Deep Learning

- Early Work
- Why Deep Learning
- Stacked Auto Encoders
- Deep Belief Networks
Deep Learning Overview

- Train networks with many layers (vs. shallow nets with just a couple of layers)

- Multiple layers work to build an improved feature space
  - First layer learns 1st order features (e.g. edges…)
  - 2nd layer learns higher order features (combinations of first layer features, combinations of edges, etc.)
  - In current models layers often learn in an unsupervised mode and discover general features of the input space – serving multiple tasks related to the unsupervised instances (image recognition, etc.)
  - Then final layer features are fed into supervised layer(s)
    - And entire network is often subsequently tuned using supervised training of the entire net, using the initial weightings learned in the unsupervised phase
  - Could also do fully supervised versions, etc. (early BP attempts)
Deep Learning Tasks

- Usually best when input space is locally structured – spatial or temporal: images, language, etc. vs arbitrary input features
- Images Example: view of a learned vision feature layer (Basis)
- Each square in the figure shows the input image that maximally activates one of the 100 units
Why Deep Learning

- Biological Plausibility – e.g. Visual Cortex
- Hastad proof - Problems which can be represented with a polynomial number of nodes with $k$ layers, may require an exponential number of nodes with $k-1$ layers (e.g. parity)
- Highly varying functions can be efficiently represented with deep architectures
  - Less weights/parameters to update than a less efficient shallow representation
- Sub-features created in deep architecture can potentially be shared between multiple tasks
  - Type of Transfer/Multi-task learning
Early Work

- Fukushima (1980) – Neo-Cognitron
- LeCun (1998) – Convolutional Neural Networks (CNN)
  - Similarities to Neo-Cognitron
- Many layered MLP with backpropagation
  - Tried early but without much success
    - Very slow
    - Diffusion of gradient
  - Very recent work has shown significant accuracy improvements by "patiently" training deeper MLPs with BP using fast machines (GPUs) – may be best most general approach
    - We are seeking improvements to basic BP algorithm
  - We will focus on deep networks with unsupervised early layers after a review of CNNs
BP Training Problems

• Error attenuation, long fruitless training
• Recent – Long patient training with GPUs and special hardware
• Becoming more popular, Rectified Linear Units
Rectified Linear Units

- More efficient gradient propagation, derivative is 0 or constant, just fold into learning rate
- More efficient computation: Only comparison, addition and multiplication.
  - Leaky ReLU $f(x) = x$ if $> 0$ else $ax$ where $0 \leq a \leq 1$, so that derivate is not 0 and can do some learning for that case.
  - Lots of other variations
- Sparse activation: For example, in a randomly initialized networks, only about 50% of hidden units are activated (having a non-zero output)
Convolutional Neural Networks

C layers are convolutions, S layers pool/sample.

Often starts with fairly raw features at initial input and lets CNN discover improved feature layer for final supervised learner – e.g., MLP/BP.
Convolutional Neural Networks

Networks built specifically for problems with low dimensional (e.g. 2-d) local structure

- Character recognition – where neighboring pixels will have high correlations and local features (edges, corners, etc.), while distant pixels (features) are un-correlated
- Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part
- While standard NN nodes take input from all nodes in the previous layer, CNNs enforce that a node receives only a small set of features which are spatially or temporally close to each other, called receptive fields from one layer to the next (e.g. 3x3, 5x5), thus enforcing ability to handle local 2-D structure.
  - Can find edges, corners, endpoints, etc.
  - Good for problems with local 2-D structure, but lousy for general learning with abstract features having no prescribed ordering
CNN – Translation Invariance

- The 2-d planes of nodes (or their outputs) at subsequent layers in a CNN are called feature maps.
- To deal with translation invariance, each node in a feature map has the same weights (based on the feature it is looking for), and each node connects to a different overlapping receptive field of the previous layer.
- Thus each feature map searches the full previous layer to see if and how often its feature occurs (precise position not critical).
  - The output will be high at each node in the map corresponding to a receptive field where the feature occurs.
  - Later layers could concern themselves with higher order combinations of features and rough relative positions.
  - Each calculation of a node’s net value, \( \Sigma xw \) in the feature map, is called a convolution, based on the similarity to standard overlapping convolutions.
Each node (e.g. convolution) is calculated for each receptive field in the previous layer

- During training the corresponding weights are always tied to be the same (ala BPTT)
- Thus a relatively small number of unique weight parameters to learn, although they are replicated many times in the feature map
- Each node output in CNN is \( \text{sigmoid}(\sum xw + b) \) (just like BP)
- Multiple feature maps in each layer
- Each feature map should learn a different translation invariant feature

Convolution layer causes total number of features to increase
Sub-Sampling (Pooling)

- Convolution and sub-sampling layers are interleaved
- Sub-sampling (Pooling) allows number of features to be diminished, non-overlapped
  - Reduces spatial resolution and thus naturally decreases importance of exactly where a feature was found, just keeping the rough location
  - Averaging or Max-Pooling (Just as long as the feature is there, take the max, as exact position is not that critical)
  - 2x2 pooling would do 4:1 compression, 3x3 9:1, etc.
  - Pooling smooths the data and makes the data invariant to small translational changes
  - Since after first layer, there are always multiple feature maps to connect to the next layer, it is a pre-made human decision as to which previous maps the current map receives inputs from
CNN Training

- Trained with BP but with weight tying in each feature map
  - Randomized initial weights through entire network
  - Just average the weight updates over the tied weights in feature map layers

- Convolution layer
  - Each feature map has one weight for each input and one bias
  - Thus a feature map with a 5x5 receptive field would have a total of 26 weights, which are the same coming into each node of the feature map
  - If a convolution layer had 10 feature maps, then only a total of 260 unique weights to be trained in that layer (much less than an arbitrary deep net layer without sharing)

- Sub-Sampling (Pooling) Layer
  - All elements of receptive field max’d, averaged, or summed, result multiplied by one trainable weight and a bias added, then squashed for each pooling node
  - If a layer had 10 pooling feature maps, then 20 unique weights to be trained

- While all weights are trained, the structure of the CNN is currently usually hand crafted with trial and error.
  - Number of total layers, number of receptive fields, size of receptive fields, size of sub-sampling (pooling) fields, which fields of the previous layer to connect to
  - Typically decrease size of feature maps and increase number of feature maps for later layers
Example – LeNet-5

To help it all sink in:

- How many weights to be trained at each layer?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I**

Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3.

---

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.
Why 32x32 to start with? Actual characters never bigger than 28x28. Just padding the edges so for example
the top corner node of the feature map can have a pad of two up and left for its feature map.

Same things happens with 14x14 to 10x10 drop from S2 to C3

C3: 6 maps connect to 3, 9 to 4, and 1 to all 6. Forces discovery of more diverse feature combinations. Table
1 only considers contiguous subsets of 3, and more mixed subsets of 4 feature maps, and one with all –
heuristic attempt

LeCun used a special RBF output approach in his LeCun-5 model. Could commonly have just gone into an
output layer at F6 with 10 output nodes. Then would have been 10*(120+1) = 1210 weights going to the last
output layer

~340,000 total connections, with 60,000 trainable parameters 97% of which are in the final MLP

<table>
<thead>
<tr>
<th>Layer</th>
<th>Trainable Weights</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>(25+1)*6 = 156</td>
<td>(25+1)<em>6</em>28*28 = 122,304</td>
</tr>
<tr>
<td>S2</td>
<td>(1+1)*6 = 12</td>
<td>(4+1)<em>6</em>14*14 = 5880 (2x2 links and bias)</td>
</tr>
<tr>
<td>C3</td>
<td>6*(25<em>3+1) + 9</em>(25<em>4+1) + 1</em>(25*6+1) = 1516</td>
<td>1516<em>10</em>10 = 151,600</td>
</tr>
<tr>
<td>S4</td>
<td>16*2 = 32</td>
<td>16<em>5</em>5*5 = 2000 (2x2 links and bias)</td>
</tr>
<tr>
<td>C5</td>
<td>120*(5<em>5</em>16+1) = 48,120</td>
<td>Same since fully connected MLP at this point</td>
</tr>
<tr>
<td>F6</td>
<td>84*(120+1) = 10,164</td>
<td>Same</td>
</tr>
<tr>
<td>Output</td>
<td>10*(84+1) = 850 (RBF)</td>
<td>Same</td>
</tr>
</tbody>
</table>
CNN Summary

- High accuracy for image applications
- Special purpose net – Just for images or problems with strong local spatial/temporal correlation
- Lots of hand crafting and CV tuning to find the right recipe of receptive fields, layer interconnections, etc.
  - Lots more Hyperparameters than standard nets, and even than other deep networks, since the structures of CNNs are so handcrafted
  - Recent research proposing simpler more consistent structure also effective. If 5x5, 2x2, depth just a function of initial image and pooling field (2x2), thus main parameters are only number of feature maps per layer, and MLP hidden nodes
  - CNNs getting wider and deeper with speed-up techniques (e.g. ReLU)
- Drop-out often used for overfit avoidance
- Advanced convolution (e.g. NiN) and Pooling approaches
Training Deep Networks

- Build a feature space
  - Note that this is what we do with SVM kernels, or trained hidden layers in BP, etc., but now we will build the feature space using deep architectures
  - Unsupervised training between layers can decompose the problem into distributed sub-problems (with higher levels of abstraction) to be further decomposed at subsequent layers
Training Deep Networks

- Difficulties of supervised training of deep networks
  - Early layers of MLP do not get trained well
    - Diffusion of Gradient – error attenuates as it propagates to earlier layers
    - Leads to very slow training
    - Exacerbated since top couple layers can usually learn any task "pretty well" and thus the error to earlier layers drops quickly as the top layers "mostly" solve the task– lower layers never get the opportunity to use their capacity to improve results, they just do a random feature map
  - Need a way for early layers to do effective work
  - Instability of gradient in deep networks: Vanishing or exploding gradient
    - Product of many terms, which unless “balanced” just right, is unstable
    - Either early or late layers stuck while “opposite” layers are learning
  - Often not enough labeled data available while there may be lots of unlabeled data
    - Can we use unsupervised/semi-supervised approaches to take advantage of the unlabeled data
  - Deep networks tend to have more sensitive training issues problems than shallow networks during supervised training
Greedy Layer-Wise Training

One answer is greedy layer-wise training

1. Train first layer using your data without the labels (unsupervised)
   - Since there are no targets at this level, labels don't help. Could also use the more abundant unlabeled data which is not part of the training set (i.e. self-taught learning).

2. Then freeze the first layer parameters and start training the second layer using the output of the first layer as the unsupervised input to the second layer

3. Repeat this for as many layers as desired
   - This builds our set of robust features

4. Use the outputs of the final layer as inputs to a supervised layer/model and train the last supervised layer(s) (leave early weights frozen)

5. Unfreeze all weights and fine tune the full network by training with a supervised approach, given the pre-training weight settings
Deep Net with Greedy Layer Wise Training

ML Model

New Feature Space

Original Inputs

Supervised Learning

Unsupervised Learning
Greedy layer-wise training avoids many of the problems of trying to train a deep net in a supervised fashion

- Each layer gets full learning focus in its turn since it is the only current "top" layer
- Can take advantage of unlabeled data
- When you finally tune the entire network with supervised training the network weights have already been adjusted so that you are in a good error basin and just need fine tuning. This helps with problems of
  - Ineffective early layer learning
  - Deep network local minima

We will discuss the two most common approaches

- Stacked Auto-Encoders
- Deep Belief Networks
Self Taught vs Unsupervised Learning

- When using Unsupervised Learning as a pre-processor to supervised learning you are typically given examples from the same distribution as the later supervised instances will come from
  - Assume the distribution comes from a set containing just examples from a defined set up possible output classes, but the label is not available (e.g. images of car vs trains vs motorcycles)

- In Self-Taught Learning we do not require that the later supervised instances come from the same distribution
  - e.g., Do self-taught learning with any images, even though later you will do supervised learning with just cars, trains and motorcycles.
  - These types of distributions are more readily available than ones which just have the classes of interest
  - However, if distributions are very different…

- New tasks share concepts/features from existing data and statistical regularities in the input distribution that many tasks can benefit from
  - Note similarities to supervised multi-task and transfer learning

- Both approaches reasonable in deep learning models
Auto-Encoders

- A type of unsupervised learning which tries to discover generic features of the data
  - Learn identity function by learning important sub-features (not by just passing through data)
  - Compression, etc.
  - Can use just new features in the new training set or concatenate both
Stacked Auto-Encoders

- Stack many (sparse) auto-encoders in succession and train them using greedy layer-wise training
- Drop the decode output layer each time
Stacked Auto-Encoders

- Do supervised training on the last layer using final features
- Then do supervised training on the entire network to fine-tune all weights
Sparse Encoders

- Auto encoders will often do a dimensionality reduction
  - PCA-like or non-linear dimensionality reduction
- This leads to a "dense" representation which is nice in terms of parsimony
  - All features typically have non-zero values for any input and the combination of values contains the compressed information
- However, this distributed and entangled representation can often make it more difficult for successive layers to pick out the salient features
- A *sparse* representation uses more features where at any given time many/most of the features will have a 0 value
  - Thus there is an implicit compression each time but with varying nodes
  - This leads to more localist variable length encodings where a particular node (or small group of nodes) with value 1 signifies the presence of a feature (small set of bases)
  - A type of simplicity bottleneck (regularizer)
  - This is easier for subsequent layers to use for learning
How do we implement a sparse Auto-Encoder?

- Use more hidden nodes in the encoder
- Use regularization techniques which encourage sparseness (e.g. a significant portion of nodes have 0 output for any given input)
  - Penalty in the learning function for non-zero nodes
  - Weight decay
  - etc.

De-noising Auto-Encoder
- Stochastically corrupt training instance each time, but still train auto-encoder to decode the uncorrupted instance, forcing it to learn conditional dependencies within the instance
- Better empirical results, handles missing values well
Sparse Representation

- For bases below, which is easier to see intuition for current pattern - if a few of these are on and the rest 0, or if all have some non-zero value?
- Easier to learn if sparse
Stacked Auto-Encoders

- Concatenation approach (i.e. using both hidden features and original features in final (or other) layers) can be better if not doing fine tuning. If fine tuning, the pure replacement approach can work well.
- Always fine tune if there is a sufficient amount of labeled data
- For real valued inputs, MLP training is like regression and thus could use linear output node activations, still sigmoid at hidden
- Stacked Auto-Encoders empirically not quite as accurate as DBNs (Deep Belief Networks)
  - (with De-noising auto-encoders, stacked auto-encoders competitive with DBNs)
  - Not generative like DBNs, though recent work with de-noising auto-encoders may allow generative capacity
Deep Belief Networks (DBN)

- Geoff Hinton (2006)
- Uses Greedy layer-wise training but each layer is an RBM (Restricted Boltzmann Machine)
- RBM is a constrained Boltzmann machine with
  - No lateral connections between hidden \((h)\) and visible \((x)\) nodes
  - Symmetric weights
  - Does not use annealing/temperature, but that is all right since each RBM not seeking a global minima, but rather an incremental transformation of the feature space
  - Typically uses probabilistic logistic node, but other activations possible
RBM Sampling and Training

- Initial state typically set to a training example \( x \) (can be real valued)
- Sampling is an iterative back and forth process
  - \( P(h_i = 1|x) = \text{sigmoid}(W_i x + c_i) = 1/(1+e^{-\text{net}(h_i)}) \) // \( c_i \) is hidden node bias
  - \( P(x_i = 1|h) = \text{sigmoid}(W'_i h + b_i) = 1/(1+e^{-\text{net}(x_i)}) \) // \( b_i \) is visible node bias
- Contrastive Divergence (CD-\( k \)): How much contrast (in the statistical distribution) is there in the divergence from the original training example to the relaxed version after \( k \) relaxation steps
- Then update weights to decrease the divergence as in Boltzmann
- Typically just do CD-1 (Good empirical results)
  - Since small learning rate, doing many of these is similar to doing fewer versions of CD-\( k \) with \( k > 1 \)
  - Note CD-1 just needs to get the gradient direction right, which it usually does, and then change weights in that direction according to the learning rate
RBMupdate($x_1, \epsilon, W, b, c$)

This is the RBM update procedure for binomial units. It can easily adapted to other types of units.

$x_1$ is a sample from the training distribution for the RBM

$\epsilon$ is a learning rate for the stochastic gradient descent in Contrastive Divergence

$W$ is the RBM weight matrix, of dimension (number of hidden units, number of inputs)

$b$ is the RBM offset vector for input units

$c$ is the RBM offset vector for hidden units

Notation: $Q(h_{2i} = 1|x_2)$ is the vector with elements $Q(h_{2i} = 1|x_2)$

for all hidden units $i$ do
  • compute $Q(h_{1i} = 1|x_1)$ (for binomial units, $\text{sigm}(c_i + \sum_j W_{ij} x_{1j})$)
  • sample $h_{1i} \in \{0, 1\}$ from $Q(h_{1i}|x_1)$
end for

for all visible units $j$ do
  • compute $P(x_{2j} = 1|h_1)$ (for binomial units, $\text{sigm}(b_j + \sum_i W_{ij} h_{1i})$)
  • sample $x_{2j} \in \{0, 1\}$ from $P(x_{2j} = 1|h_1)$
end for

for all hidden units $i$ do
  • compute $Q(h_{2i} = 1|x_2)$ (for binomial units, $\text{sigm}(c_i + \sum_j W_{ij} x_{2j})$)
end for

$W \leftarrow W + \epsilon(h_1 x'_1 - Q(h_{2.} = 1|x_2)x'_2)$

$b \leftarrow b + \epsilon(x_1 - x_2)$

$c \leftarrow c + \epsilon(h_1 - Q(h_{2.} = 1|x_2))$

$$
\Delta W_{ij} = \epsilon(h_{1,j} \cdot x_{1,i} - Q(h_{k+1,j} = 1|x_{k+1})x_{k+1,i})
$$
RBM Update Notes and Variations

- Binomial unit means the standard MLP sigmoid unit
- $Q$ and $P$ are probability distribution vectors for hidden ($h$) and visible/input ($x$) vectors respectively
- During relaxation/weight update can alternatively do updates based on the real valued probabilities ($\text{sigmoid}(\text{net})$) rather than the 1/0 sampled logistic states
  - Always use actual/binary values from initial $x \rightarrow h$
    - Doing this makes the hidden nodes a sparser bottleneck and is a regularizer helping to avoid overfit
  - Could use probabilities on the $h \rightarrow x$ and/or final $x \rightarrow h$
    - in CD-$k$ the final update of the hidden nodes usually use the probability value to decrease the final arbitrary sampling variation (sampling noise)
- Lateral restrictions of RBM allow this fast sampling
RBM Update Variations and Notes

- Initial weights, small random, 0 mean, sd ~ .01
  - Don't want hidden node probabilities early on to be close to 0 or 1, else slows learning, since less early randomness/mixing? Note that this is a bit like annealing/temperature in Boltzmann

- Set initial $x$ bias values as a function of how often node is on in the training data, and $h$ biases to 0 or negative to encourage sparsity

- Better speed when using momentum ($\sim .5$)

- Weight decay good for smoothing and also encouraging more mixing (hidden nodes more stochastic when they do not have large net magnitudes)
  - Also a reason to increase $k$ over time in CD-$k$ as mixing decreases as weight magnitudes increase
Deep Belief Network Training

- Same greedy layer-wise approach
- First train lowest RBM ($h^0 - h^1$) using RBM update algorithm (note $h^0$ is $x$)
- Freeze weights and train subsequent RBM layers
- Then connect final outputs to a supervised model and train that model
- Finally, unfreeze all weights, and train full DBN with supervised model to fine-tune weights

During execution typically iterate multiple times at the top RBM layer
Can use DBN as a Generative model to create sample x vectors

1. Initialize top layer to an arbitrary vector
   - Gibbs sample (relaxation) between the top two layers $m$ times
   - If we initialize top layer with values obtained from a training example, then need less Gibbs samples

2. Pass the vector down through the network, sampling with the calculated probabilities at each layer

3. Last sample at bottom is the generated x value (can be real valued if we use the probability vector rather than sample)

Alternatively, can start with an x at the bottom, relax to a top value, then start from that vector when generating a new x, which is the dotted lines version. More like standard Boltzmann machine processing.
Train a DBN in a purely unsupervised way, with the greedy layer-wise procedure in which each added layer is trained as an RBM (e.g. by Contrastive Divergence).

\( \hat{P} \) is the input training distribution for the network
\( \epsilon \) is a learning rate for the RBM training
\( \ell \) is the number of layers to train
\( W^k \) is the weight matrix for level \( k \), for \( k \) from 1 to \( \ell \)
\( b^k \) is the visible units offset vector for RBM at level \( k \), for \( k \) from 1 to \( \ell \)
\( c^k \) is the hidden units offset vector for RBM at level \( k \), for \( k \) from 1 to \( \ell \)

\text{mean-field computation} is a Boolean that is true iff training data at each additional level is obtained by a mean-field approximation instead of stochastic sampling.

\[ \text{for } k = 1 \text{ to } \ell \text{ do} \]
  \[ \text{• initialize } W^k = 0, b^k = 0, c^k = 0 \]
  \[ \text{while not stopping criterion do} \]
    \[ \text{• sample } h^0 = \mathbf{x} \text{ from } \hat{P} \]
    \[ \text{for } i = 1 \text{ to } k - 1 \text{ do} \]
      \[ \text{if mean-field computation then} \]
        \[ \text{• assign } h^i_j \text{ to } Q(h^i_j = 1|h^{i-1}), \text{ for all elements } j \text{ of } h^i \]
      \[ \text{else} \]
        \[ \text{• sample } h^i_j \text{ from } Q(h^i_j|h^{i-1}), \text{ for all elements } j \text{ of } h^i \]
      \[ \text{end if} \]
    \[ \text{end for} \]
    \[ \text{• RBMUpdate}(h^{k-1}, \epsilon, W^k, b^k, c^k) \{\text{thus providing } Q(h^k|h^{k-1}) \text{ for future use}\} \]
  \[ \text{end while} \]
\[ \text{end for} \]
DBN Execution

- After all layers have learned then the output of the last layer can be input to a supervised learning model.
- Note that at this point we could potentially throw away all the downward weights in the network as they will not actually be used during the normal feedforward execution process (as we did with the Stacked Auto Encoder).
  - Note that except for the downward bias weights $b$ they are the same symmetric weights anyways.
  - If we are relaxing $M$ times in the top layer then we would still need the downward weights for that layer.
  - Also if we are generating $x$ values we would need all of them.
- The final weight tuning is usually done with backpropagation, which only updates the feedforward weights, ignoring any downward weights.
RBM stopping criteria still in issue. One common approach is reconstruction error which is the probability that the final $x$ after CD-$k$ is the initial $x$. (i.e. $P(x|E[h|x])$. The other most common approach is AIC (Annealed Importance Sampling). Both have been shown to be problematic.

Each layer updates weights so as to make training sample patterns more likely (lower energy) in the free state (and non-training sample patterns less likely).

This unsupervised approach learns broad features (in the hidden/subsequent layers of RBMs) which can aid in the process of making the types of patterns found in the training set more likely. This discovers features which can be associated across training patterns, and thus potentially helpful for other goals with the training set (classification, compression, etc.)

Note still pairwise weights in RBMs, but because we can pick the number of hidden units and layers, we can represent any arbitrary distribution.
To be consistent just use 28×28 (764) data set of gray scale values (0-255)
- Normalize to 0-1
- Could try better preprocessing if want and helps in published accuracies, but start/stay with this
- Small random initial weights

Parameters
- Hinton Paper, others – do a little searching and e-mail me a reference for extra credit points
- http://yann.lecun.com/exdb/mnist/ for sample approaches

Straight 200 hidden node MLP does quite good ~98%
- Rough Hyperparameters - LR: ~.05-.1, Momentum ~.5

Best class deep net results: ~98.5% - which is competitive
- About half students never beat MLP baseline
- Can you beat the 98.5%?
Deep Learning Project Past Experience

- Structure: ~3 hidden layers, ~500ish nodes/layer, more nodes/layers can be better but training is longer

- Training time:
  - DBN: ~10 epochs with the 60K set, small LR ~.005 often good
  - Can go longer, does not seem to overfit with the large data set
  - SAE: Can saturate/overfit, ~3 epochs good, but will be a function of your denoising approach, which is essential for sparsity, use small LR ~.005, long training – up to 50 hours, got 98.55
    - Larger learning rates often lead low accuracy for both DBN and SAE

- Sampling vs using real probability value in DBN
  - Best results found when using real values vs. sampling
  - Some found sampling on the back-step of learning helps
  - When using sampling, probably requires longer training, but could actually lead to bigger improvements in the long run
  - Typical forward flow real during execution, but could do some sampling on the *m* iterations at the top layer. Some success with back-step at the top layer iteration (most don't do this at all)
  - We need to try/discover better variations
Deep Learning Project Past Experience

- Note: If we held out 50K of the dataset as unsupervised, then deep nets would more readily show noticeable improvement over BP.

- A final full network fine tune with BP always helps
  - But can take 20+ hours

- Key take away – Most actual time spent training with different parameters. Thus, start early, and then you will have time to try multiple long runs to see which variations work. This does not take that much personal time, as you simply start it with some different parameters and go away for a day. If you wait until the last few days, there is no time to do these experiments.
DBN Notes

- Can use lateral connections in RBM (no longer RBM) but sampling becomes more difficult – ala standard Boltzmann requiring longer sampling chains.
  - Lateral connections can capture pairwise dependencies allowing the hidden nodes to focus on higher order issues. Can get better results.

- Deep Boltzmann machine – Allow continual relaxation across the full network
  - Receive input from above and below rather than sequence through RBM layers
  - Typically for successful training, first initialize weights using the standard greedy DBN training with RBM layers
  - Requires longer sampling chains ala Boltzmann

- Conditional and Temporal RBMs – allow node probabilities to be conditioned by some other inputs – context, recurrence (time series changes in input and internal state), etc.
Discrimination with Deep Networks

- Discrimination approaches with DBNs (Deep Belief Net)
  - Use outputs of DBNs as inputs to supervised model (i.e. just an unsupervised preprocessor for feature extraction)
    - Basic approach we have been discussing
  - Train a DBN for each class. For each clamp the unknown x and iterate $m$ times. The DBN that ends with the lowest normalized free energy (softmax variation) is the winner.
  - Train just one DBN for all classes, but with an additional visible unit for each class. For each output class:
    - Clamp the unknown x, relax, and then see which final state has lowest free energy – no need to normalize since all energies come from the same network.

See [http://deeplearning.net/demos/](http://deeplearning.net/demos/)
Conclusion

- Much recent excitement, still much to be discovered
- "Google-Brain"
- Sum of Products Nets
- Biological Plausibility
- Potential for significant improvements
- Good in structured/Markovian spaces
  - Important research question: To what extent can we use Deep Learning in more arbitrary feature spaces?
  - Recent deep training of MLPs with BP shows potential in this area