

10707

Deep Learning

Russ Salakhutdinov

Machine Learning Department
rsalakhu@cs.cmu.edu

Neural Networks II

Initialization

- Initialize biases to 0
- For weights
 - Can not initialize weights to 0 with tanh activation
 - All gradients would be zero (saddle point)
 - Can not initialize all weights to the same value
 - All hidden units in a layer will always behave the same
 - Need to break symmetry
 - Sample $\mathbf{W}_{i,j}^{(k)}$ from $U[-b, b]$, where

$$b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$$

Sample around 0 and
break symmetry



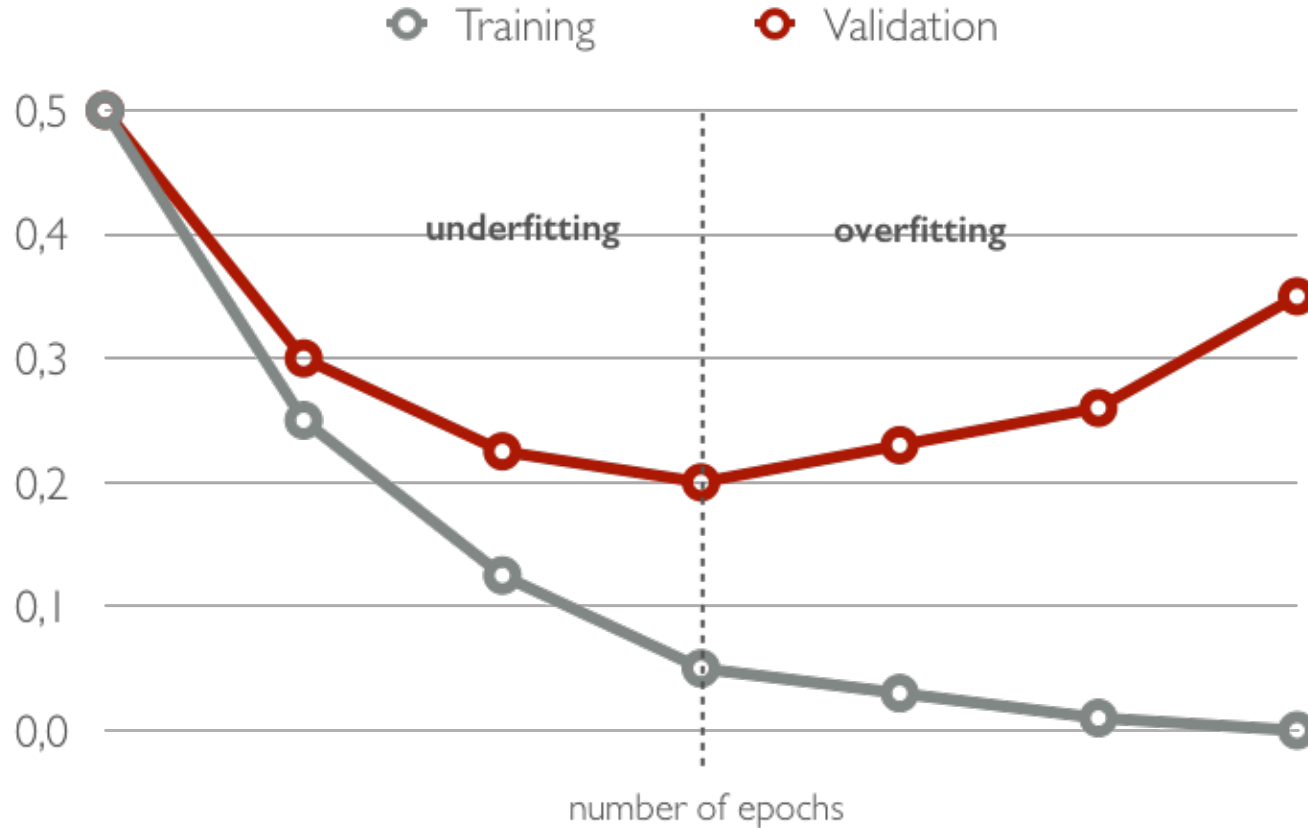
Size of $\mathbf{h}^{(k)}(\mathbf{x})$

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** $\mathcal{D}^{\text{test}}$
- Remember: 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).



Tricks of the Trade:

- Normalizing your (real-valued) data:
 - for each dimension x_i subtract its training set mean
 - divide each dimension x_i by its training set standard deviation
 - this can speed up training
- Decreasing the learning rate: As we get closer to the optimum, 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)

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:

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

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

Adapting Learning Rates

- Updates with adaptive learning rates (“one learning rate per parameter”)

- **Adagrad**: learning rates are scaled by the square root of the cumulative sum of squared gradients

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

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

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

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

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

Gradient Checking

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

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

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

Debugging on Small Dataset

- Next, make sure your model can overfit on a smaller dataset (~ 500-1000 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
- This does not mean that you have computed gradients correctly:
 - You could still overfit with some of the gradients being wrong

Feedforward Neural Networks

- ▶ How neural networks predict $f(\mathbf{x})$ given an input $\mathbf{x}$:

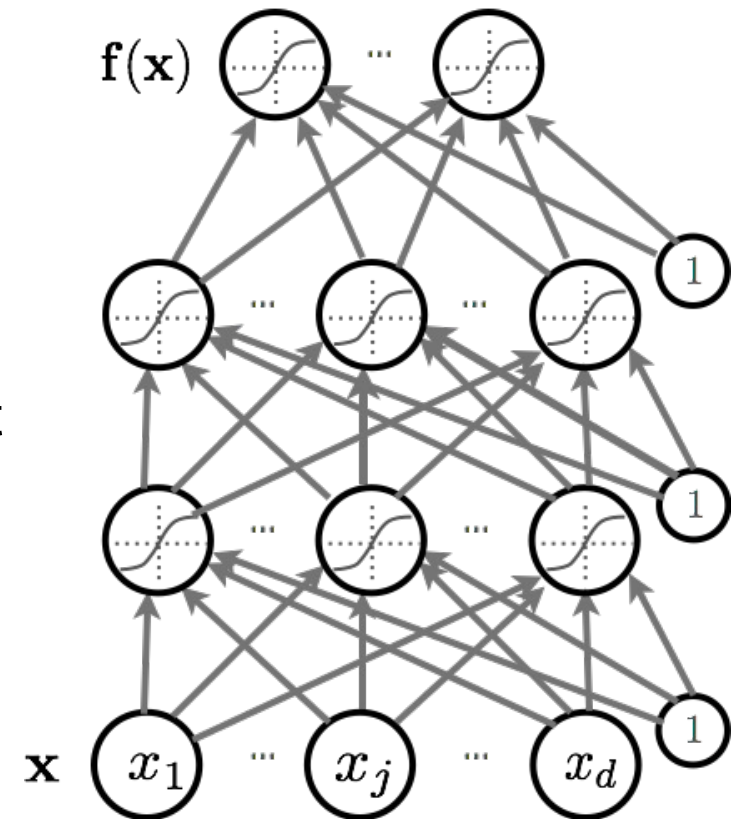
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



Feedforward Neural Networks

- ▶ How neural networks predict $f(\mathbf{x})$ given an input $\mathbf{x}$:

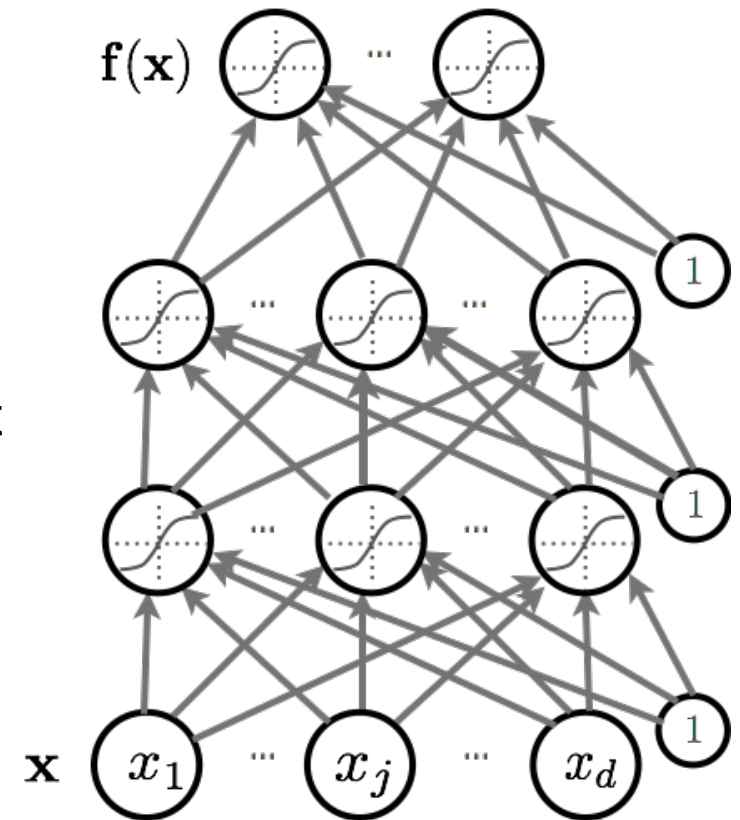
- Forward propagation
- Types of units
- Capacity of neural networks

- ▶ How to train neural nets:

- Loss function
- Backpropagation with gradient descent

- ▶ More recent techniques:

- Dropout
- Batch normalization
- Unsupervised Pre-training



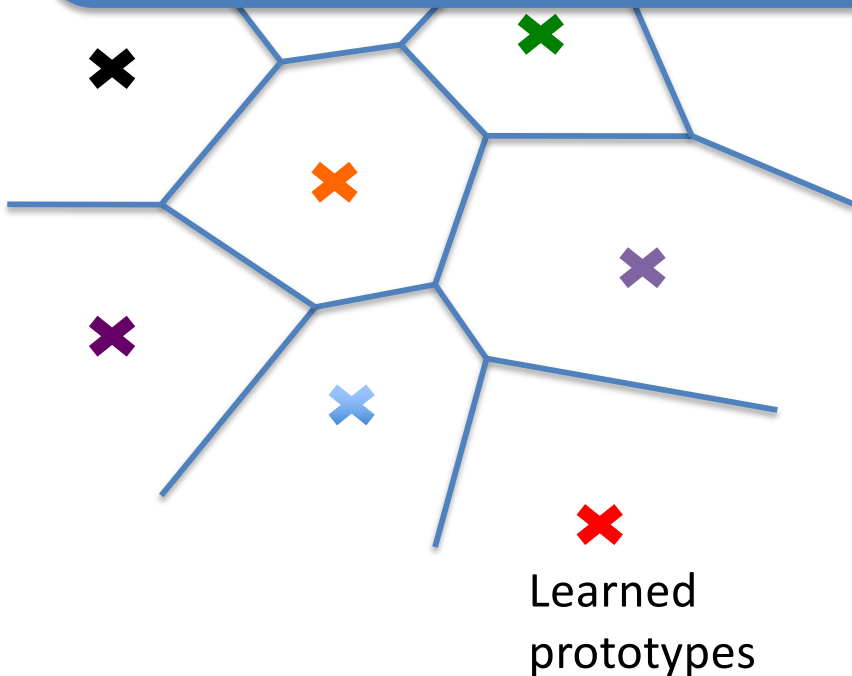
Learning Distributed Representations

- 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

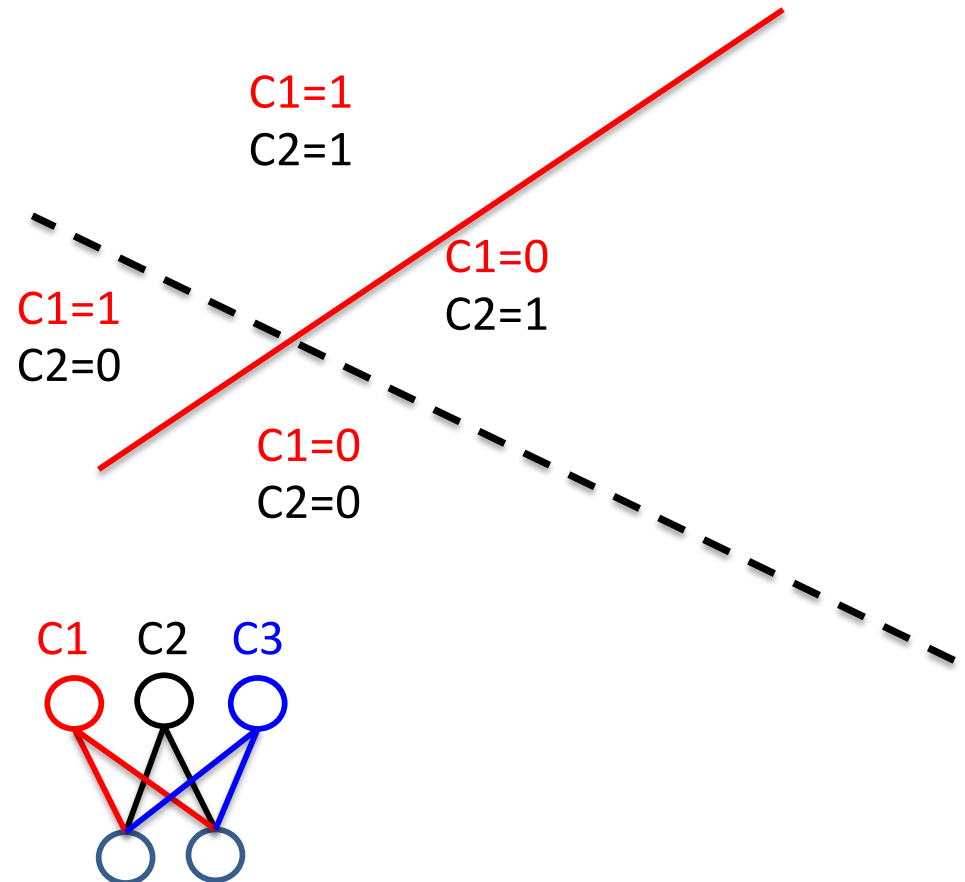
Local vs. Distributed Representations

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

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



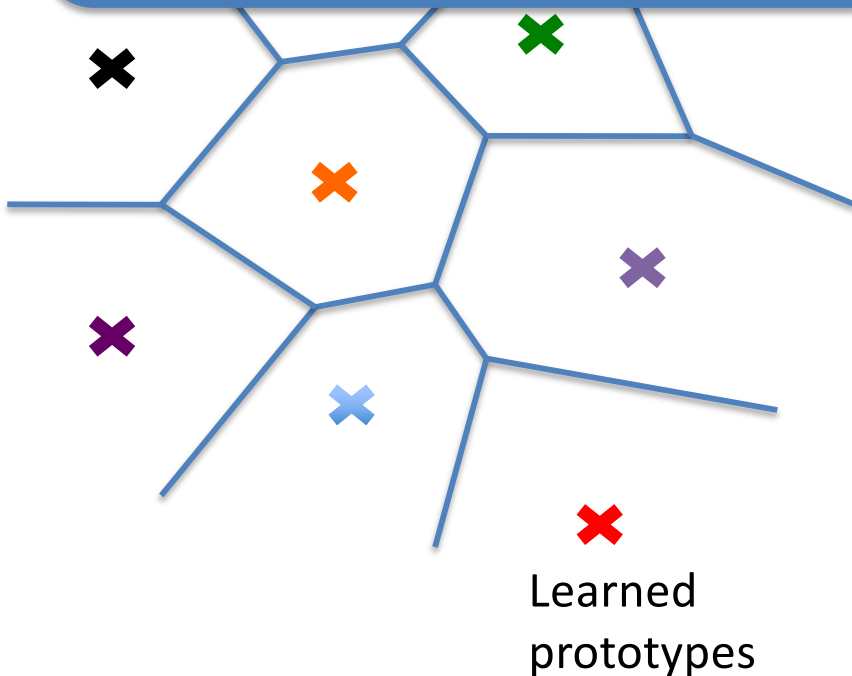
- RBMs, Factor models, PCA, Sparse Coding, Deep models



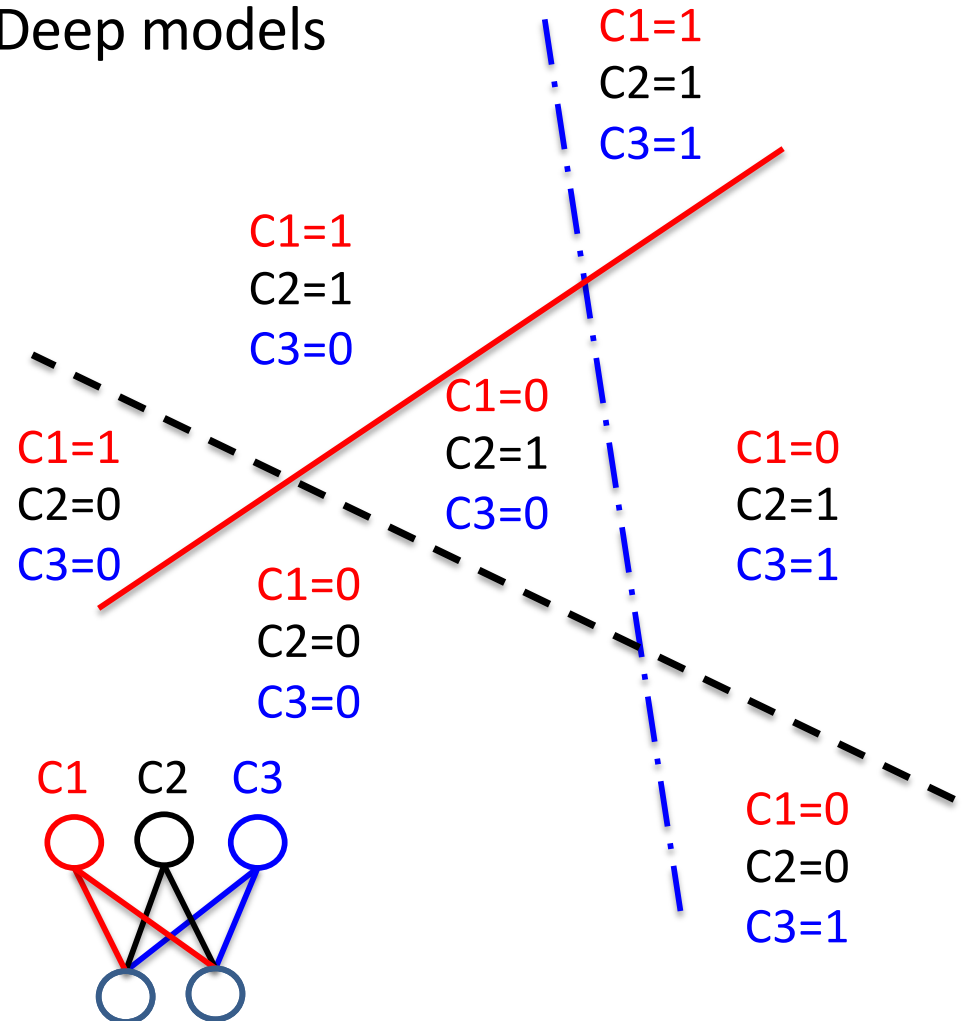
Local vs. Distributed Representations

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

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



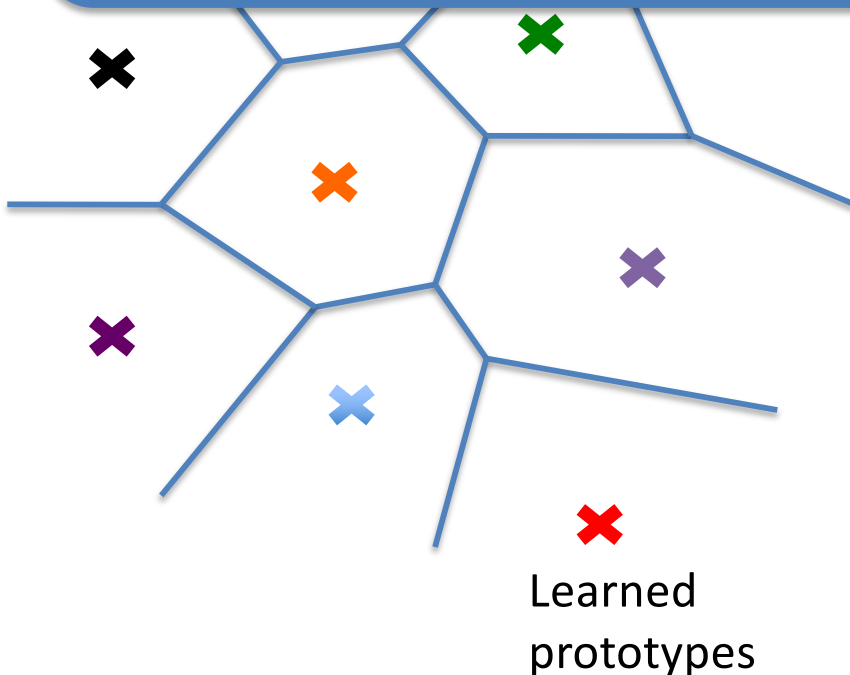
- RBMs, Factor models, PCA, Sparse Coding, Deep models



Local vs. Distributed Representations

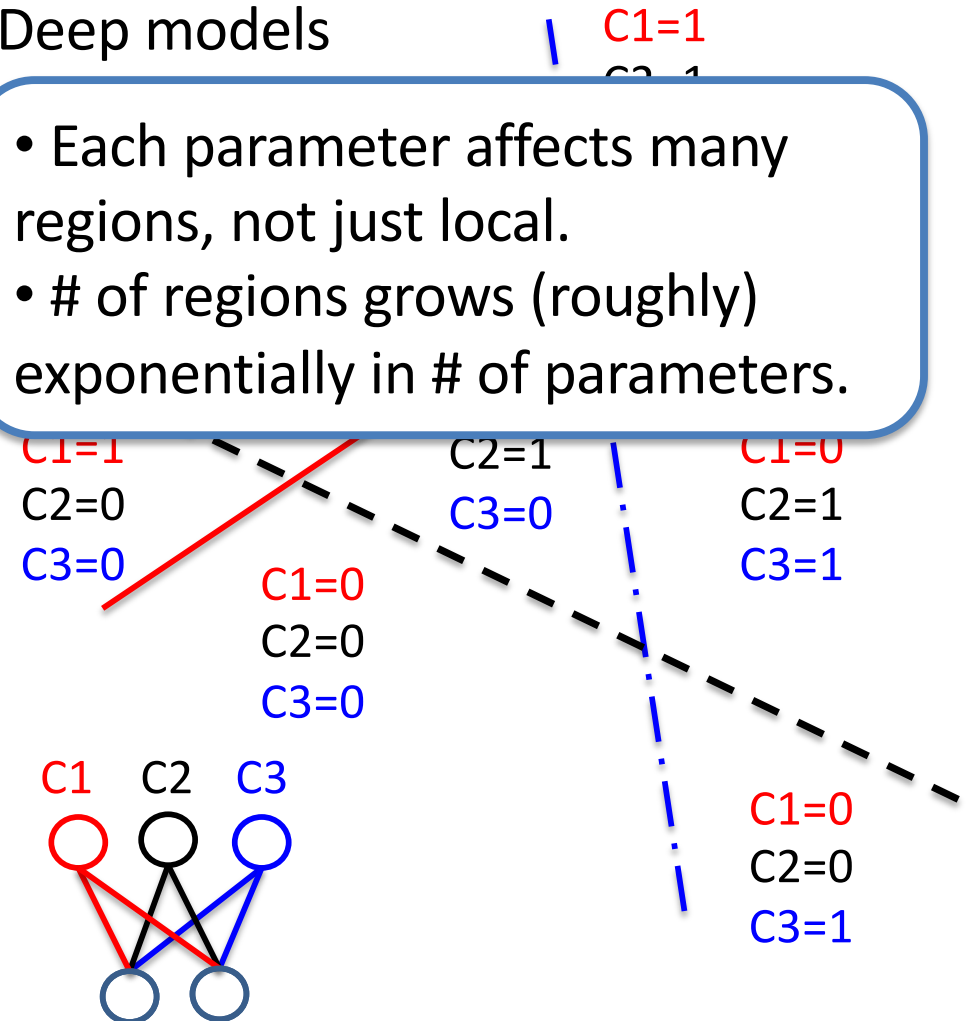
- Clustering, Nearest Neighbors, RBF SVM, local density estimators

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

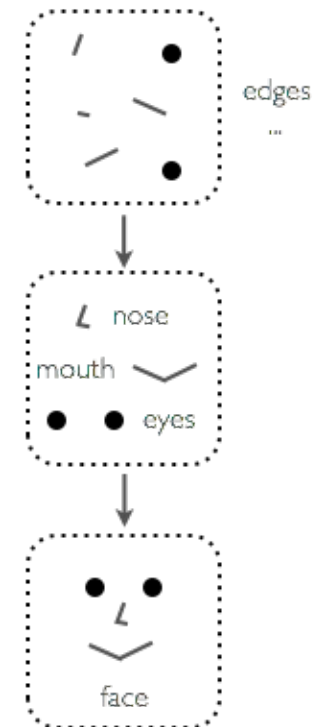
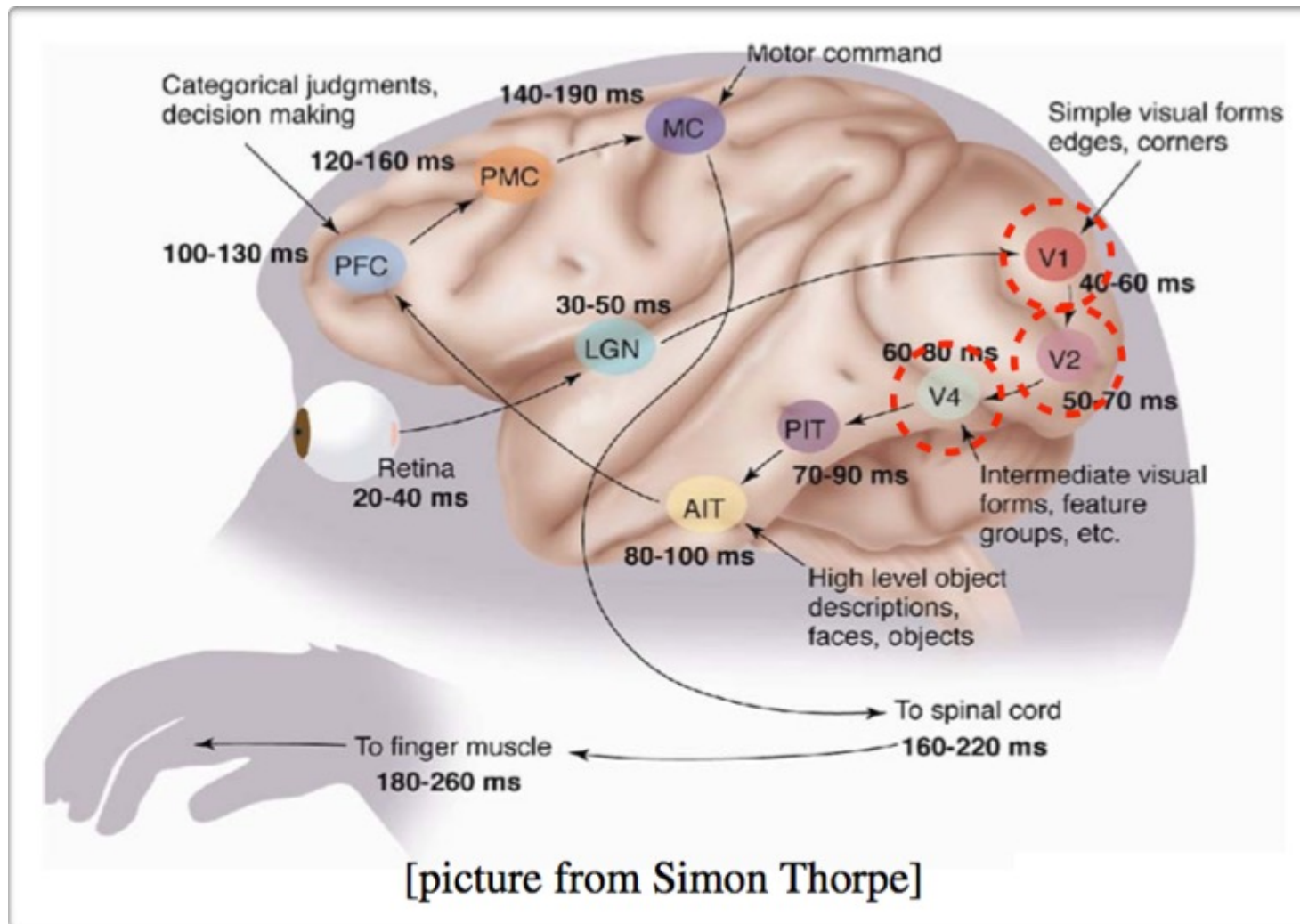


- RBMs, Factor models, PCA, Sparse Coding, Deep models

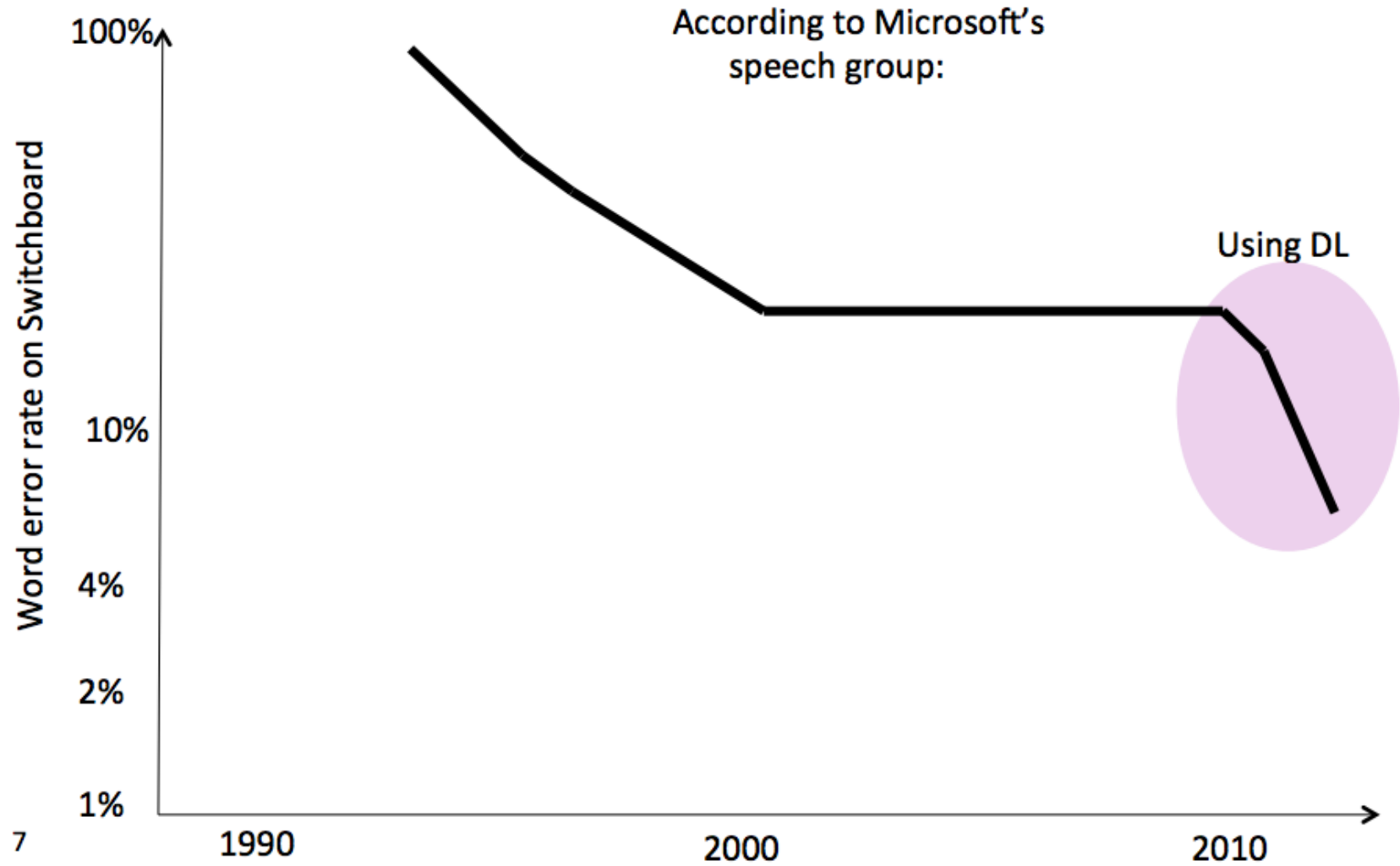
- Each parameter affects many regions, not just local.
- # of regions grows (roughly) exponentially in # of parameters.



Inspiration from Visual Cortex

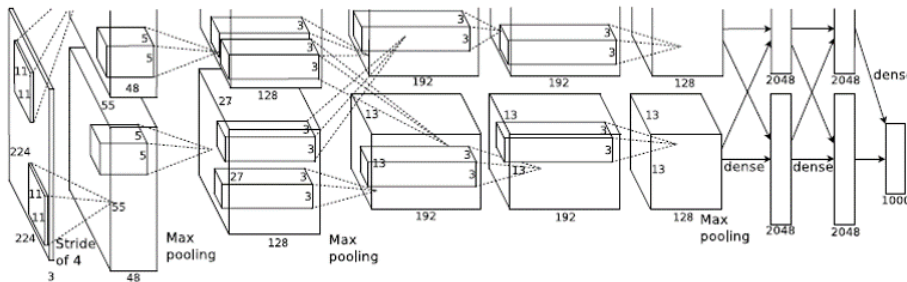


Success Story: Speech Recognition



Success Story: Image Recognition

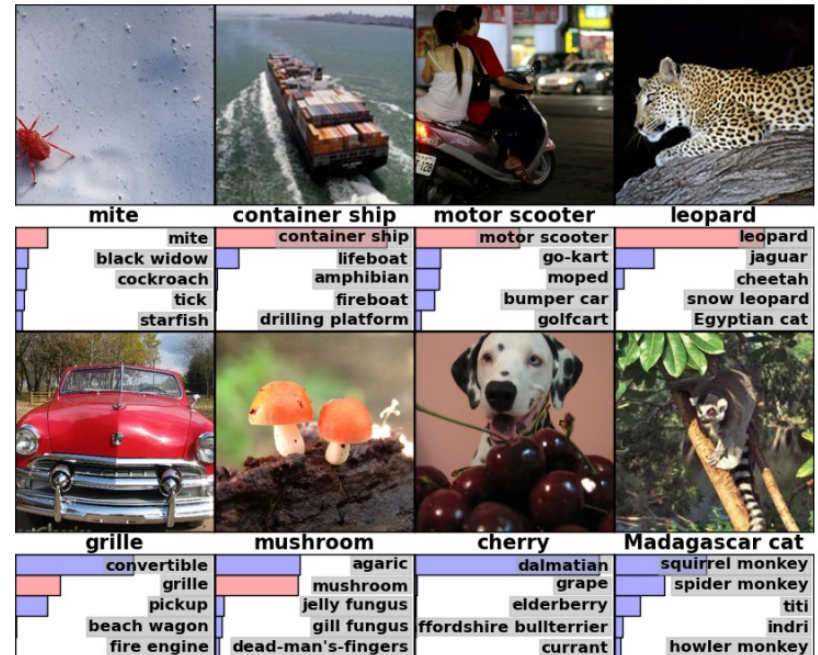
- Deep Convolutional Nets for Vision (Supervised)



IMAGENET

1.2 million training images

1000 classes

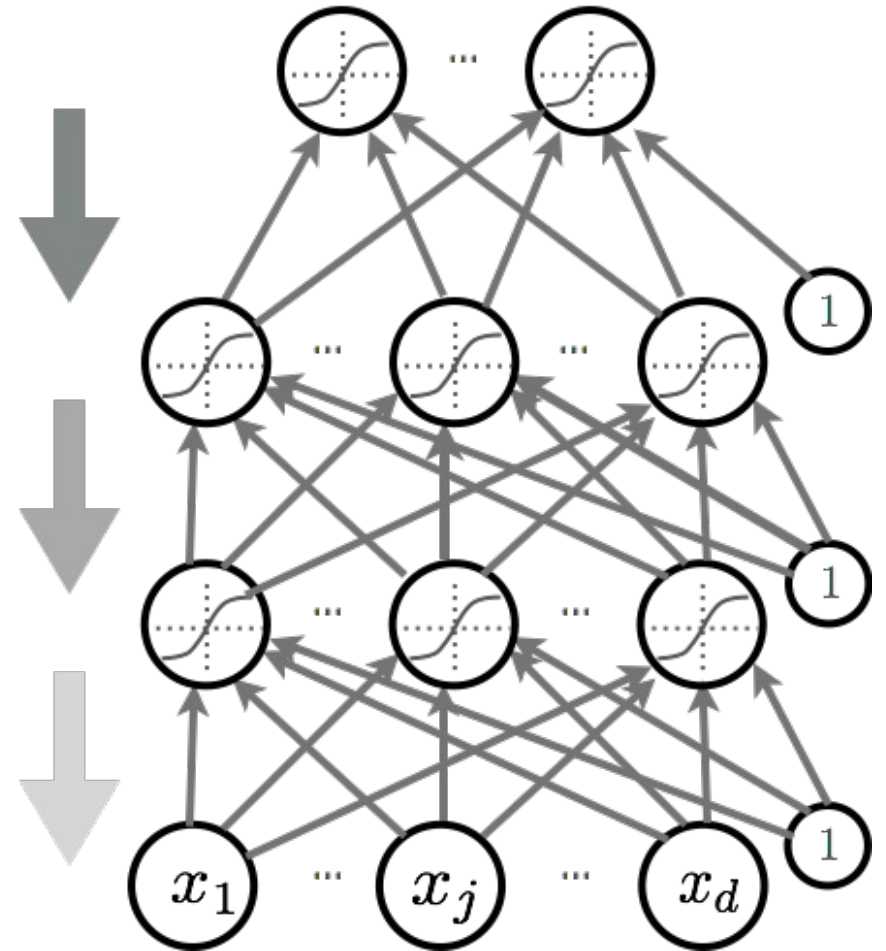


Why Training is Hard

- First hypothesis: **Hard optimization problem** (underfitting)

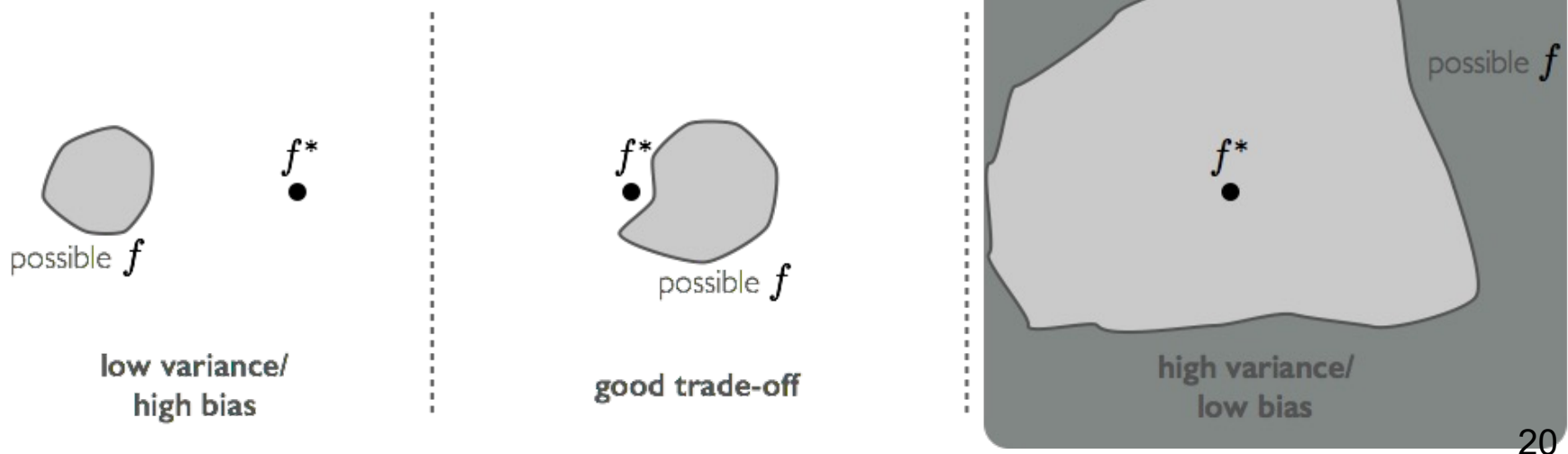
- vanishing gradient problem
- saturated units block gradient propagation

- 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

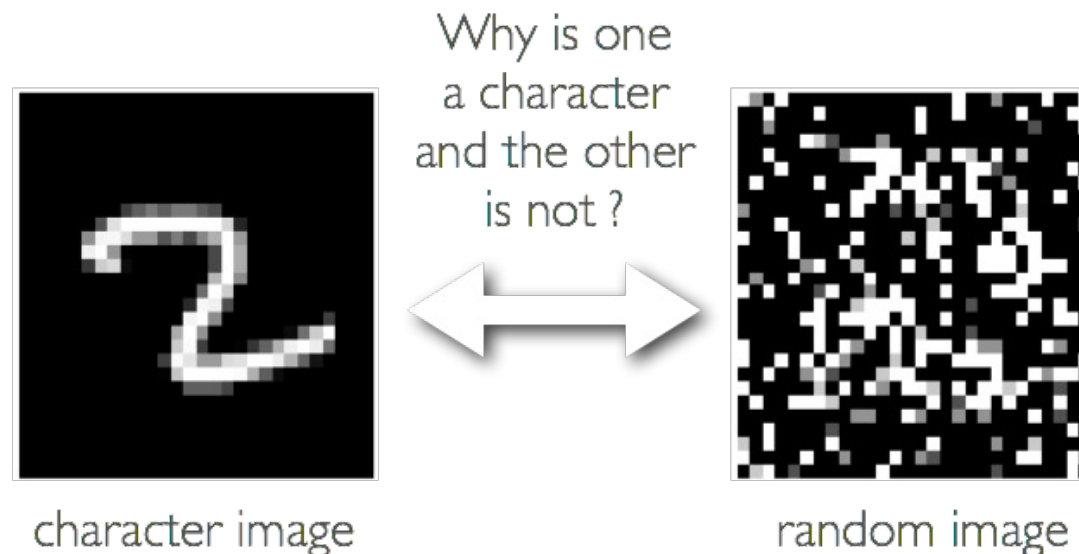


Why Training is Hard

- 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

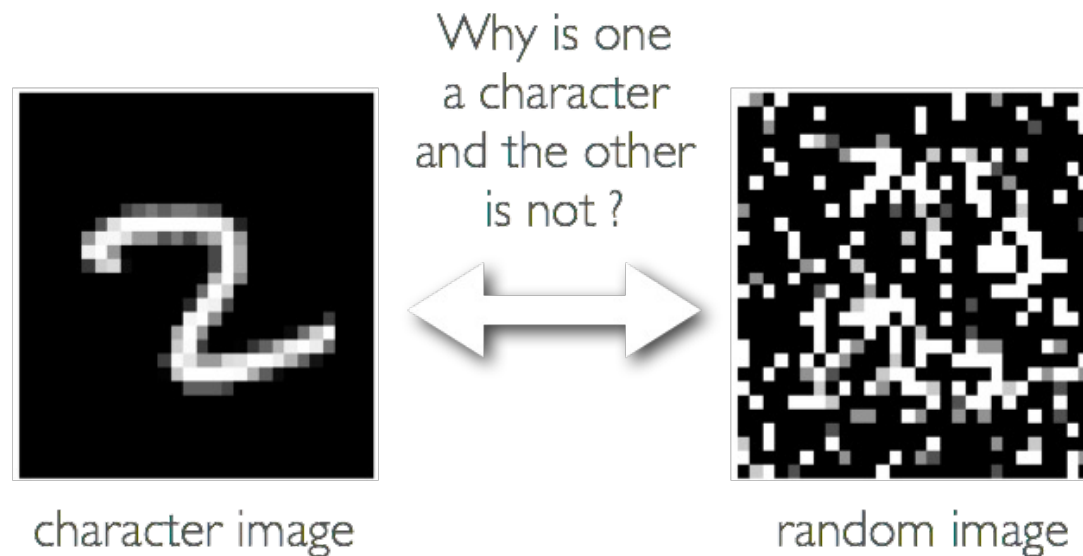
- Initialize hidden layers using **unsupervised learning**
 - Force network to represent latent structure of input distribution



- Encourage hidden layers to encode that structure

Unsupervised Pre-training

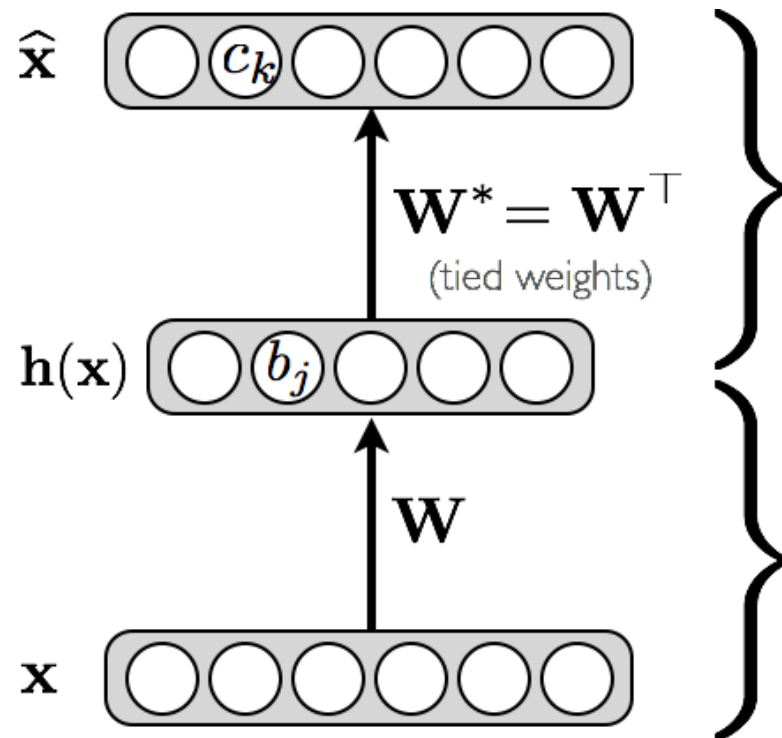
- Initialize hidden layers using **unsupervised learning**
 - This is a harder task than supervised learning (classification)



- Hence we expect less overfitting

Autoencoders: Preview

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



Decoder

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

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**

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

- Cross-entropy error function $f(\mathbf{x}) \equiv \hat{\mathbf{x}}$

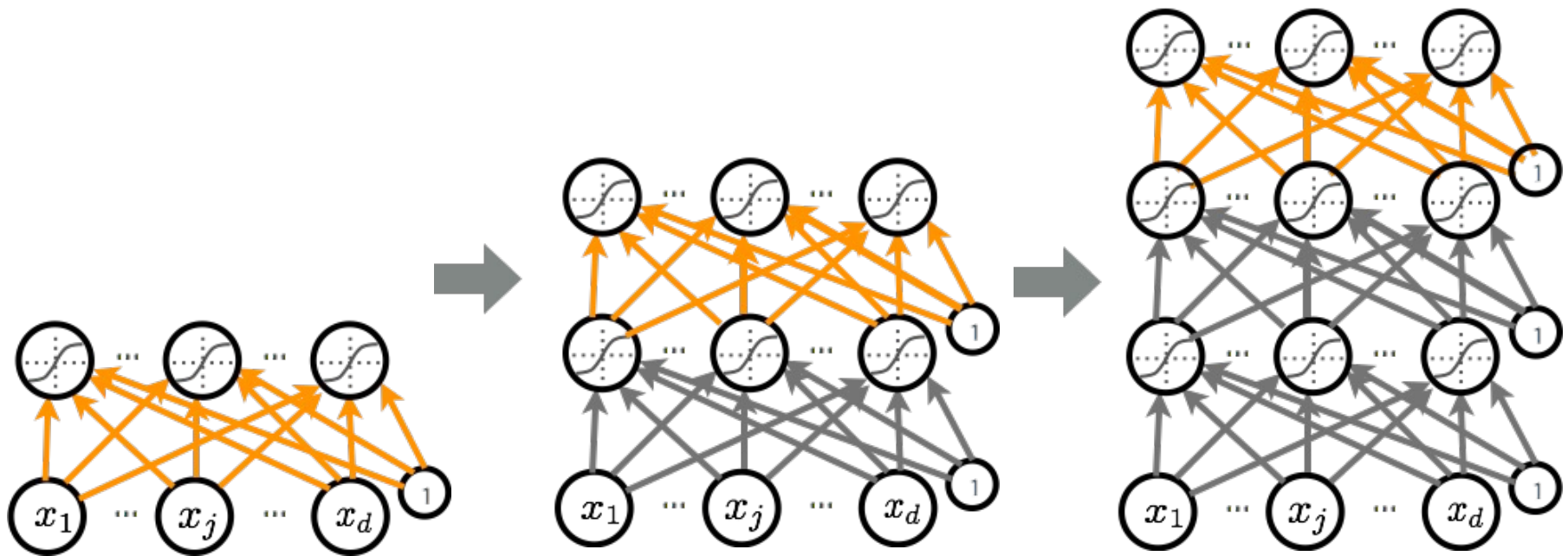
- Loss function for **real-valued inputs**

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

- 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

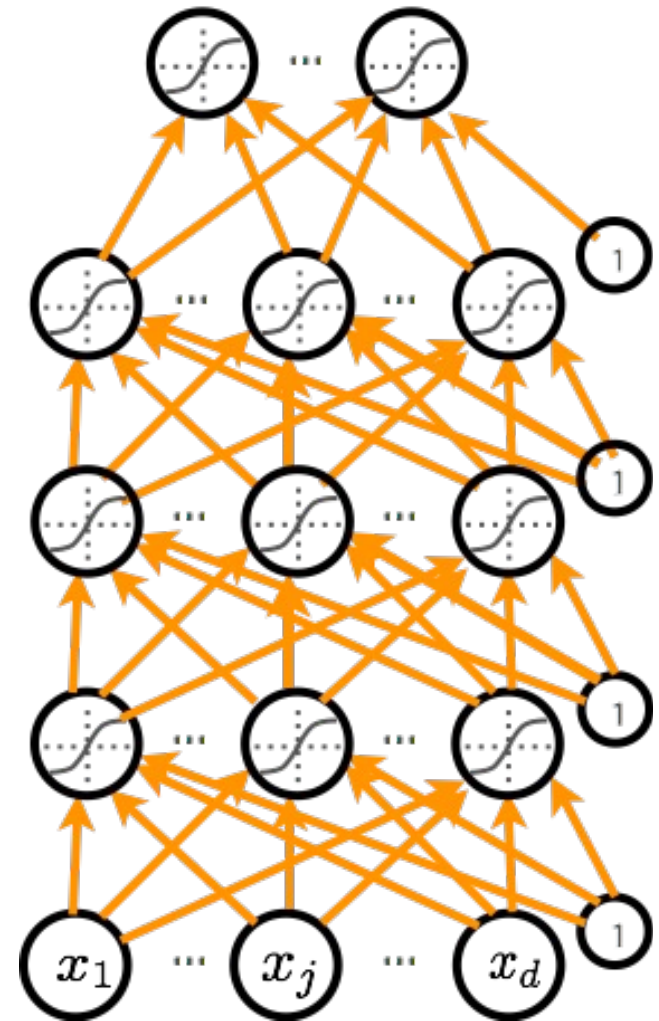


Pre-training

- 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 ...
- **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

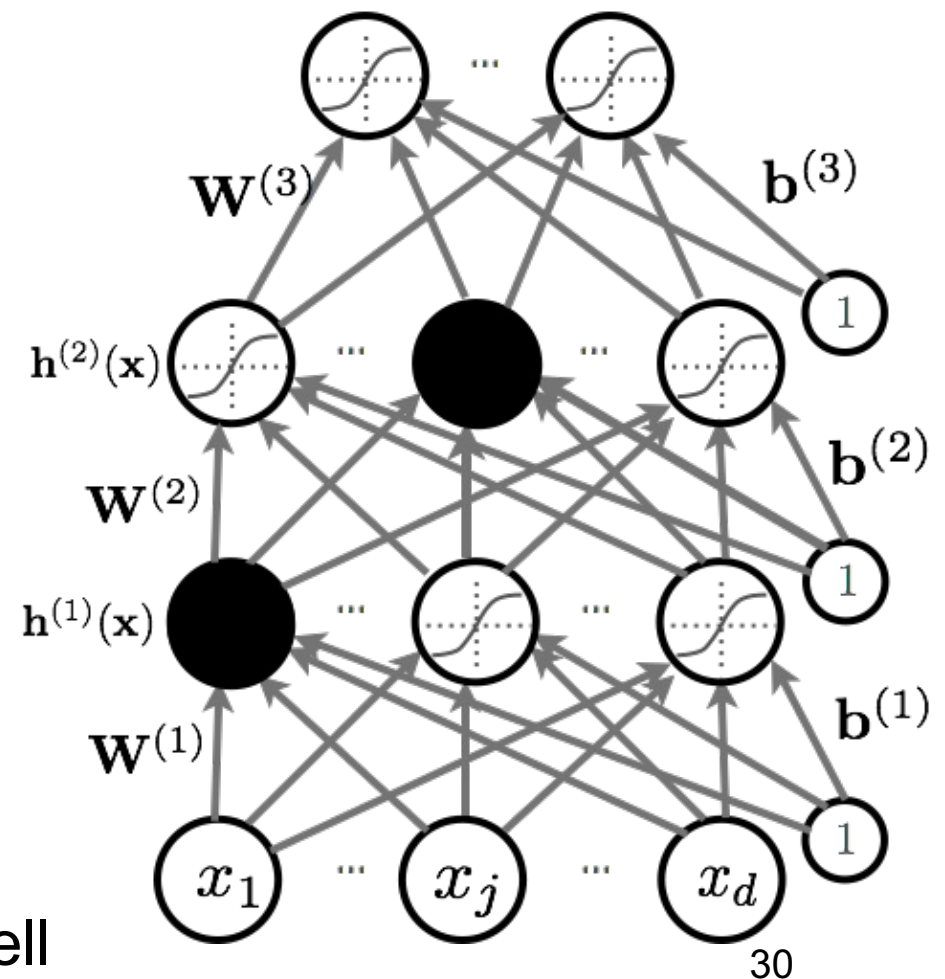
- First hypothesis (underfitting): better optimize
 - Use better optimization tools (e.g. batch-normalization, second order methods, such as KFAC)
 - Use GPUs, distributed computing.
- Second hypothesis (overfitting): use better regularization
 - Unsupervised pre-training
 - Stochastic drop-out training
- 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 $k > 0$

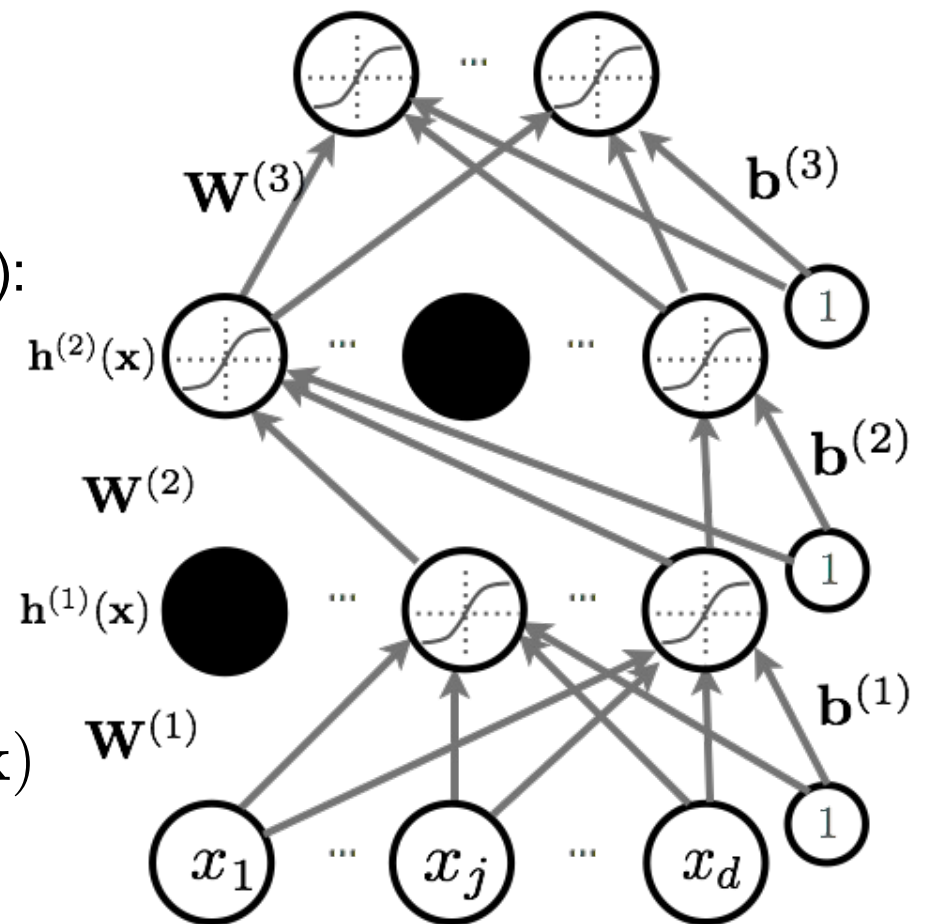
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

- 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):

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

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

- For $k=L+1$ to 1

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

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

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

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

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

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

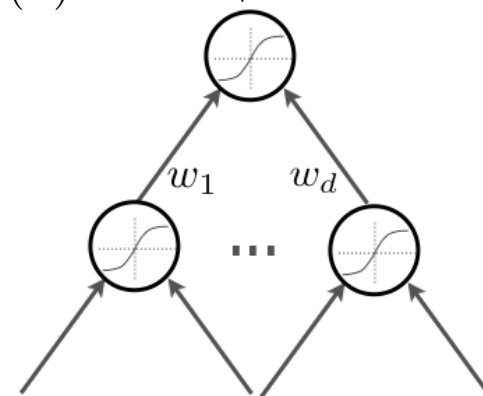
- Unsupervised pre-training
- Stochastic drop-out training

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

Parameters to be learned: γ, β

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

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

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

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

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

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 $y_i \leftarrow \gamma \hat{x}_i + \beta$ 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

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

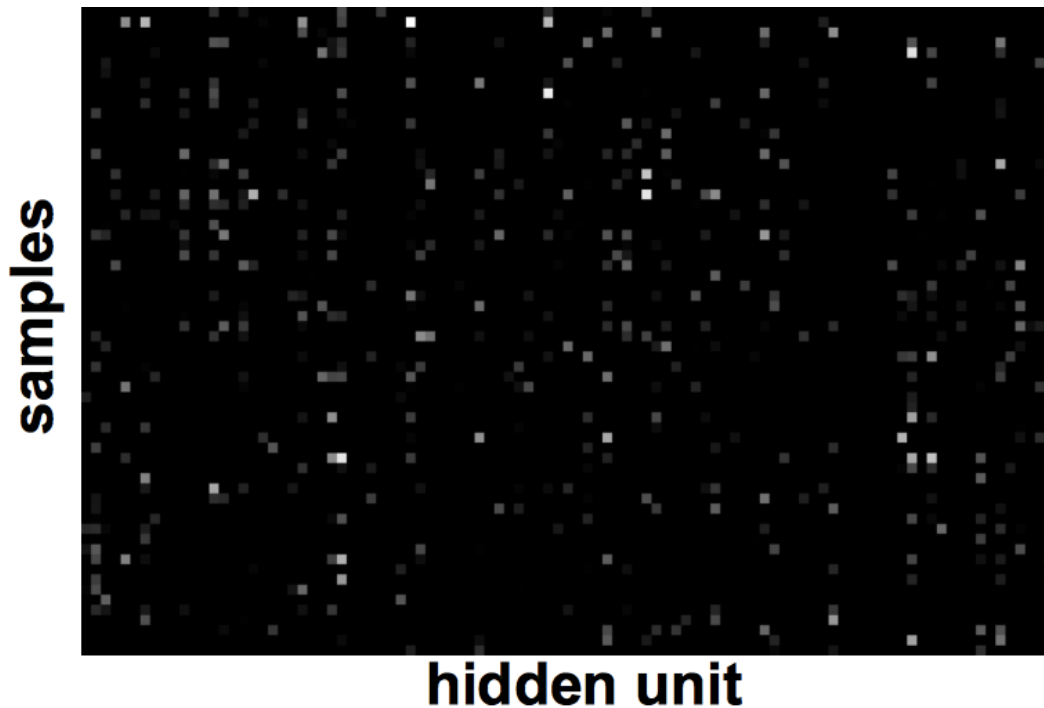
- SGD with momentum, batch-normalization, and dropout usually works very well
- Pick learning rate by running on a subset of the data
 - Start with large learning rate & divide by 2 until loss does not diverge
 - Decay learning rate by a factor of ~ 100 or more by the end of training
- Use ReLU nonlinearity
- Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.

Improving Generalization

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

Visualization

- Check gradients numerically by finite differences
- Visualize features (features need to be uncorrelated) and have high variance

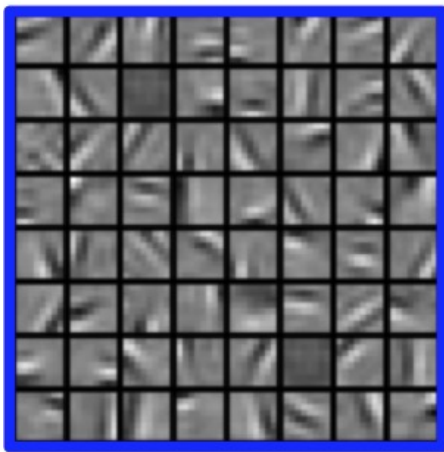


- **Good training:** hidden units are sparse across samples

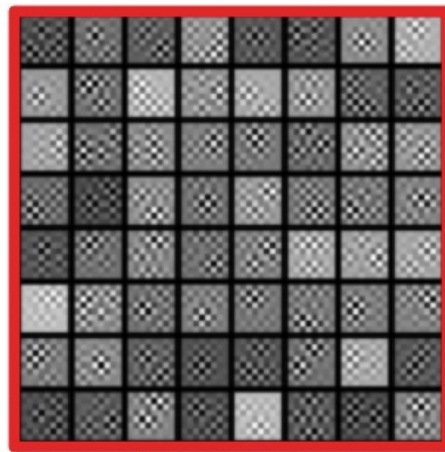
Visualization

- 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

GOOD

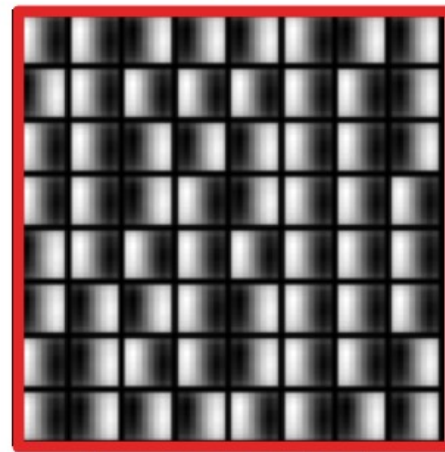


BAD



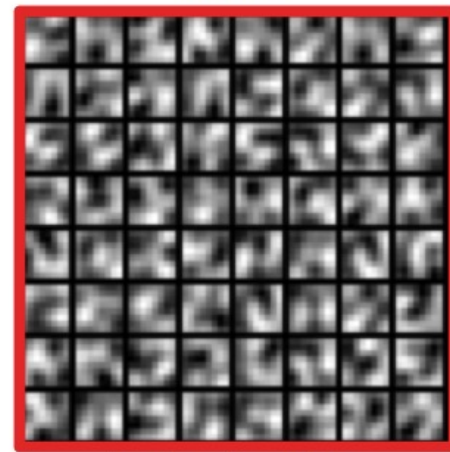
too noisy

BAD



too correlated

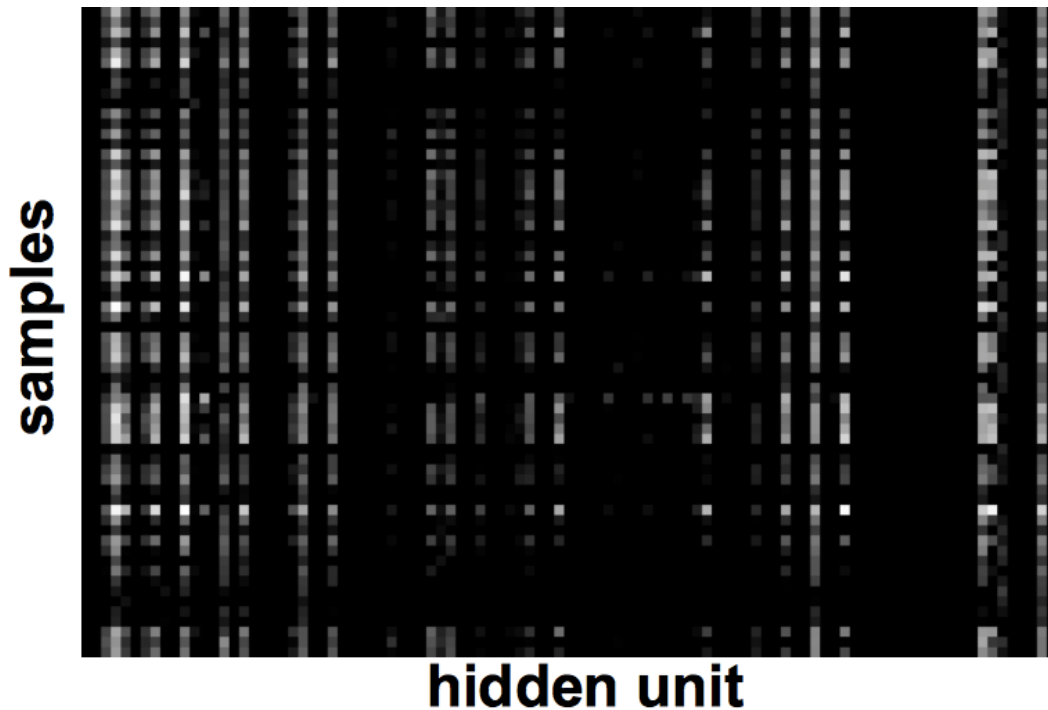
BAD



lack structure

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

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 $\rightarrow 0$.

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