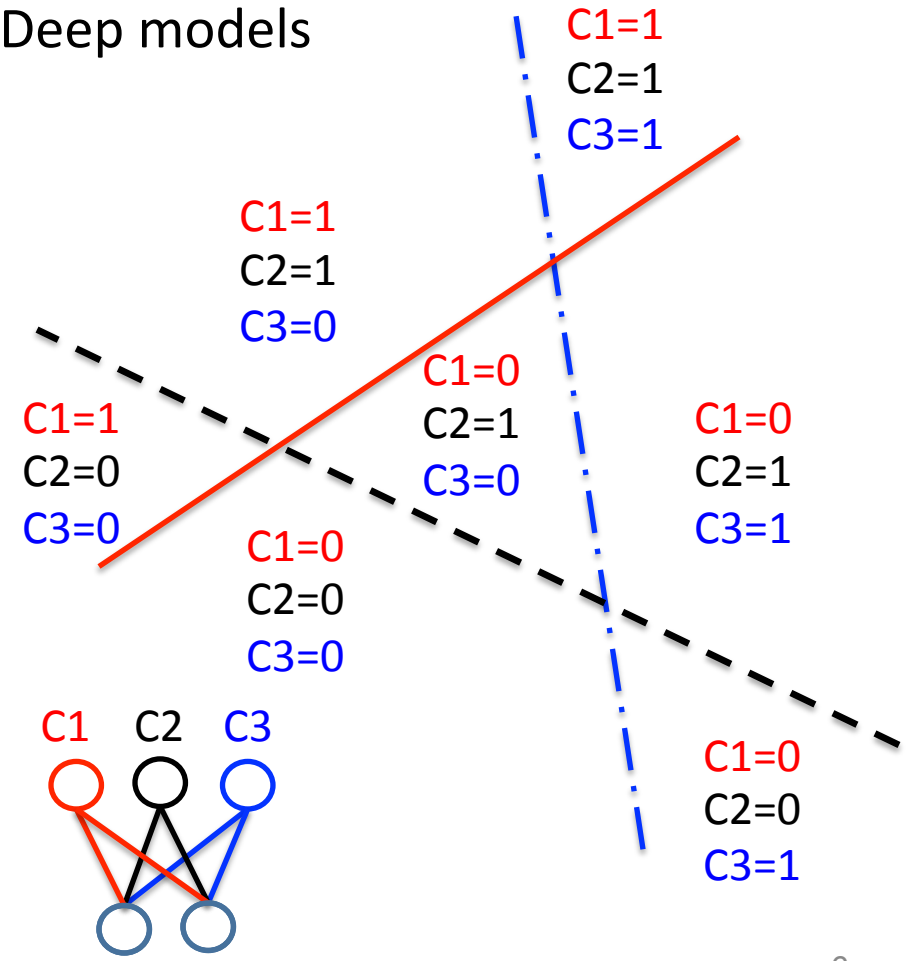
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

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



Learned prototypes

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

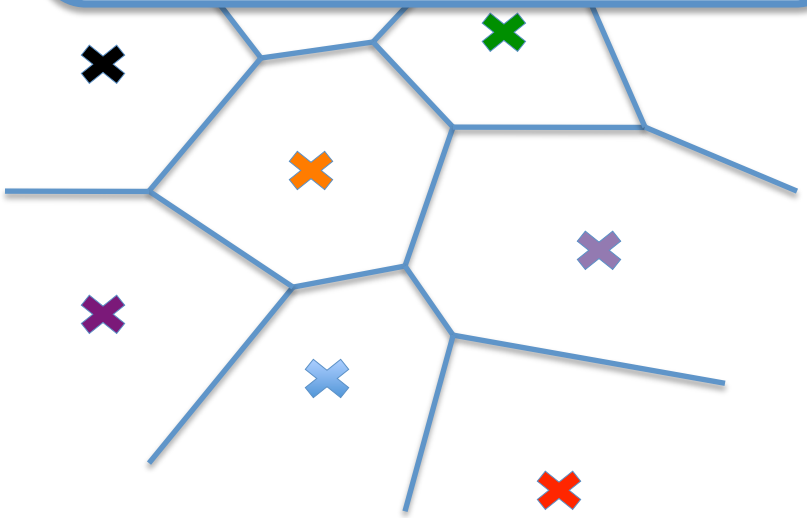


Bengio, 2009, Foundations and Trends in Machine Learning

Local vs. Distributed Representations

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

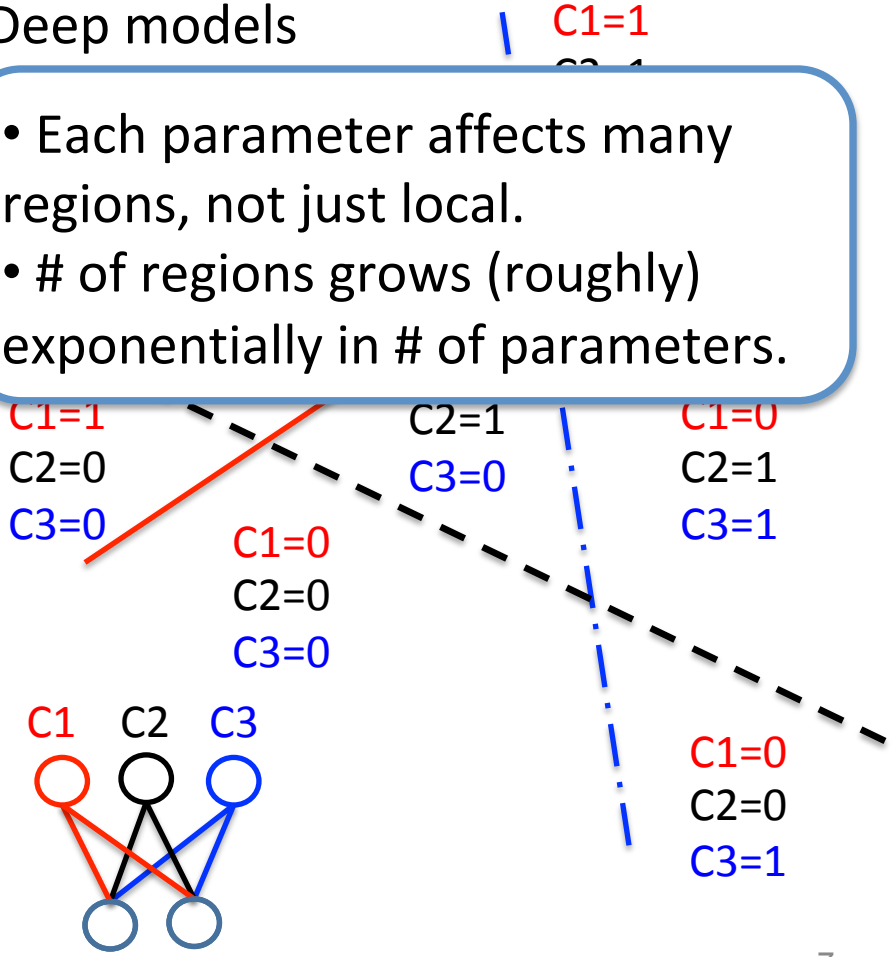
- Parameters for each region.
- # of regions is linear with # of parameters.



Learned prototypes

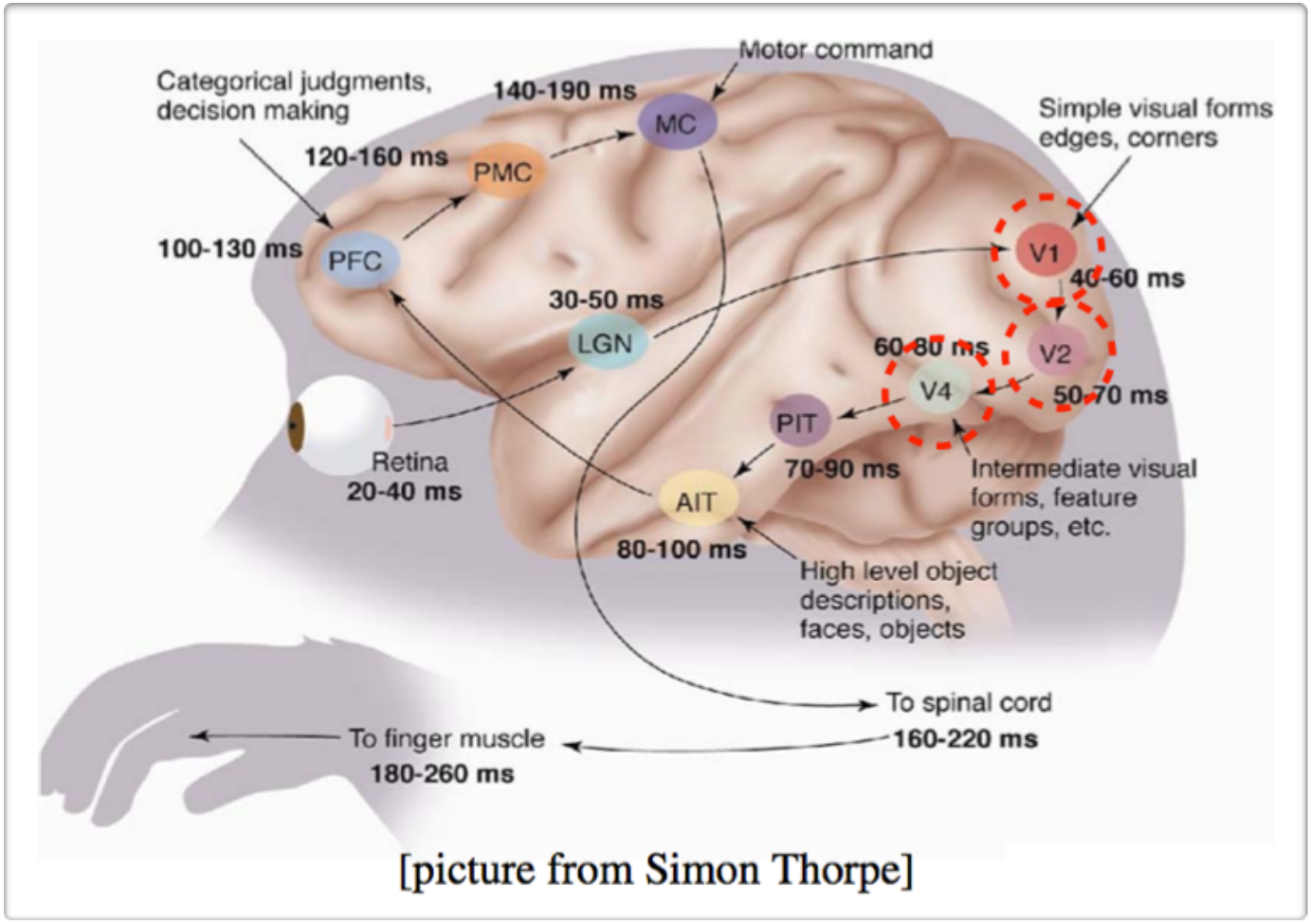
- RBMs, Factor models, PCA, Sparse Coding, Deep models

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

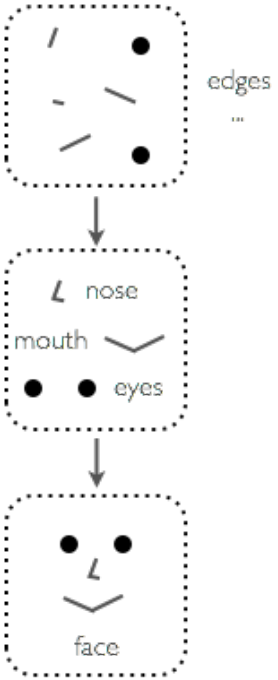


Bengio, 2009, Foundations and Trends in Machine Learning

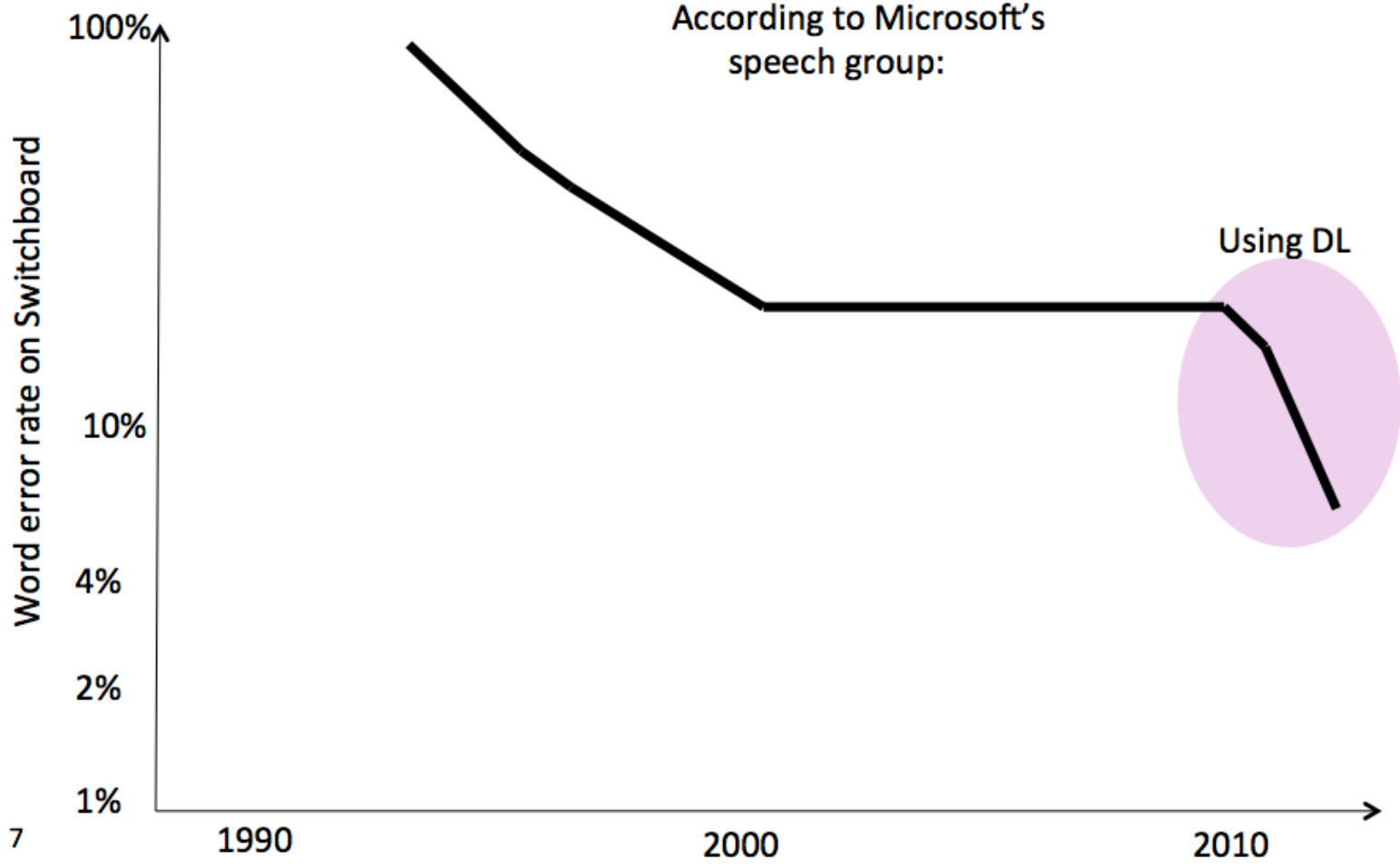
Inspiration from Visual Cortex



[picture from Simon Thorpe]



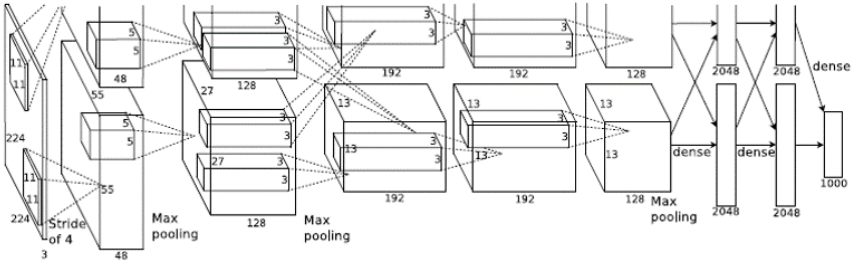
Success Story: Speech Recognition



7

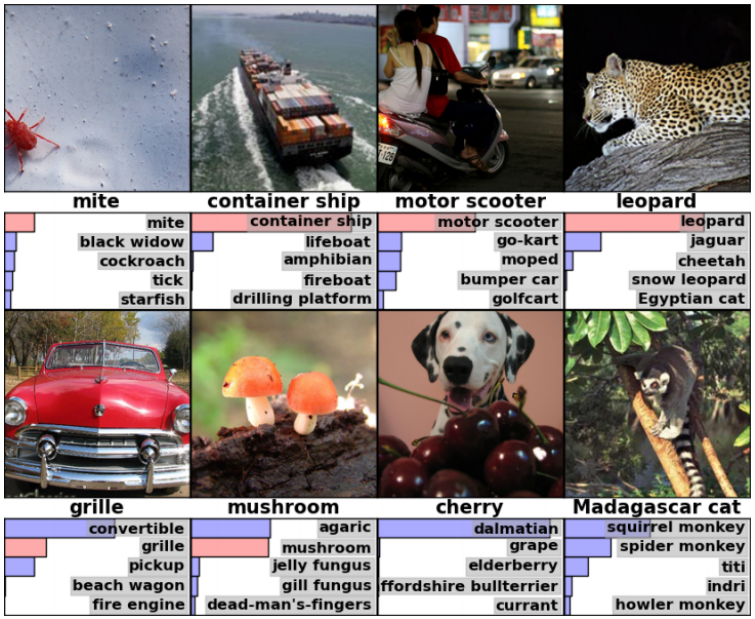
Success Story: Image Recognition

- Deep Convolutional Nets for Vision (Supervised)



IMAGENET

1.2 million training images
1000 classes

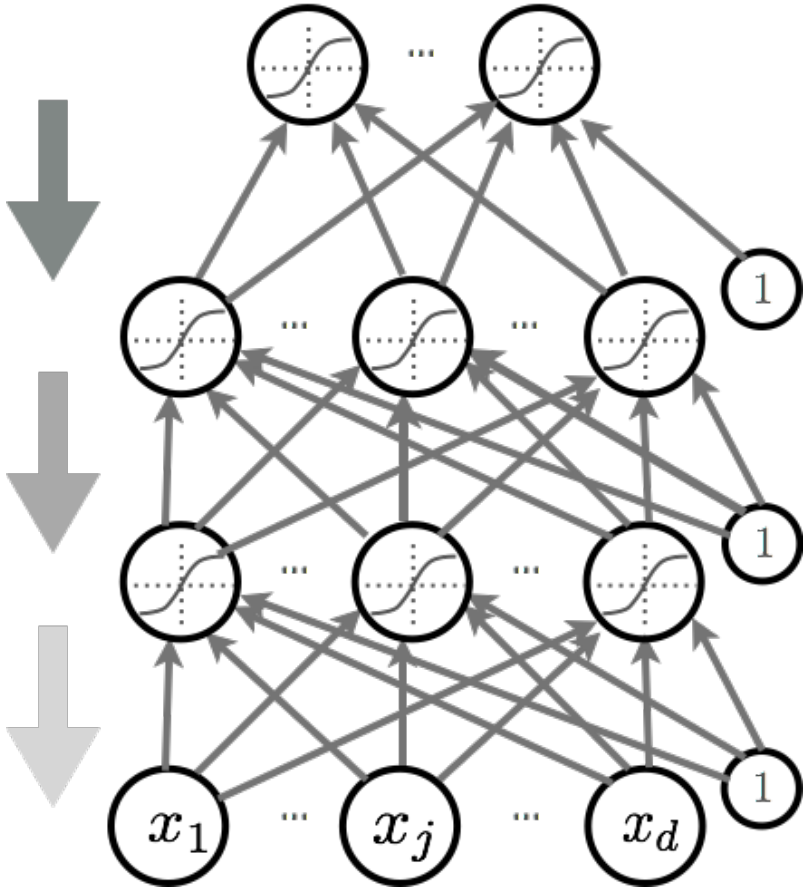


Why Training is Hard

• First hypothesis: Hard optimization problem (underfitting)

- vanishing gradient problem
- saturated units block gradient propagation

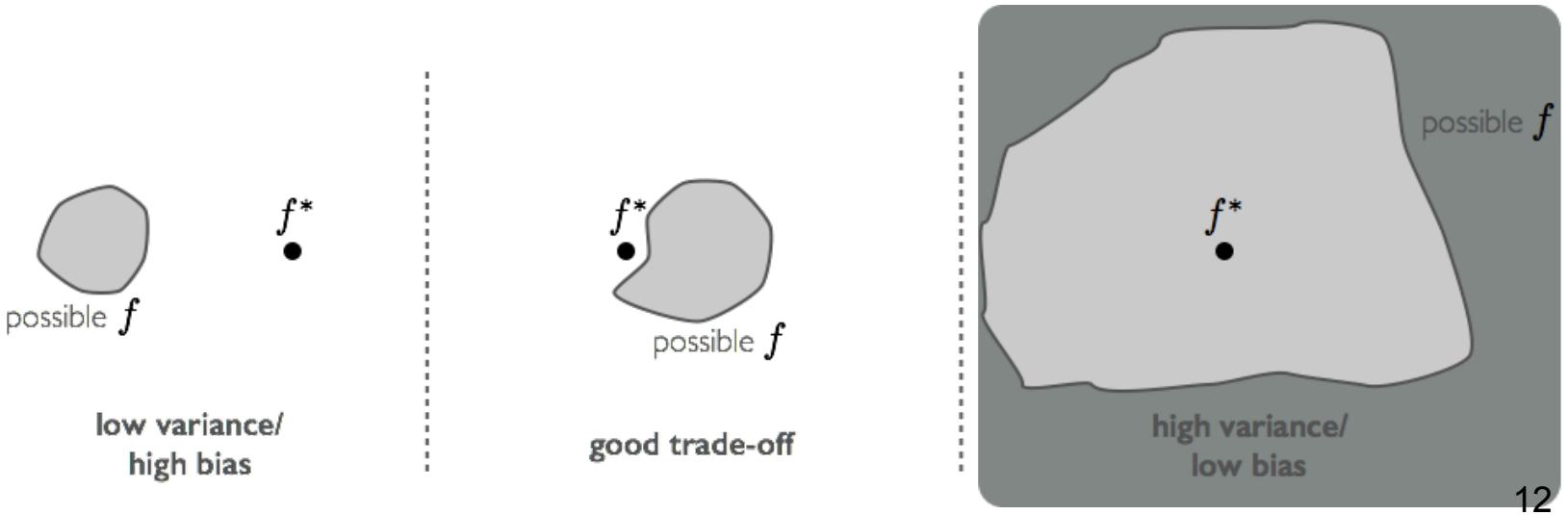
•This is a well known problem in recurrent neural networks



Why Training is Hard

- 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



Why Training is Hard

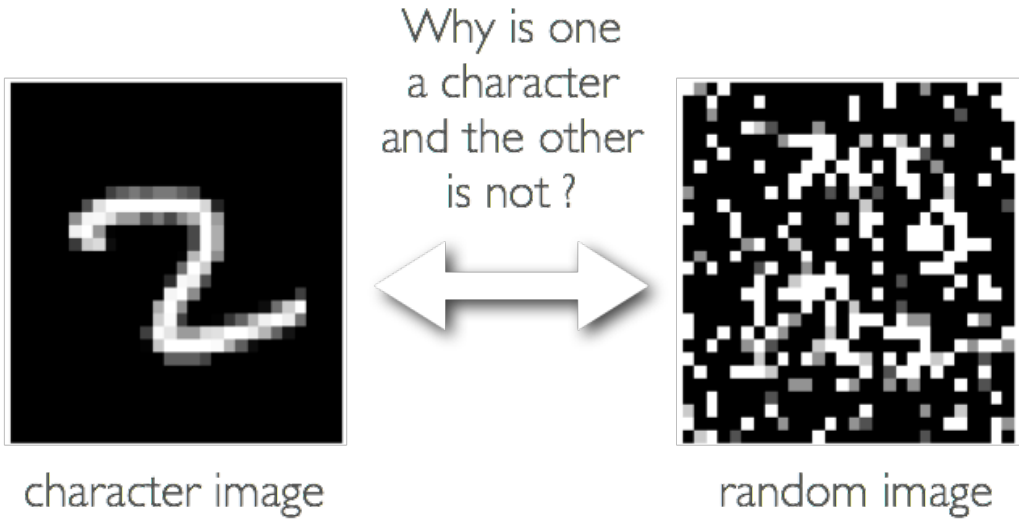
- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.

- 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

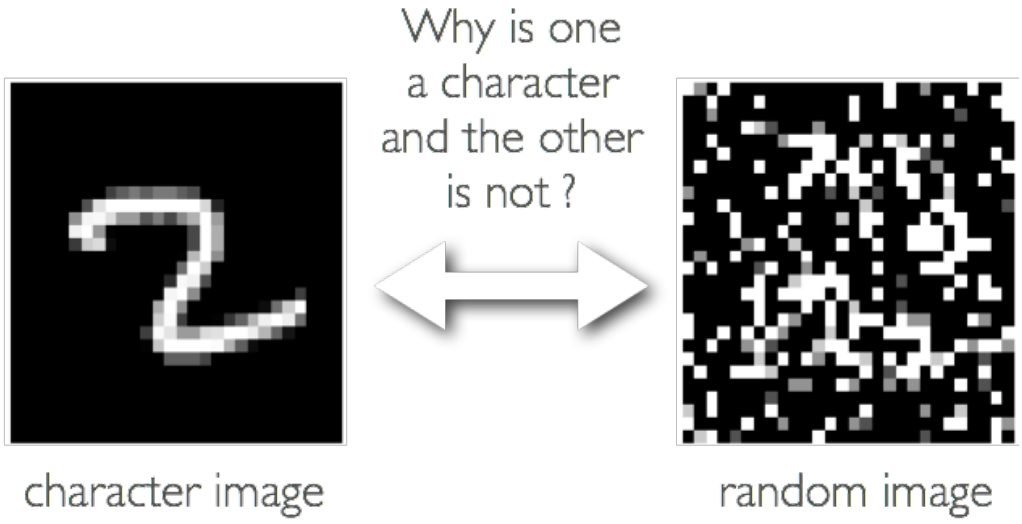
- Initialize hidden layers using unsupervised learning
  - Force network to represent latent structure of input distribution



  - Encourage hidden layers to encode that structure

Unsupervised Pre-training

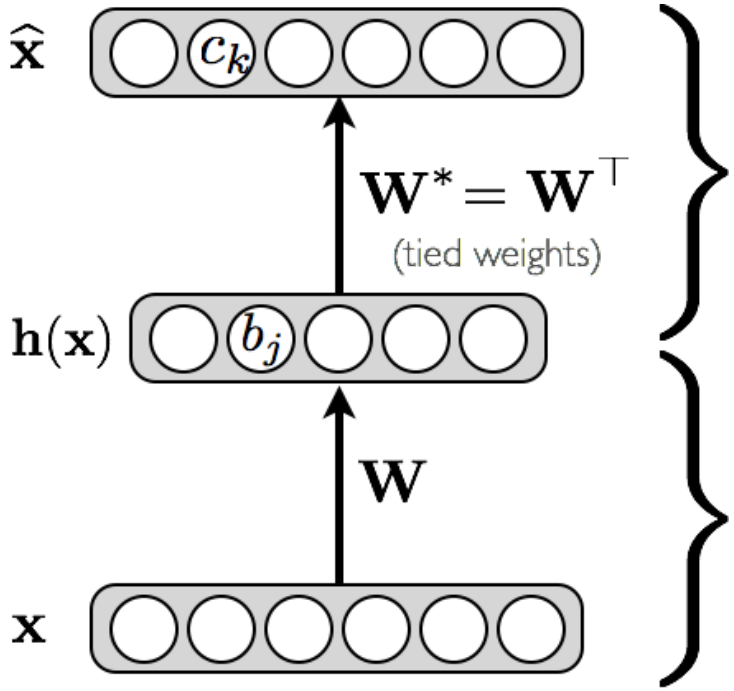
- Initialize hidden layers using unsupervised learning
  - This is a harder task than supervised learning (classification)



  - Hence we expect less overfitting

Autoencoders: Preview

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



Decoder

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

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

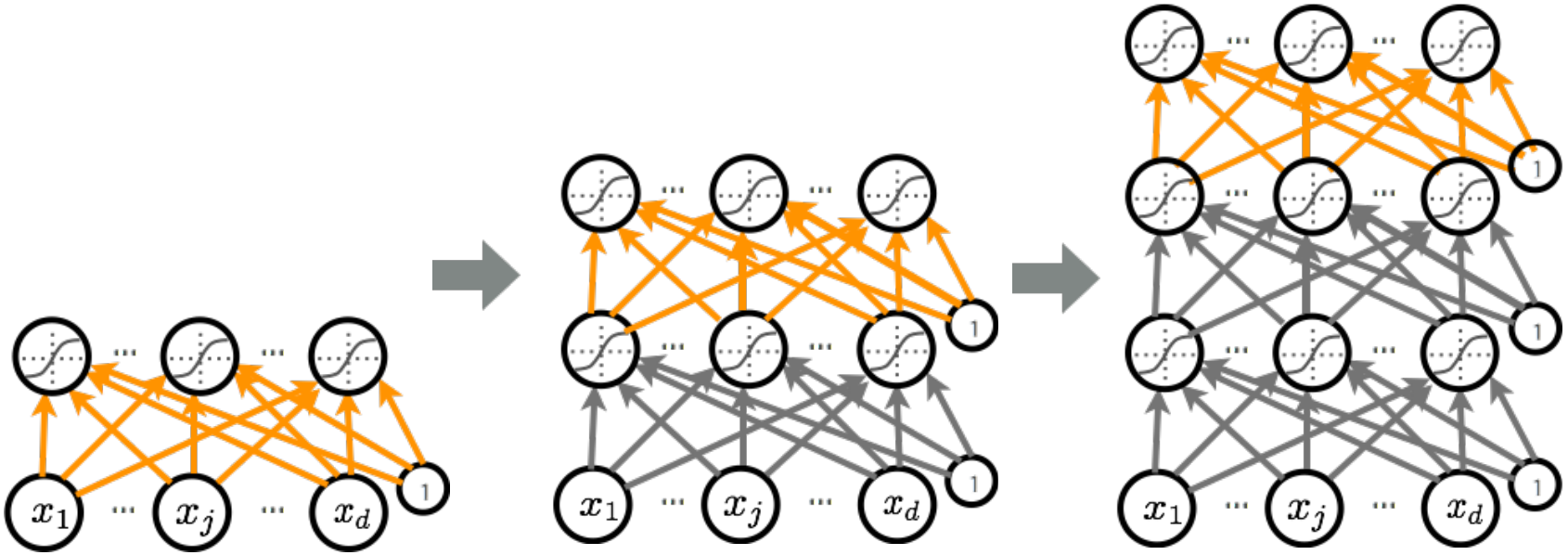
- Loss function for real-valued inputs

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

  - 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



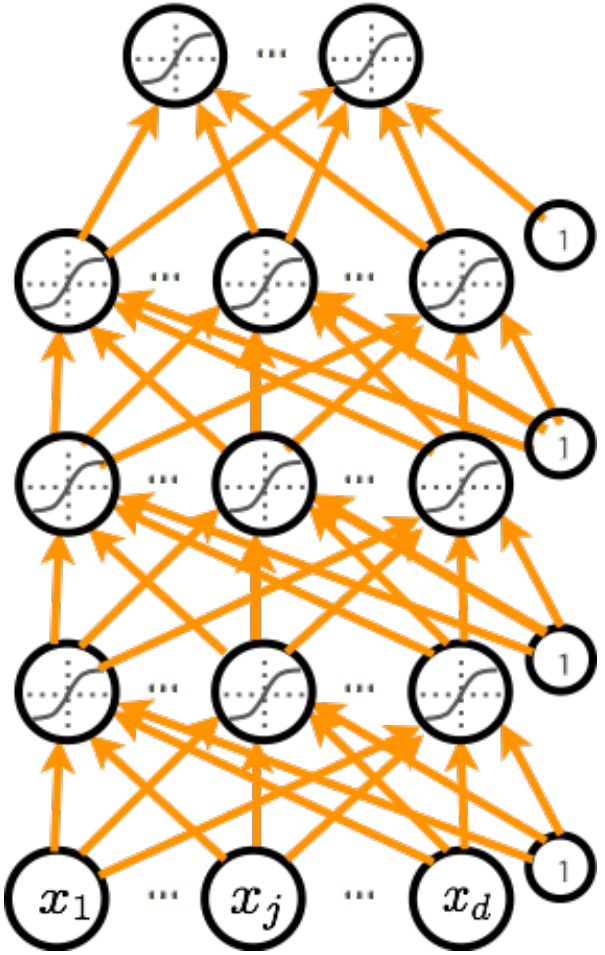
Pre-training

- Unsupervsed 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 ...

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

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 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



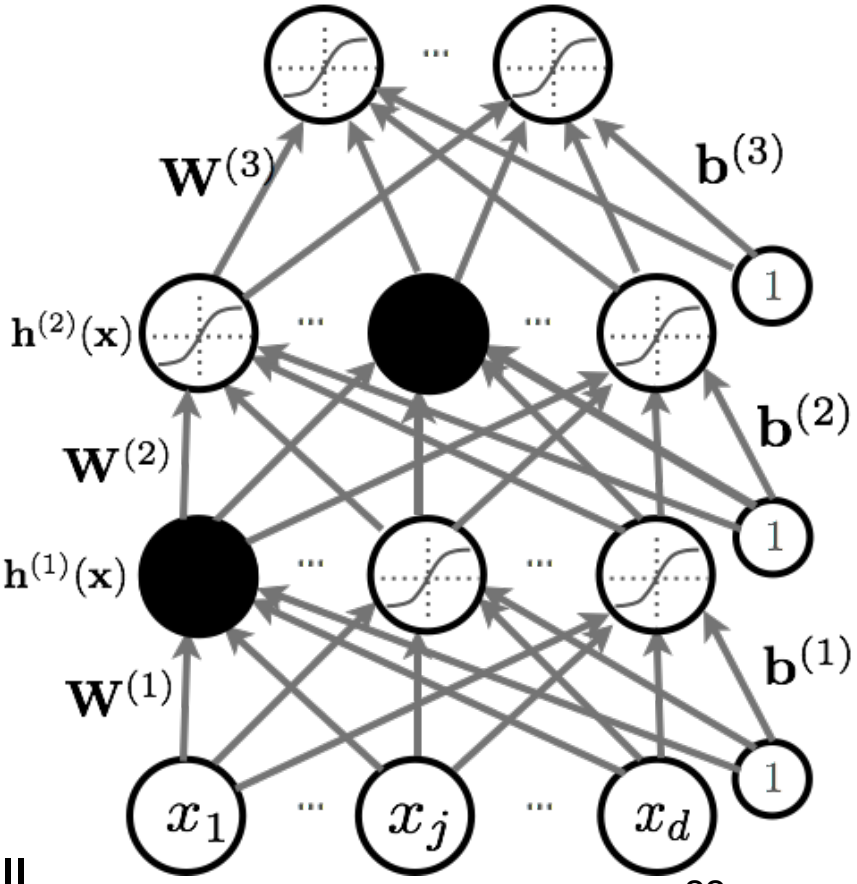
Why Training is Hard

- 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

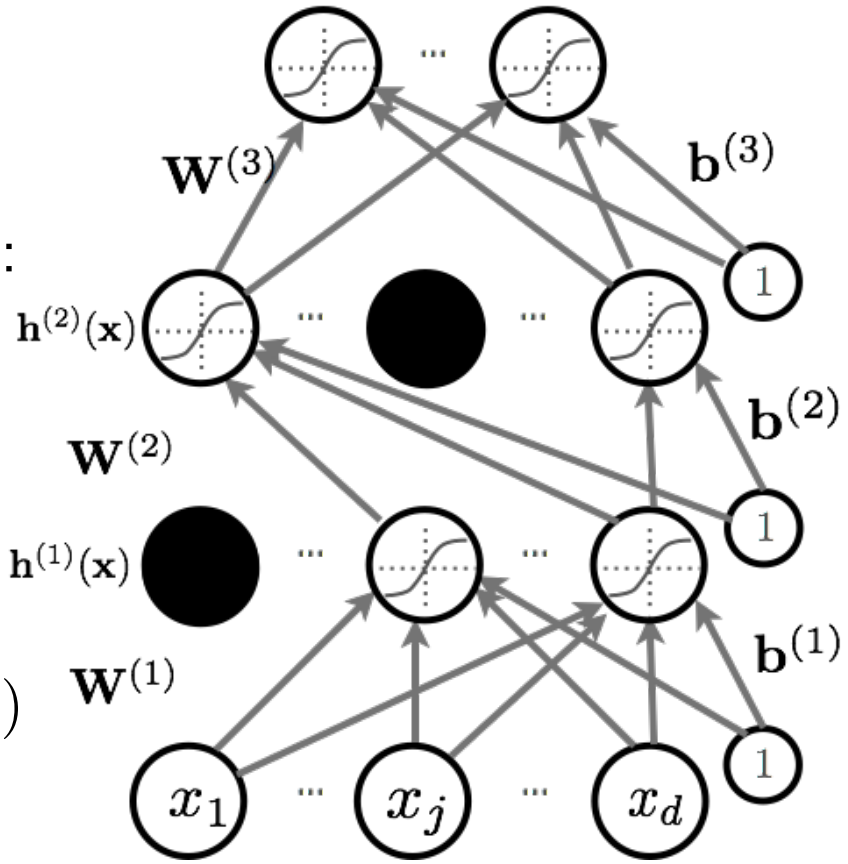
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - 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)

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



Backpropagation Algorithm

- Perform forward propagation
- Compute output gradient (before activation):

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

Includes the mask $m^{(k-1)}$

- For k=L+1 to 1
  - Compute gradients w.r.t. the hidden layer parameters:

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

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

  - Compute gradients w.r.t. the hidden layer below:

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

  - Compute gradients w.r.t. the hidden layer below (before activation):

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

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

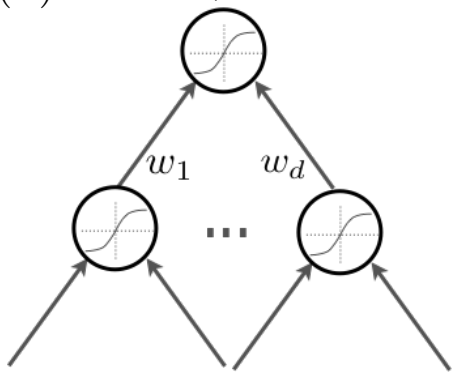
- First hypothesis (underfitting): better optimize
  - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
  - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
  - Unsupervised pre-training
  - Stochastic drop-out training
- 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

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



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)

- 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

Batch Normalization

- How to take into account the normalization in backdrop?
  - derivative w.r.t. x_i depends on the partial derivative of both: the mean and stddev
  - must also update γ and β
- Why 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

Optimization Tricks

- SGD with momentum, batch-normalization, and dropout usually works very well

- Pick learning rate by running on a subset of the data
  - Start with large learning rate & divide by 2 until loss does not diverge
  - Decay learning rate by a factor of ~100 or more by the end of training

- Use ReLU nonlinearity

- Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Improving Generalization

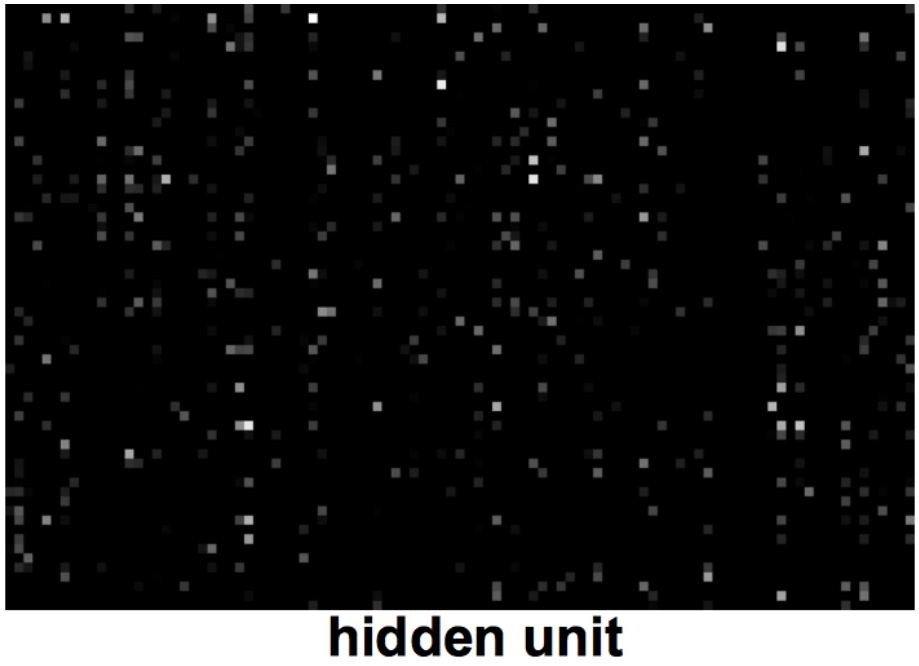
- Weight sharing (greatly reduce the number of parameters)
- Dropout
- Weight decay (L2, L1)
- Sparsity in the hidden units

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Visualization

- Check gradients numerically by finite differences

- Visualize features (features need to be uncorrelated) and have high variance



- Good training: hidden units are sparse across samples

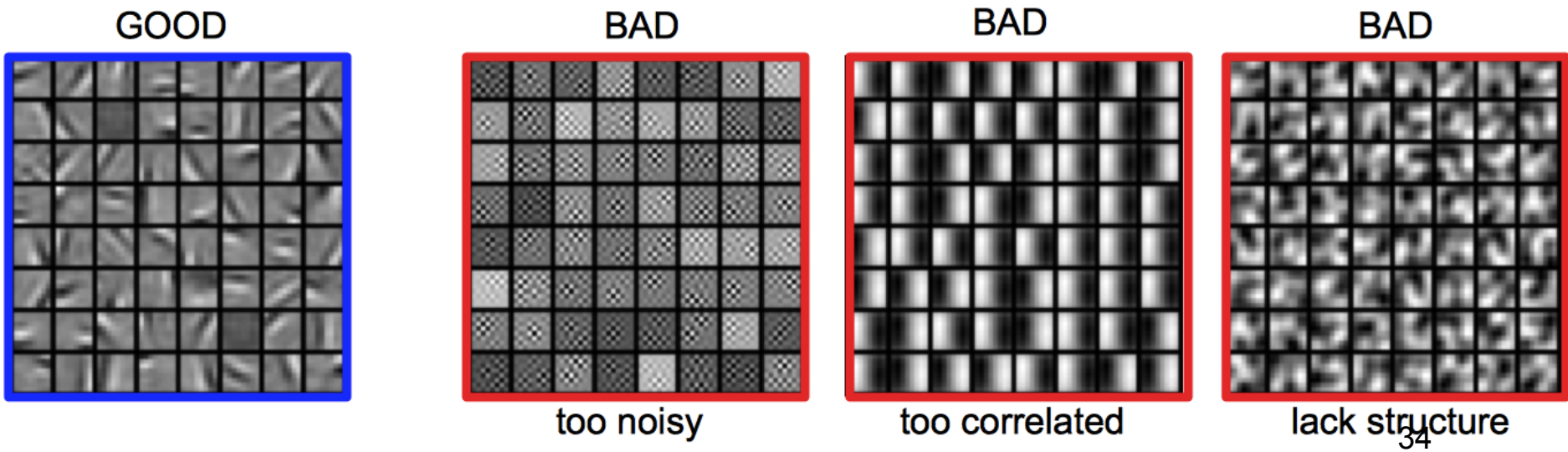
[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Visualization

- Check gradients numerically by finite differences

- Visualize features (features need to be uncorrelated) and have high variance

- Visualize parameters: learned features should exhibit structure and should be uncorrelated and are uncorrelated

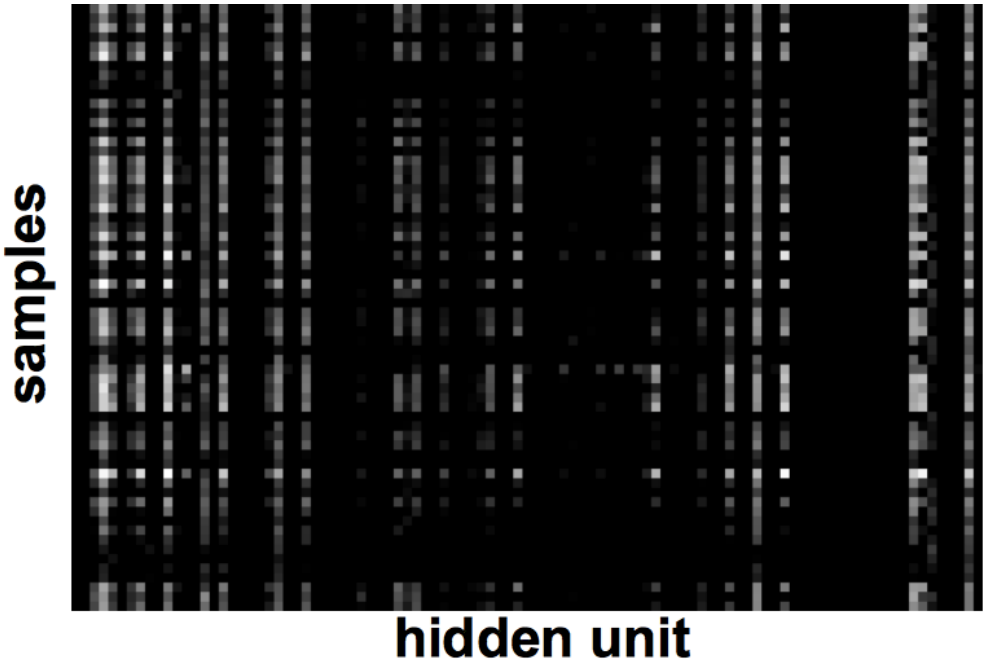


[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Visualization

- Check gradients numerically by finite differences

- Visualize features (features need to be uncorrelated) and have high variance



- Bad training: many hidden units ignore the input and/or exhibit strong correlations

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

Visualization

- Check gradients numerically by finite differences

- Visualize features (features need to be uncorrelated) and have high variance

- Visualize parameters: learned features should exhibit structure and should be uncorrelated and are uncorrelated

- Measure error on both training and validation set

- Test on a small subset of the data and check the error $\rightarrow$ 0.

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

When it does not work

- Training diverges:
  - Learning rate may be too large → decrease learning rate
  - BPROP is buggy → numerical gradient checking
- Parameters collapse / loss is minimized but accuracy is low
  - Check loss function: Is it appropriate for the task you want to solve?
  - Does it have degenerate solutions?
- Network is underperforming
  - Compute flops and nr. params. → if too small, make net larger
  - Visualize hidden units/params → fix optimization
- Network is too slow
  - GPU,distrib. framework, make net smaller

[From Marc'Aurelio Ranzato, CVPR 2014 tutorial]

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

- 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

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

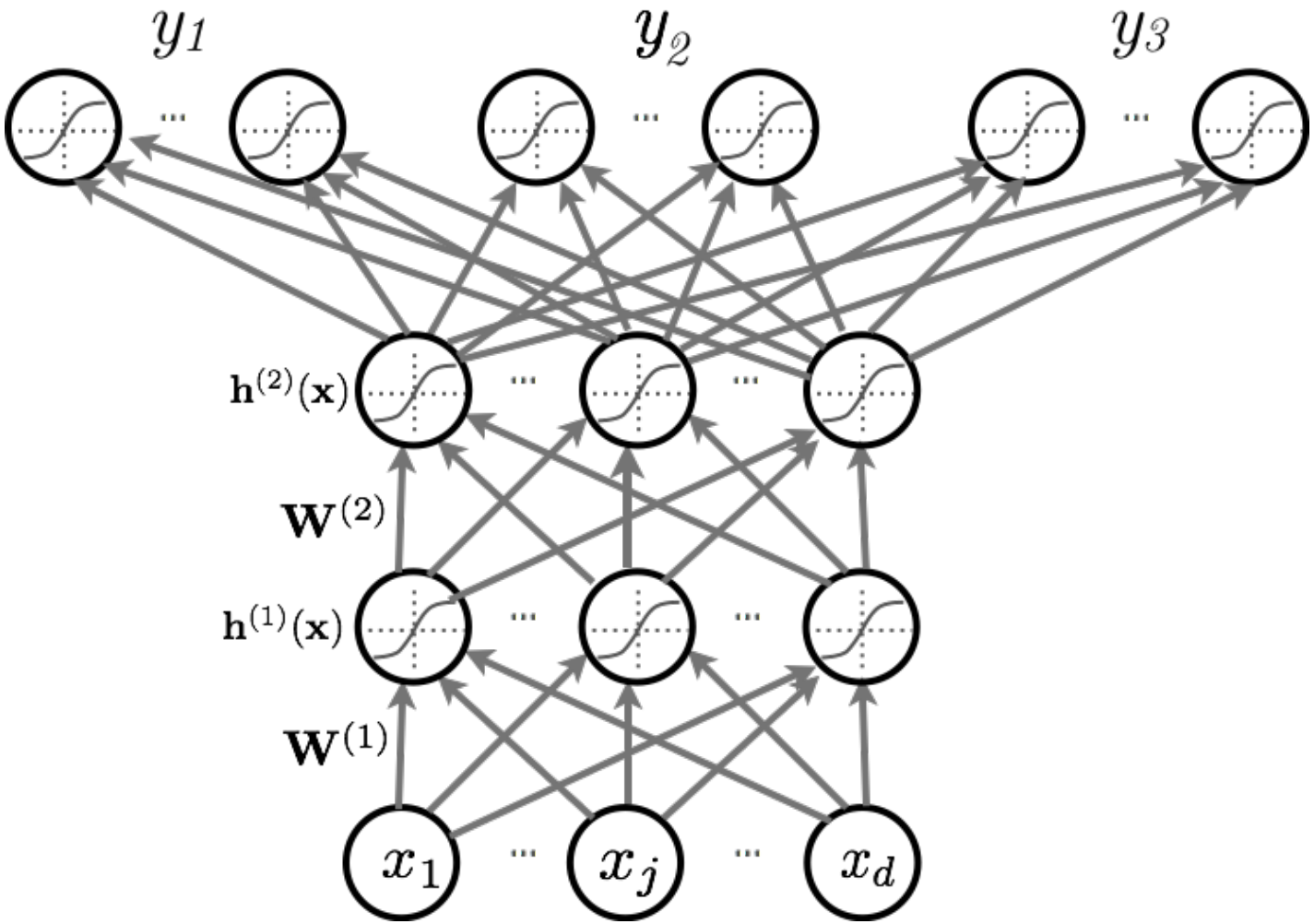
Multi-Task Learning

- Training time
  - Data:
    $\{\mathbf{x}^{(t)}, y_1^{(t)}, \ldots, y_M^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y_1^{(t)}, \ldots, y_M^{(t)} \sim p(\mathbf{x}, y_1, \ldots, y_M)$

- Test time
  - Data:
    $\{\mathbf{x}^{(t)}, y_1^{(t)}, \ldots, y_M^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y_1^{(t)}, \ldots, y_M^{(t)} \sim p(\mathbf{x}, y_1, \ldots, y_M)$

- Example: object recognition in images with multiple objects

Multi-Task Learning



Structured Output Prediction

- Training time
  - Data:

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

Data of arbitrary structure (vector, sequence, graph).

  - Setting:

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

- Test time
  - Data:

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

Data of arbitrary structure (vector, sequence, graph).

  - Setting:

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

- Example: Image caption generation, machine translation

One-Shot Learning

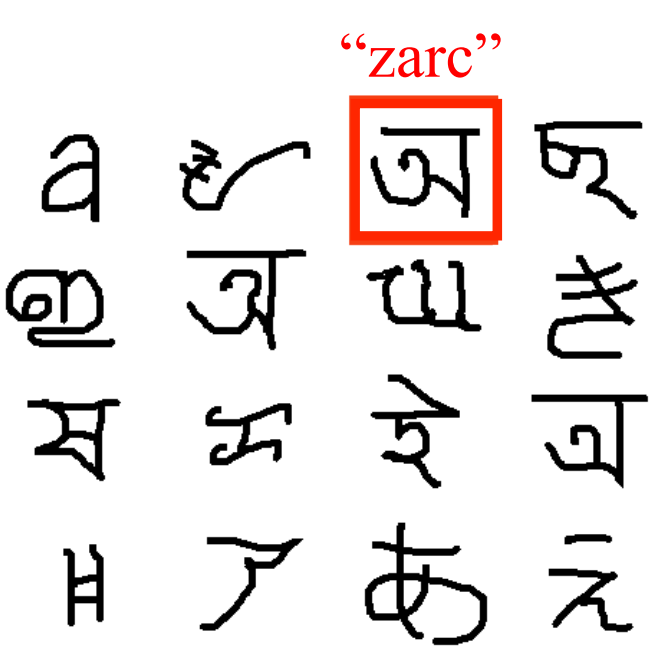
- Training time
  - Data:
    $\{\mathbf{x}^{(t)}, y^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$
    $y^{(t)} \in \{1, \ldots, C\}$

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

- Test time
  - Data:
    $\{\mathbf{x}^{(t)}, y^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$
    $y^{(t)} \in \{C+1, \ldots, C+M\}$

Additional data: A single labeled example from each of the M new classes

Transfer Learning



How can we learn a novel concept – a high dimensional statistical object – from few examples.

Supervised Learning

Segway

Motorcycle

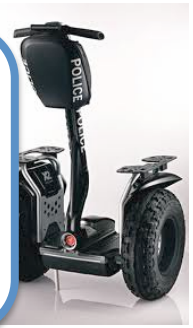
Test:

Learning to Learn

Background Knowledge

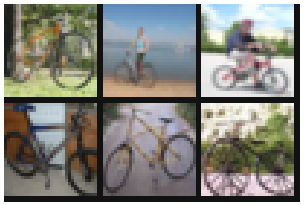
Millions of unlabeled images

Learn to Transfer Knowledge

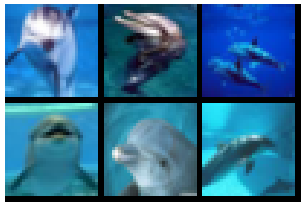


Key problem in computer vision, speech perception, natural language processing, and many other domains.

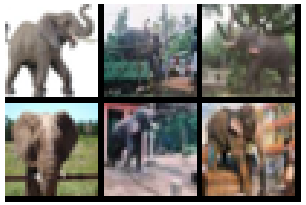
Some labeled images



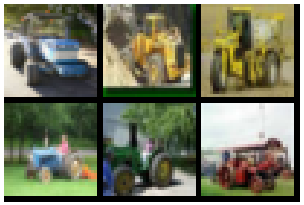
Bicycle



Dolphin



Elephant

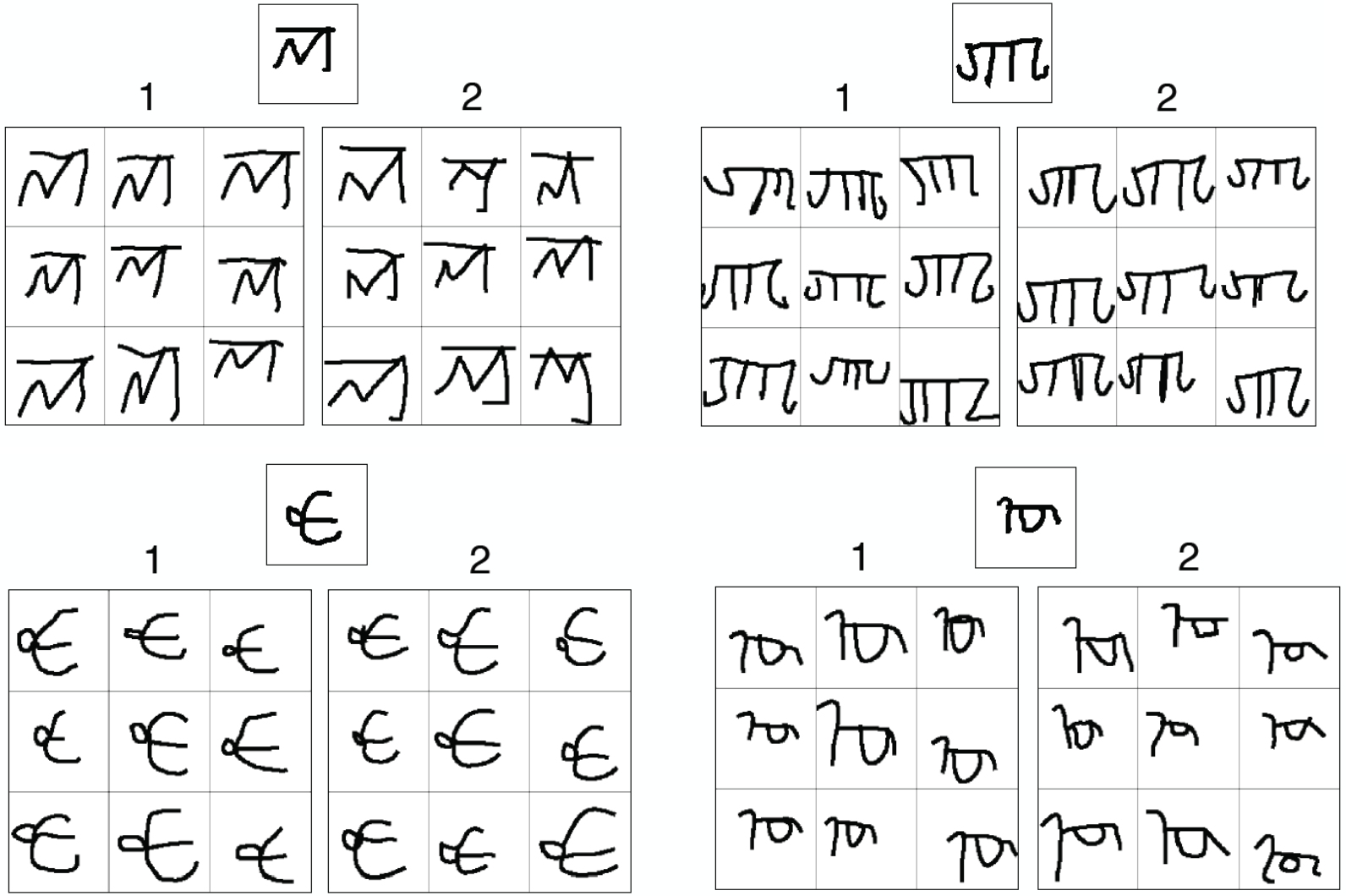


Tractor

Learn novel concept from one example

Test:

One-Shot Learning: Humans vs. Machines



(Lake, Salakhutdinov, Tenenbaum, Science, 2015)

Zero-Shot Learning

- Training time
  - Data:
    $\{\mathbf{x}^{(t)}, y^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$
    $y^{(t)} \in \{1, \ldots, C\}$

Additional data: Description vector $\mathbf{z}_c$ of each of the C classes

- Test time
  - Data:
    $\{\mathbf{x}^{(t)}, y^{(t)}\}$
  - Setting:
    $\mathbf{x}^{(t)}, y^{(t)} \sim p(\mathbf{x}, y)$
    $y^{(t)} \in \{C+1, \ldots, C+M\}$

Additional data: description vector $\mathbf{z}_c$ of each of the new M classes