

# Lecture 13

## Deep Belief Networks

Michael Picheny, Bhuvana Ramabhadran, Stanley F. Chen

IBM T.J. Watson Research Center  
Yorktown Heights, New York, USA  
{picheny, bhuvana, stanchen}@us.ibm.com

12 December 2012

## A spectrum of Machine Learning Tasks

### Typical Statistics

- Low-dimensional data (e.g. less than 100 dimensions)
- Lots of noise in the data
- There is not much structure in the data, and what structure there is, can be represented by a fairly simple model.
- The main problem is distinguishing true structure from noise.

# A spectrum of Machine Learning Tasks Cont'd

## Artificial Intelligence

- High-dimensional data (e.g. more than 100 dimensions)
- The noise is not sufficient to obscure the structure in the data if we process it right.
- There is a huge amount of structure in the data, but the structure is too complicated to be represented by a simple model.
- The main problem is figuring out a way to represent the complicated structure so that it can be learned.

3/58

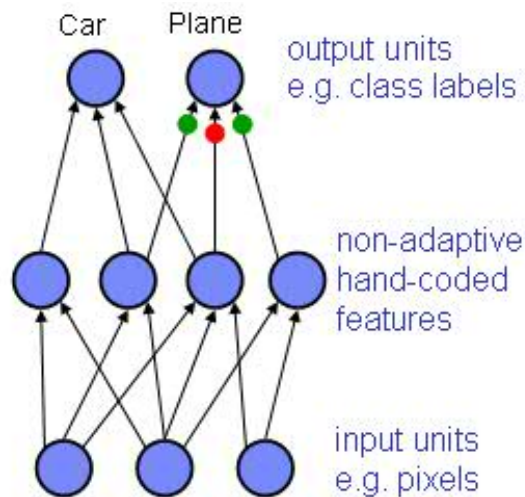
## Why are Neural Networks interesting?

- GMMs and HMMs to model our data
- Neural networks give a way of defining a complex, non-linear model with parameters  $W$  (weights) and biases ( $b$ ) that we can fit to our data
- In past 3 years, DBNs have shown large improvements on small tasks in image recognition and computer vision
- DBNs are slow to train, limiting research for large tasks
- More recently extensive use of DBNs for large vocabulary

4/58

# Initial Neural Networks

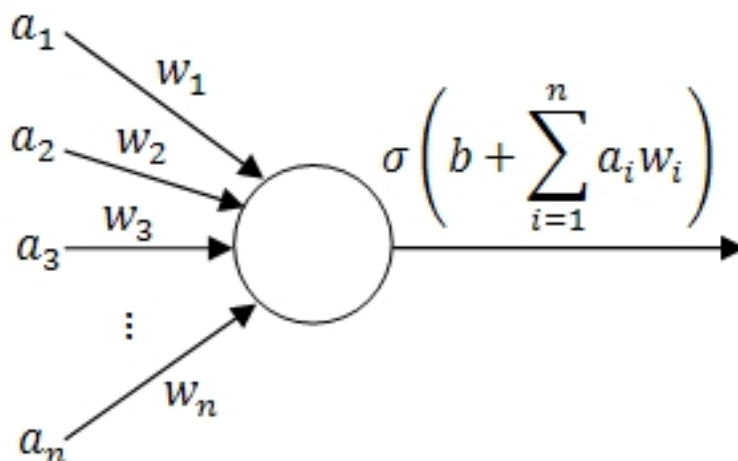
- Perceptrons ( 1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
- Simple learning algorithm for adjusting the weights.
- Building Blocks of modern day networks



5 / 58

# Perceptrons

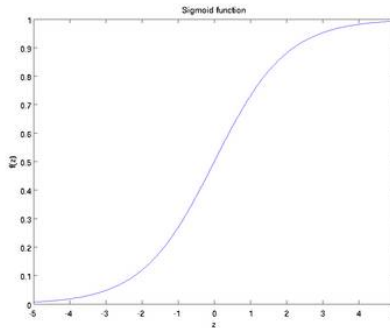
- The simplest classifiers from which neural networks are built are perceptrons.
- A perceptron is a linear classifier which takes a number of inputs  $a_1, \dots, a_n$ , scales them using some weights  $w_1, \dots, w_n$ , adds them all up (together with some bias  $b$ ) and feeds the result through an activation function,  $\sigma$ .



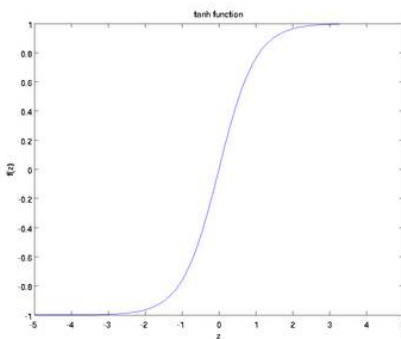
6 / 58

# Activation Function

- Sigmoid  $f(z) = \frac{1}{1+\exp(-z)}$



- Hyperbolic tangent  $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



7/58

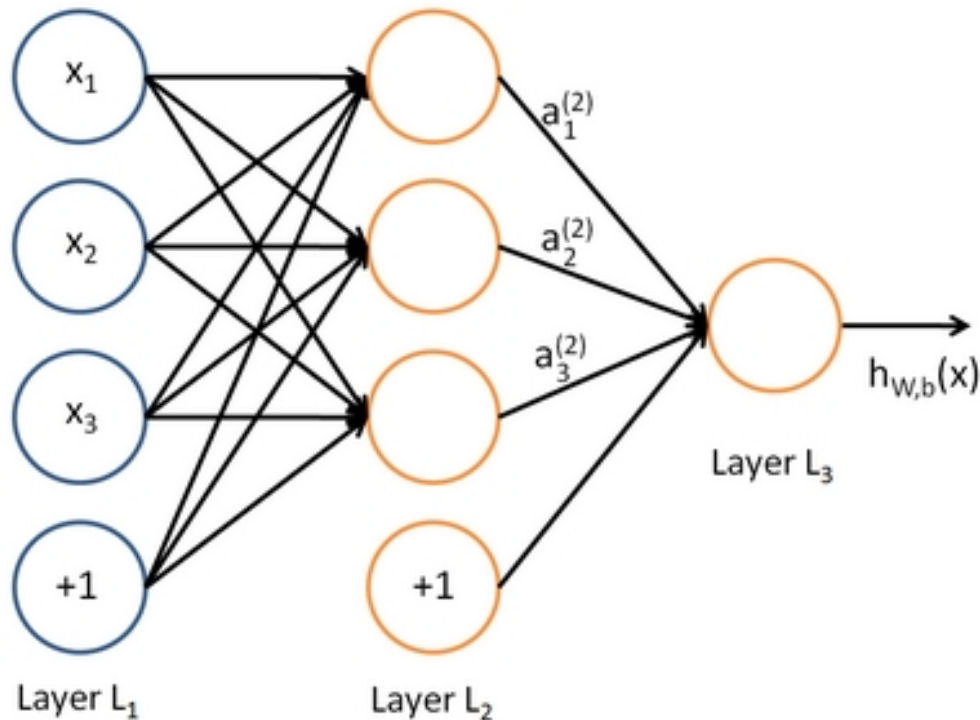
## Derivatives of these activation functions

- If  $f(z)$  is the sigmoid function, then its derivative is given by  $f'(z) = f(z)(1 - f(z))$ .
- If  $f(z)$  is the tanh function, then its derivative is given by  $f'(z) = 1 - (f(z))^2$ .
- Remember this for later!

8/58

# Neural Network

A neural network is put together by putting together many of our simple building blocks.



9/58

## Definitions

- $n_l$  denotes the number of layers in the network;
- $L_1$  is the input layer, and layer  $L_{n_l}$  the output layer.
- Parameters  $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ , where
- $W_{ij}^{(l)}$  is the parameter (or weight) associated with the connection between unit  $j$  in layer  $l$ , and unit  $i$  in layer  $l + 1$ .
- $b_i^{(l)}$  is the bias associated with unit  $i$  in layer  $l + 1$ . Note that bias units don't have inputs or connections going into them, since they always output
- $a_i^{(l)}$  denotes the "activation" (meaning output value) of unit  $i$  in layer  $l$ .

10/58

# Definitions

- This neural network defines  $h_{W,b}(x)$  that outputs a real number. Specifically, the computation that this neural network represents is given by:

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

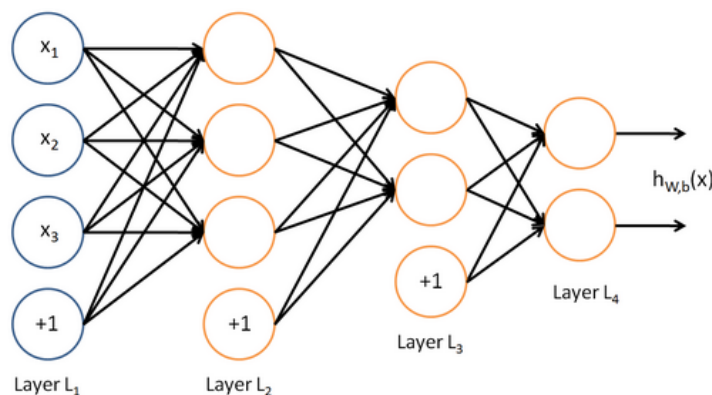
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

- This is called forward propagation.
- Use matrix vector notation and take advantage of linear algebra for efficient computations.

11 / 58

# Another Example

- Generally networks have multiple layers and predict more than one output value.
- Another example of a feed forward network



12 / 58

# How do you train these networks?

- Use Gradient Descent (batch)
- Given a training set  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$
- Define the cost function (error function) with respect to a single example to be:

$$J(W, b; x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2$$

13/58

## Training (contd.)

- For  $m$  samples, the overall cost function becomes

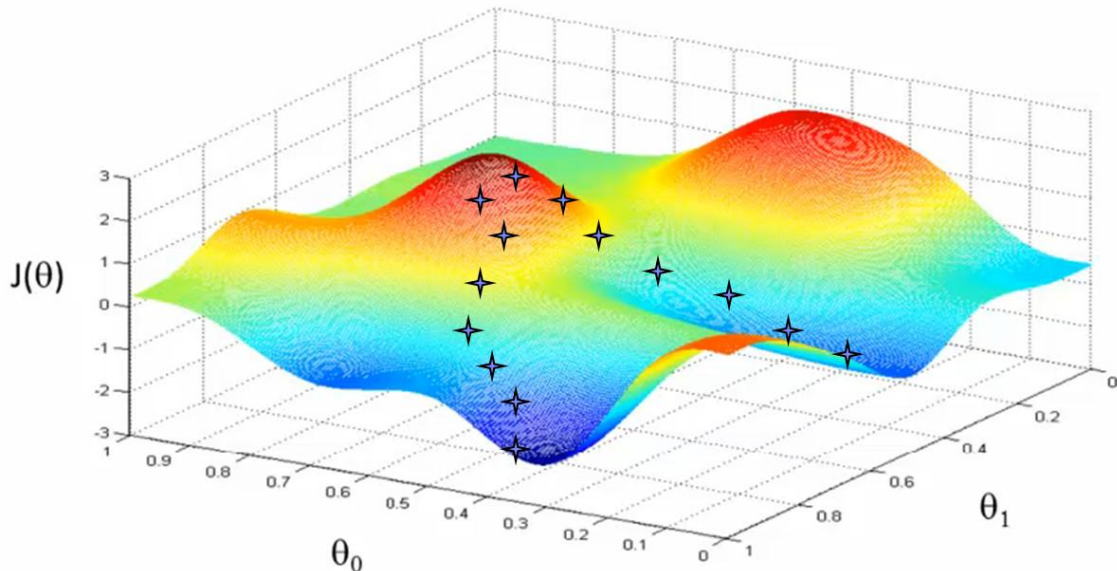
$$\begin{aligned} J(W, b) &= \left[ \frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2 \\ &= \left[ \frac{1}{m} \sum_{i=1}^m \left( \frac{1}{2} \|h_{W,b}(x^{(i)}) - y^{(i)}\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2 \end{aligned}$$

- The second term is a regularization term ("weight decay") that prevent overfitting.
- Goal: minimize  $J(W, b)$  as a function of  $W$  and  $b$ .

14/58

# Gradient Descent

- Cost function is  $J(\theta)$
- minimize  $J(\theta)$
- $\theta$  are the parameters we want to vary



15/58

# Gradient Descent

- Repeat until convergence
- Update  $\theta$  as

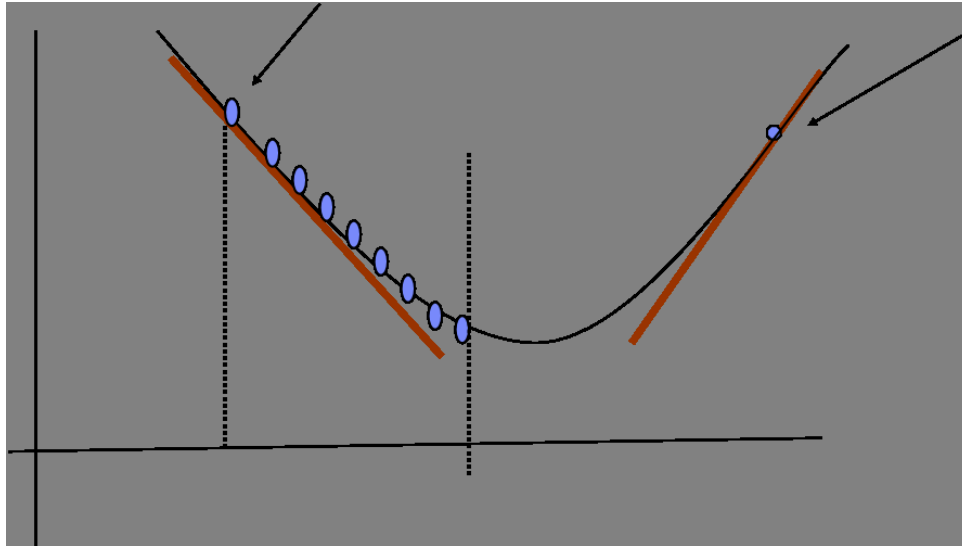
$$\theta_j - \alpha * \frac{\partial}{\partial \theta_j} J(\theta) \nabla j$$

- $\alpha$  determines how big a step in the right direction and is called the learning rate.
- Why is taking the derivative the correct thing to do?

16/58



# Gradient Descent



- As you approach the minimum, you take smaller steps as the gradient gets smaller

17/58

## Returning to our network...

- Goal: minimize  $J(W, b)$  as a function of  $W$  and  $b$ .
- Initialize each parameter  $W_{ij}^{(l)}$  and each  $b_i^{(l)}$  to a small random value near zero (for example, according to a Normal distribution)
- Apply an optimization algorithm such as gradient descent.
- $J(W, b)$  is a non-convex function, gradient descent is susceptible to local optima; however, in practice gradient descent usually works fairly well.

18/58

# Estimating Parameters

- It is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input.
- One iteration of Gradient Descent yields the following parameter updates:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$

- The backpropagation algorithm is an efficient way to computing these partial derivatives.

19/58

# Backpropagation Algorithm

- Let's compute  $\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y)$  and  $\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y)$ , the partial derivatives of the cost function  $J(W, b; x, y)$  with respect to a single example  $(x, y)$ .
- Given the training sample, run a forward pass through the network and compute all the activations
- For each node  $i$  in layer  $l$ , compute an "error term"  $\delta_i^{(l)}$ . This measures how much that node was "responsible" for any errors in the output.

20/58

# Backpropogation Algorithm

- This error term will be different for the output units and the hidden units.
- Output node: Difference between the network's activation and the true target value defines  $\delta_i^{(n_l)}$
- Hidden node: Use a weighted average of the error terms of the nodes that uses  $\delta_i^{(n_l)}$  as an input.

21 / 58

# Backpropogation Algorithm

- Let  $z_i^{(l)}$  denote the total weighted sum of inputs to unit  $i$  in layer  $l$ , including the bias term

$$z_i^{(2)} = \sum_{j=1}^n W_{ij}^{(1)} x_j + b_i^{(1)}$$

- Perform a feedforward pass, computing the activations for layers  $L_2$ ,  $L_3$ , and so on up to the output layer  $L_{n_l}$ .
- For each output unit  $i$  in layer  $n_l$  (the output layer), define

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

22 / 58

## Backpropogation Algorithm Cont'd

- For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$ , define
- For each node  $i$  in layer  $l$ , define

$$\delta_i^{(l)} = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

- We can now compute the desired partial derivatives as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

- Note If  $f(z)$  is the sigmoid function, then its derivative is given by  $f'(z) = f(z)(1 - f(z))$  which was computed in the forward pass.

23/58

## Backpropogation Algorithm Cont'd

- Derivative of the overall cost function  $J(W, b)$  over all training samples can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) = \left[ \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b) = \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)})$$

- Once we have the derivatives, we can now perform gradient descent to update our parameters.

24/58

# Updating Parameters via Gradient Descent

- Using matrix notation

$$W^{(l)} = W^{(l)} - \alpha \left[ \left( \frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[ \frac{1}{m} \Delta b^{(l)} \right]$$

Now we can repeatedly take steps of gradient descent to reduce the cost function  $J(W, b)$  till convergence.

25 / 58

## Optimization Algorithm

- We used Gradient Descent. But that is not the only algorithm.
- More sophisticated algorithms to minimize  $J(\theta)$  exist.
- An algorithm that uses gradient descent, but automatically tunes the learning rate  $\alpha$  so that the step-size used will approach a local optimum as quickly as possible.
- Other algorithms try to find an approximation to the Hessian matrix, so that we can take more rapid steps towards a local optimum (similar to Newton's method).

26 / 58

# Optimization Algorithm

- Examples include the "L-BFGS" algorithm, "conjugate gradient" algorithm, etc.
- These algorithms need for any  $\theta$ ,  $J(\theta)$  and  $\nabla_{\theta}J(\theta)$ . These optimization algorithms will then do their own internal tuning of the learning rate/step-size and compute its own approximation to the Hessian, etc., to automatically search for a value of  $\theta$  that minimizes  $J(\theta)$ .
- Algorithms such as L-BFGS and conjugate gradient can often be much faster than gradient descent.

27 / 58

# Optimization Algorithm Cont'd

- In practice, on-line or Stochastic Gradient Descent is used
- The true gradient is approximated by the gradient from a single or a small number of training samples (mini-batches)
- Typical implementations may also randomly shuffle training examples at each pass and use an adaptive learning rate.

28 / 58

# Administrivia

- Lab 4 handed back today.
  - Answers:  
`/user1/faculty/stanchen/e6870/lab4_ans/`.
- Next Monday: presentations for non-reading projects.
  - Gu and Yang (15m).
  - Yi and Zehui (15m).
  - Laura (10m).
  - Dawen (10m).
  - Colin and Zhuo (15m).
  - Jeremy (10m).
  - Mohammad (10m).
- Papers due next Monday, 11:59pm.
  - Submit via Courseworks DropBox.

29 / 58

# Recap of Neural Networks

- A neural network has multiple hidden layers, where each layer consists of a linear weight matrix a non-linear function (sigmoid)
- Outputs targets: Number of classes (sub-word units)
- Output probabilities used as acoustic model scores (HMM scores)
- Objective function that minimizes loss between target and hypothesized classes
- Benefits: No assumption about a specific data distribution and parameters are shared across all data
- Training is extremely challenging with the objective function being non-convex.
- Recall the weights randomly initialized and can get stuck in local optima.

30 / 58

# Neural Networks and Speech Recognition

- Introduced in the 80s and 90s to speech recognition, but extremely slow and poor in performance compared to the state-of-the-art GMMs/HMMs
- Several papers published by Morgan et. al at ICSI, CMU
- Over the last couple of years, renewed interest with what is known as Deep Belief Networks.

31 / 58

## Deep Belief Networks (DBNs)

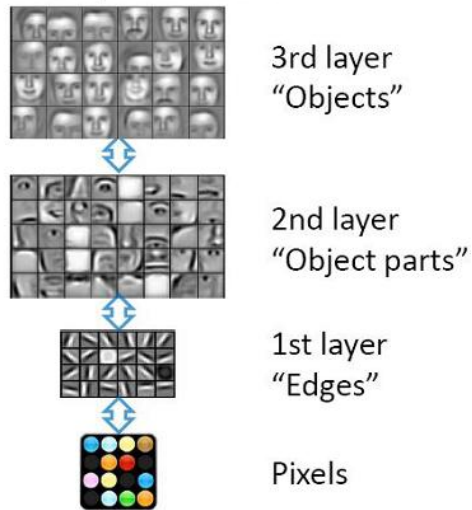
- Deep Belief Networks [Hinton, 2006] Capture higher-level representations of input features Pre-train ANN weights in an unsupervised fashion, followed by fine-tuning (backpropagation)
- Address issues with MLPs getting stuck at local optima.
- DBN Advantages first applied to image recognition tasks, showing gains between 10-30% relative successful application on small vocabulary phonetic recognition task
- Also known as Deep Neural Networks (DNNs)

32 / 58



# What does a DBN learn?

Feature representation



33 / 58

## Good improvement in speech recognition

PER	PER
State of the Art GMM/HMM system, ( <i>Sainath et al., ASRU 2009</i> )	19.4%
DBN system, ( <i>Mohamed et al., ICASSP 2011</i> )	<b>19.0%</b>

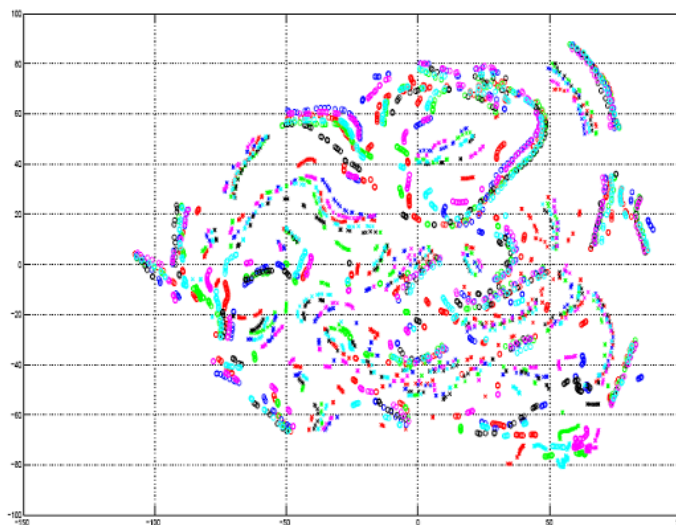
34 / 58

# Why Deepness in Speech?

- Want to analyze activations by different speakers to see what DBN is capturing
- t-SNE [van der Maaten, JMLR 2008] plots produce 2-D embeddings in which points that are close together in the high-dimensional vector space remain close in the 2-D space
- Similar phones from different-speakers are grouped together better at higher layers
- Better discrimination between classes is performed at higher layers

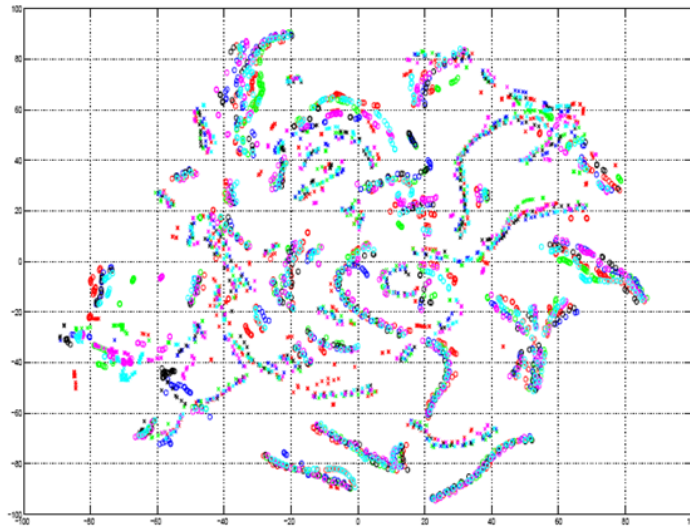
35 / 58

# What does each layer capture?



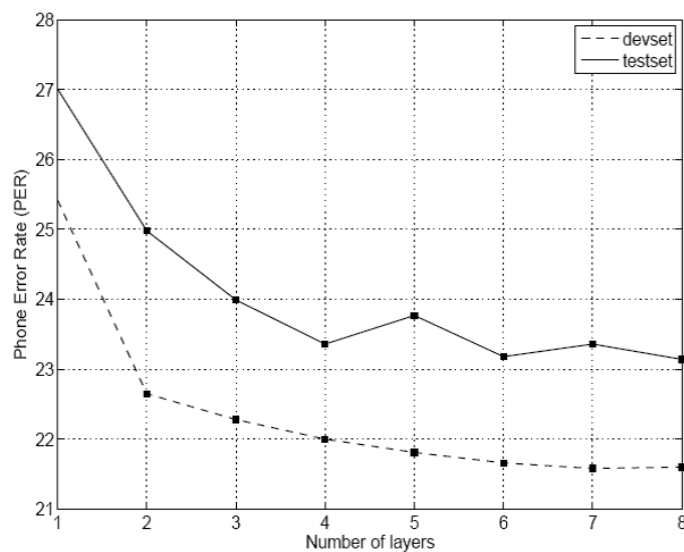
36 / 58

# Second layer



37 / 58

# Experimental Observation for impact of many layers



38 / 58

# DBNs

- What is pretraining? It is unsupervised learning of the network.
  - Learning of multi-layer generative models of unlabelled data by learning one layer of features at a time.
- Keep the efficiency and simplicity of using a gradient method for adjusting the weights, but use it for modeling the structure of the input.
- Adjust the weights to maximize the probability that a generative model would have produced the input.
- But this is hard to do.

39 / 58

# DBNs

- Learning is easy if we can get an unbiased sample from the posterior distribution over hidden states given the observed data.
- For each unit, maximize the log probability that its binary state in the sample from the posterior would be generated by the sampled binary states of its parents.
- We need to integrate over all possible configurations of the higher variables to get the prior for first hidden layer

40 / 58

Some ways to learn DBNs:

- Monte Carlo methods can be used to sample from the posterior. But its painfully slow for large, deep models.
- In the 1990s people developed variational methods for learning deep belief nets These only get approximate samples from the posterior. Nevertheless, the learning is still guaranteed to improve a variational bound on the log probability of generating the observed data.
- If we connect the stochastic units using symmetric connections we get a Boltzmann Machine (Hinton and Sejnowski, 1983). If we restrict the connectivity in a special way, it is easy to learn a Restricted Boltzmann machine.

41 / 58

## Restricted Boltzmann Machines

In an RBM, the hidden units are conditionally independent given the visible states. This enables us to get an unbiased sample from the posterior distribution when given a data-vector.


42 / 58

## Notion of Energies and Probabilities

The probability of a joint configuration over both visible and hidden units depends on the energy of that joint configuration compared with the energy of all other joint configurations.

$$p(v, h) = \frac{e^{-E(v, h)}}{\sum_{u, g} e^{-E(u, g)}}$$

partition function



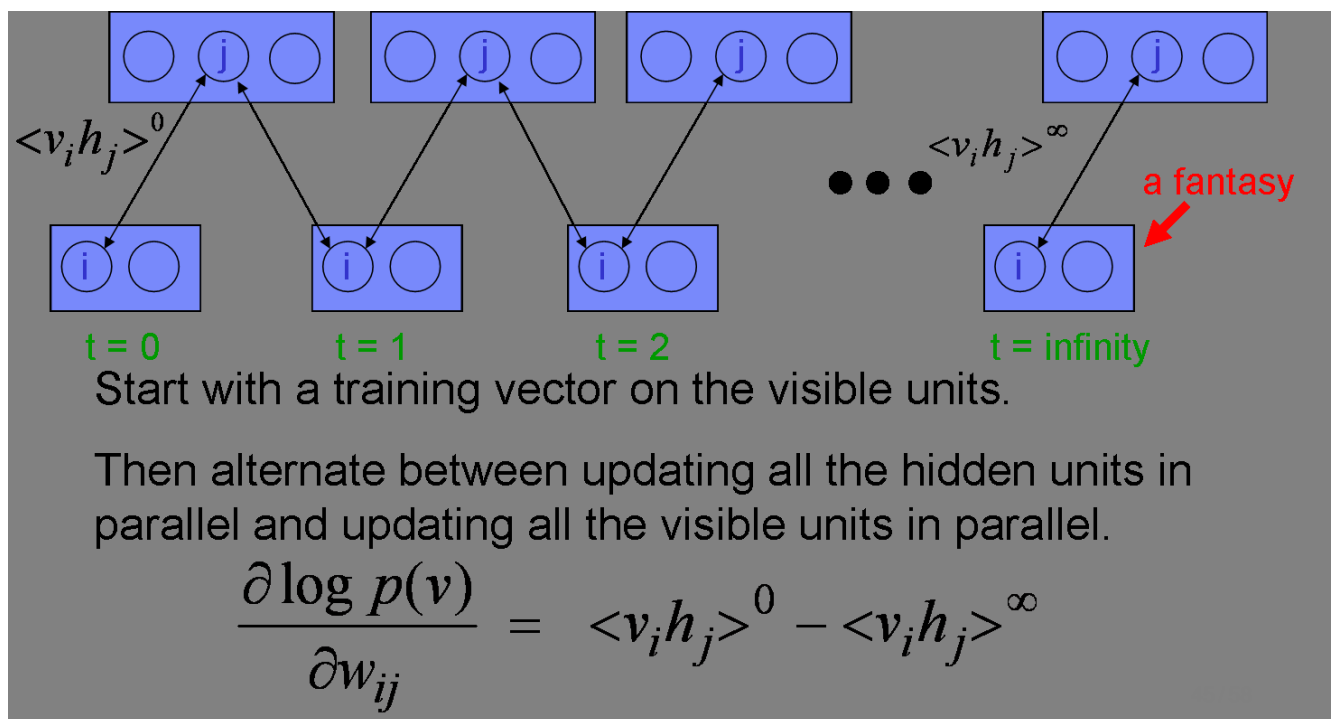
43 / 58

## Notion of Energies and Probabilities

The probability of a configuration of the visible units is the sum of the probabilities of all the joint configurations that contain it.

44 / 58

# A Maximum Likelihood Learning Algorithm for an RBM



## Training a deep network

- First train a layer of features that receive input directly from the audio.
- Then treat the activations of the trained features as if they were input features and learn features of features in a second hidden layer.
- It can be proved that each time we add another layer of features we improve a variational lower bound on the log probability of the training data.
- The proof is complicated. But it is based on a neat equivalence between an RBM and a deep directed model

# Training a deep network

- First learn one layer at a time greedily. Then treat this as pre-training that finds a good initial set of weights which can be fine-tuned by a local search procedure.
- Contrastive wake-sleep is one way of fine-tuning the model to be better at generation.
- Backpropagation can be used to fine-tune the model for better discrimination.

47 / 58

## Why does it work?

- Greedily learning one layer at a time scales well to really big networks, especially if we have locality in each layer.
- We do not start backpropagation until we already have sensible feature detectors that should already be very helpful for the discrimination task. So the initial gradients are sensible and backprop only needs to perform a local search from a sensible starting point.

48 / 58



## Another view

- Most of the information in the final weights comes from modeling the distribution of input vectors.
- The input vectors generally contain a lot more information than the labels.
- The precious information in the labels is only used for the final fine-tuning.
- The fine-tuning only modifies the features slightly to get the category boundaries right. It does not need to discover features.
- This type of backpropagation works well even if most of the training data is unlabeled. The unlabeled data is still very useful for discovering good features.

49 / 58

## In speech recognition...

- We know with GMM/HMMs, increasing the number of context-dependent states (i.e., classes) improves performance
- MLPs typically trained with small number of output targets increasing output targets becomes a harder optimization problem and does not always improve WER
- It increases parameters and increases training time
- With DBNs, pre-training putting weights in better space, and thus we can increase output targets effectively

50 / 58

## Performance of DBNs

Number of Targets	WER
384	21.3
512	20.8
1024	19.4
2,220	<b>18.5</b>

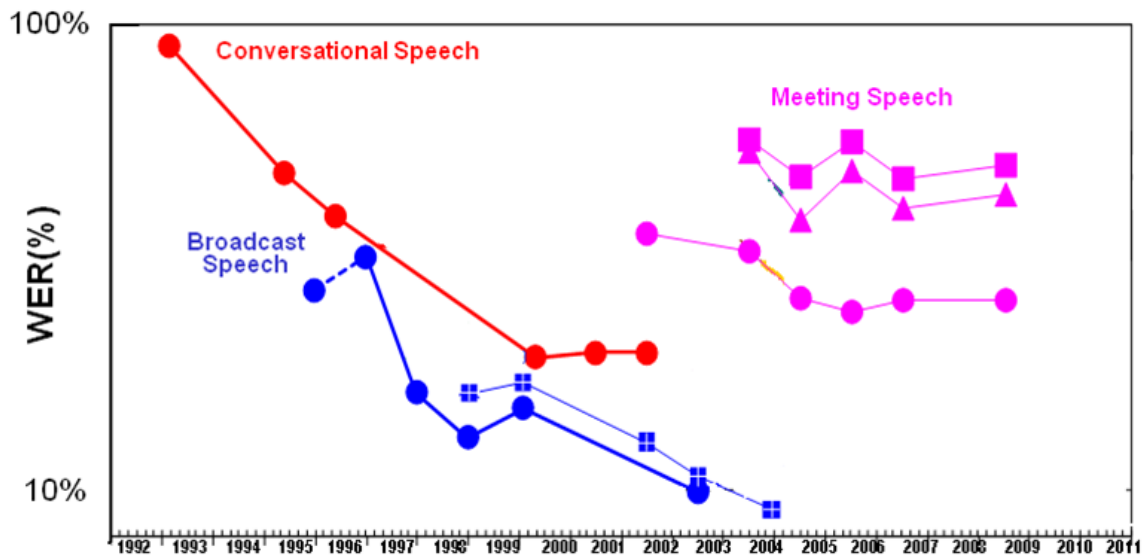
51/58

## LVCSR Performance

	50 Hour English Broadcast News	300 Hour Switchboard Telephony	400 Hour English Broadcast News
GMM/HMM	17.2	14.3	16.5
DBN	<b>14.9</b>	<b>12.2</b>	<b>15.2</b>
% Relative Improvement	<b>13.4</b>	<b>14.7</b>	<b>7.9</b>

52/58

# Historical Performance



53/58

## Issues with DBNs

- Training time!!
- Architecture: Context of 11 Frames, 2,048 Hidden Units, 5 layers, 9,300 output targets implies 43 million parameters !!
- Training time on 300 hours (100 M frames) of data takes 30 days on one 12 core CPU)!
- Compare to a GMM/HMM system with 16 M parameters that takes roughly 2 days to train!!
- Need to speed up.

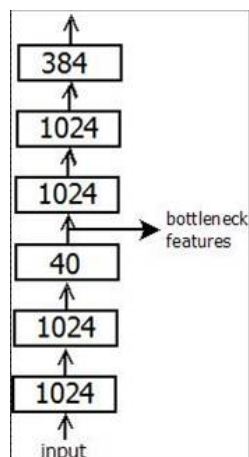
54/58

# One way to speed up..

- One reason DBN training is slow is because we use a large number of output targets (context dependent targets)
- Bottleneck feature DBNs generally have few output targets - these are features we extract to train GMMs on them.
- We can use standard GMM processing techniques on these features

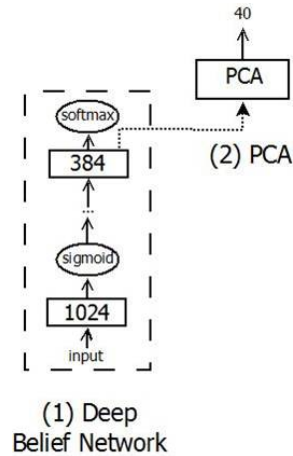
55 / 58

## Example bottleneck feature extraction



56 / 58

# Example bottleneck feature extraction



57 / 58

## What's in the future?

- Better pre training ( parallelization of gradient and larger batches)
- Better Bottleneck Features
- Convolutional neural Networks (LeCunn et al)

58 / 58