10-701 Introduction to Machine Learning

Deep Learning

Readings:
Bishop Ch. 4.1.7, Ch. 5
Murphy Ch. 16.5, Ch. 28
Mitchell Ch. 4

Matt Gormley
Lecture 13
October 19, 2016
Reminders

• Homework 3:
  – due 10/24/16
Outline

• Deep Neural Networks (DNNs)
  – Three ideas for training a DNN
  – Experiments: MNIST digit classification
  – Autoencoders
  – Pretraining

• Recurrent Neural Networks (RNNs)
  – Bidirectional RNNs
  – Deep Bidirectional RNNs
  – Deep Bidirectional LSTMs
  – Connection to forward-backward algorithm

• Convolutional Neural Networks (CNNs)
  – Convolutional layers
  – Pooling layers
  – Image recognition
PRE-TRAINING FOR DEEP NETS
A Recipe for Machine Learning

1. Given training data:

2. Choose each of these:
   - Decision function
   - Loss function

3. Define goal:

4. Train with SGD:
   
   \[ \theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]

Goals for Today’s Lecture

1. Explore a new class of decision functions (Deep Neural Networks)

2. Consider variants of this recipe for training
Idea #1: No pre-training

- Idea #1: (Just like a shallow network)
  - Compute the supervised gradient by backpropagation.
  - Take small steps in the direction of the gradient (SGD)
Comparison on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)

% Error

Shallow Net  Idea #1 (Deep Net, no-pretraining)  Idea #2 (Deep Net, supervised pre-training)  Idea #3 (Deep Net, unsupervised pre-training)
Comparison on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
Idea #1: No pre-training

- **Idea #1: (Just like a shallow network)**
  - Compute the supervised gradient by backpropagation.
  - Take small steps in the direction of the gradient (SGD)

- **What goes wrong?**
  - A. Gets stuck in local optima
    - Nonconvex objective
    - Usually start at a random (bad) point in parameter space
  - B. Gradient is progressively getting more dilute
    - “Vanishing gradients”
Where does the nonconvexity come from?

Even a simple quadratic $z = xy$ objective is nonconvex:
Problem A: Nonconvexity

- Where does the nonconvexity come from?
- Even a simple quadratic $z = xy$ objective is nonconvex:
Stochastic Gradient Descent...

...climbs to the top of the nearest hill...
Training

Stochastic Gradient Descent...

...climbs to the top of the nearest hill...

Problem A: Nonconvexity
Stochastic Gradient Descent...

...climbs to the top of the nearest hill...
Training

Problem A: Nonconvexity

Stochastic Gradient Descent...

...climbs to the top of the nearest hill...
Stochastic Gradient Descent...

...climbs to the top of the nearest hill...

...which might not lead to the top of the mountain

Problem A: Nonconvexity
The gradient for an edge at the base of the network depends on the gradients of many edges above it.

The chain rule multiplies many of these partial derivatives together.
Problem B: Vanishing Gradients

The gradient for an edge at the base of the network depends on the gradients of many edges above it.

The chain rule multiplies many of these partial derivatives together.
Problem B: Vanishing Gradients

The gradient for an edge at the base of the network depends on the gradients of many edges above it.

The chain rule multiplies many of these partial derivatives together.
Idea #1: No pre-training

• Idea #1: (Just like a shallow network)
  • Compute the supervised gradient by backpropagation.
  • Take small steps in the direction of the gradient (SGD)

• What goes wrong?
  A. Gets stuck in local optima
     • Nonconvex objective
     • Usually start at a random (bad) point in parameter space
  B. Gradient is progressively getting more dilute
     • “Vanishing gradients”
Training

Idea #2: Supervised Pre-training

• Idea #2: (Two Steps)
  • Train each level of the model in a greedy way
  • Then use our original idea

1. Supervised Pre-training
   – Use labeled data
   – Work bottom-up
     • Train hidden layer 1. Then fix its parameters.
     • Train hidden layer 2. Then fix its parameters.
     • ...
     • Train hidden layer n. Then fix its parameters.

2. Supervised Fine-tuning
   – Use labeled data to train following “Idea #1”
   – Refine the features by backpropagation so that they become tuned to the end-task
Idea #2: Supervised Pre-training

- Idea #2: (Two Steps)
  - Train each level of the model in a greedy way
  - Then use our original idea
Idea #2: (Two Steps)
- Train each level of the model in a greedy way
- Then use our original idea
Idea #2: Supervised Pre-training

- Idea #2: (Two Steps)
  - Train each level of the model in a greedy way
  - Then use our original idea

Input

Hidden Layer 1

Hidden Layer 2

Hidden Layer 3

Output
Idea #2: (Two Steps)
- Train each level of the model in a greedy way
- Then use our original idea...
Comparison on MNIST

• Results from Bengio et al. (2006) on MNIST digit classification task
• Percent error (lower is better)
• Results from Bengio et al. (2006) on MNIST digit classification task
• Percent error (lower is better)
Training

Idea #3: Unsupervised Pre-training

- **Idea #3: (Two Steps)**
  - Use our original idea, but **pick a better starting point**
  - **Train each level** of the model in a **greedy** way

1. **Unsupervised Pre-training**
   - Use **unlabeled** data
   - Work bottom-up
     - Train hidden layer 1. Then fix its parameters.
     - Train hidden layer 2. Then fix its parameters.
     - ...
     - Train hidden layer n. Then fix its parameters.

2. **Supervised Fine-tuning**
   - Use **labeled** data to train following “Idea #1”
   - Refine the features by backpropagation so that they become tuned to the end-task
Unsupervised pre-training of the first layer:

• What should it predict?
• What else do we observe?
• The input!
Unsupervised pre-training of the first layer:

• What should it predict?
• What else do we observe?
• The input!

This topology defines an Auto-encoder.
Auto-Encoders

Key idea: Encourage \( z \) to give small reconstruction error:

- \( x' \) is the reconstruction of \( x \)
- Loss = \( \| x - \text{DECODER(ENCODER(x))} \|^2 \)
- Train with the same backpropagation algorithm for 2-layer Neural Networks with \( x_m \) as both input and output.

DECODER: \( x' = h(W'z) \)

ENCODER: \( z = h(Wx) \)
Unsupervised pre-training

- Work bottom-up
  - Train hidden layer 1. Then fix its parameters.
  - Train hidden layer 2. Then fix its parameters.
  - ...
  - Train hidden layer n. Then fix its parameters.
Unsupervised pre-training

• Work bottom-up
  – Train hidden layer 1. Then fix its parameters.
  – Train hidden layer 2. Then fix its parameters.
  – …
  – Train hidden layer n. Then fix its parameters.
The solution: 
Unsupervised pre-training

Unsupervised pre-training

• Work bottom-up
  – Train hidden layer 1. Then fix its parameters.
  – Train hidden layer 2. Then fix its parameters.
  – ...
  – Train hidden layer n. Then fix its parameters.
The solution: Unsupervised pre-training

Unsupervised pre-training

• Work bottom-up
  – Train hidden layer 1. Then fix its parameters.
  – Train hidden layer 2. Then fix its parameters.
  – ...
  – Train hidden layer n. Then fix its parameters.

Supervised fine-tuning
Backprop and update all parameters
Deep Network Training

• **Idea #1:**
  1. Supervised fine-tuning only

• **Idea #2:**
  1. Supervised layer-wise pre-training
  2. Supervised fine-tuning

• **Idea #3:**
  1. Unsupervised layer-wise pre-training
  2. Supervised fine-tuning
Comparison on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
• Results from Bengio et al. (2006) on MNIST digit classification task
• Percent error (lower is better)
Is layer-wise pre-training always necessary?

In 2010, a record on a hand-writing recognition task was set by standard supervised backpropagation (our Idea #1).

How? A very fast implementation on GPUs.

See Ciresen et al. (2010)
• Goal: learn features at different levels of abstraction

• Training can be tricky due to...
  – Nonconvexity
  – Vanishing gradients

• Unsupervised layer-wise pre-training can help with both!
RECURRENT NEURAL NETWORKS
Recurrent Neural Networks (RNNs)

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
  h_t &= \mathcal{H} (W_{xh} x_t + W_{hh} h_{t-1} + b_h) \\
  y_t &= W_{hy} h_t + b_y
\end{align*}
\]
Recurrent Neural Networks (RNNs)

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( h = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:

\[
\begin{align*}
h_t &= \mathcal{H} (W_{xh} x_t + W_{hh} h_{t-1} + b_h) \\
y_t &= W_{hy} h_t + b_y
\end{align*}
\]
Recurrent Neural Networks (RNNs)

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
h_t &= \mathcal{H} \left( W_{xh} x_t + W_{hh} h_{t-1} + b_h \right) \\
y_t &= W_{hy} h_t + b_y
\end{align*}
\]
Recurrent Neural Networks (RNNs)

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
  h_t &= \mathcal{H} \left( W_{xh} x_t + W_{hh} h_{t-1} + b_h \right) \\
  y_t &= W_{hy} h_t + b_y
\end{align*}
\]

- If \( T=1 \), then we have a standard feed-forward neural net with one hidden layer
- All of the deep nets from last lecture (DNN, DBN, DBM) required fixed size inputs/outputs
Recurrent Neural Networks (RNNs)

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( h = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
    h_t &= \mathcal{H} (W_{xh} x_t + W_{hh} h_{t-1} + b_h) \\
y_t &= W_{hy} h_t + b_y
\end{align*}
\]

- By unrolling the RNN through time, we can share parameters and accommodate arbitrary length input/output pairs
- Applications: time-series data such as sentences, speech, stock-market, signal data, etc.
Background: Backprop through time

**Recurrent neural network:**

1. Unroll the computation over time

   \[ y_{t+1}, x_{t+1}, b_t, a, x_t \]

**BPTT:**

2. Run backprop through the resulting feedforward network

   (Robinson & Fallside, 1987)  
   (Werbos, 1988)  
   (Mozer, 1995)
Bidirectional RNN

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} \) and \( \mathbf{h} \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\begin{align*}
\mathbf{h}_t &= \mathcal{H} \left( W_{x_h} x_t + W_{h_h} \mathbf{h}_{t-1} + b_h \right) \\
\mathbf{h}_t &= \mathcal{H} \left( W_{x_h} x_t + W_{h_h} \mathbf{h}_{t+1} + b_h \right) \\
y_t &= W_{h_y} \mathbf{h}_t + W_{h_y} \mathbf{h}_t + b_y 
\end{align*}
\]
Bidirectional RNN

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathbb{R}^I \)

hidden units: \( \vec{h} \) and \( \overleftarrow{h} \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathbb{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\vec{h}_t = \mathcal{H} \left( W_{x \vec{h}} x_t + W_{\vec{h} \vec{h}} \overrightarrow{h}_{t-1} + b_{\vec{h}} \right)
\]

\[
\overleftarrow{h}_t = \mathcal{H} \left( W_{x \overleftarrow{h}} x_t + W_{\overleftarrow{h} \overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}} \right)
\]

\[
y_t = W_{\vec{h} y} \overrightarrow{h}_t + W_{\overleftarrow{h} y} \overleftarrow{h}_t + b_y
\]
Bidirectional RNN

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \vec{h} \) and \( \hat{h} \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\vec{h}_t = \mathcal{H} \left( W_{x \vec{h}} x_t + W_{\vec{h} \vec{h}} \vec{h}_{t-1} + b_{\vec{h}} \right)
\]

\[
\hat{h}_t = \mathcal{H} \left( W_{x \hat{h}} x_t + W_{\hat{h} \hat{h}} \hat{h}_{t+1} + b_{\hat{h}} \right)
\]

\[
y_t = W_{\vec{h} y} \vec{h}_t + W_{\hat{h} y} \hat{h}_t + b_y
\]
Bidirectional RNN

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)
hidden units: \( \overrightarrow{h} \) and \( \overleftarrow{h} \)
outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)
nonlinearity: \( \mathcal{H} \)

Recursive Definition:
\[
\overrightarrow{h}_t = \mathcal{H} \left( W_{x\rightarrow h} x_t + W_{h\rightarrow h} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}} \right)
\]
\[
\overleftarrow{h}_t = \mathcal{H} \left( W_{x\leftarrow h} x_t + W_{h\leftarrow h} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}} \right)
\]
\[
y_t = W_{h\rightarrow y} \overrightarrow{h}_t + W_{h\leftarrow y} \overleftarrow{h}_t + b_y
\]

Is there an analogy to some other recursive algorithm(s) we know?
Deep RNNs

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)
outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)
nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\begin{align*}
  h_t^n &= \mathcal{H} \left( W_{h_{t-1} h_t} h_{t-1}^{n-1} + W_{h_t h_t} h_t^{n-1} + b_h^n \right) \\
  y_t &= W_{h_N y} h_t^N + b_y
\end{align*}
\]

![Diagram of Deep RNNs](Image)

Figure from (Graves et al., 2013)
Deep Bidirectional RNNs

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

- Notice that the upper level hidden units have input from **two previous layers** (i.e. wider input)
- Likewise for the output layer
- What analogy can we draw to DNNs, DBNs, DBMs?

Figure from (Graves et al., 2013)
Long Short-Term Memory (LSTM)

Motivation:

• Standard RNNs have trouble learning long distance dependencies
• LSTMs combat this issue
Long Short-Term Memory (LSTM)

Motivation:
• Vanishing gradient problem for Standard RNNs
• Figure shows sensitivity (darker = more sensitive) to the input at time t=1

Figure from (Graves, 2012)
Long Short-Term Memory (LSTM)

Motivation:
• LSTM units have a rich internal structure
• The various “gates” determine the propagation of information and can choose to “remember” or “forget” information

Figure 4.4: Preservation of gradient information by LSTM. As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all gates are either entirely open (‘O’) or closed (‘—’). The memory cell ‘remembers’ the first input as long as the forget gate is open and the input gate is closed. The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

4.2 Influence of Preprocessing
The above discussion raises an important point about the influence of preprocessing. If we can find a way to transform a task containing long range contextual dependencies into one containing only short-range dependencies before presenting it to a sequence learning algorithm, then architectures such as LSTM become somewhat redundant. For example, a raw speech signal typically has a sampling rate of over 40 kHz. Clearly, a great many timesteps would have to be spanned by a sequence learning algorithm attempting to label or model an utterance presented in this form. However when the signal is first transformed into a 100 Hz series of mel-frequency cepstral coefficients, it becomes feasible to model the data using an algorithm whose contextual range is relatively short, such as a hidden Markov model.

Nonetheless, if such a transform is difficult or unknown, or if we simply wish to get a good result without having to design task-specific preprocessing methods, algorithms capable of handling long time dependencies are essential.

4.3 Gradient Calculation
Like the networks discussed in the last chapter, LSTM is a differentiable function approximator that is typically trained with gradient descent. Recently, non-gradient-based training methods of LSTM have also been considered (Wierstra et al., 2005; Schmidhuber et al., 2007), but they are outside the scope of this book.

Figure from (Graves, 2012)
Long Short-Term Memory (LSTM)

![LSTM Network Diagram]
Long Short-Term Memory (LSTM)

- **Input gate:** masks out the standard RNN inputs
- **Forget gate:** masks out the previous cell
- **Cell:** stores the input/forget mixture
- **Output gate:** masks out the values of the next hidden

\[
\begin{align*}
  i_t &= \sigma \left( W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i \right) \\
  f_t &= \sigma \left( W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f \right) \\
  c_t &= f_t c_{t-1} + i_t \tanh \left( W_{xc}x_t + W_{hc}h_{t-1} + b_c \right) \\
  o_t &= \sigma \left( W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o \right) \\
  h_t &= o_t \tanh(c_t)
\end{align*}
\]

Figure from (Graves et al., 2013)
Long Short-Term Memory (LSTM)
Deep Bidirectional LSTM (DBLSTM)

- Figure: input/output layers not shown
- **Same general topology** as a Deep Bidirectional RNN, but with LSTM units in the hidden layers
- No additional representational power over DBRNN, but **easier to learn** in practice
3. NETWORK TRAINING

Network training follows the standard approach used in hybrid systems [4]. Frame-level state targets are provided on the training set by a forced alignment given by a GMM-HMM system. The network is then trained to minimise the cross-entropy error of the targets using a softmax output layer with as many units as the total number of possible HMM states. At decoding time, the state probabilities yielded by the network are combined with a dictionary and language model to determine the most probable transcription.

For a length $T$ acoustic sequence $x$ the network produces a length $T$ output sequence $y$, where each $y_t$ defines a probability distribution over the $K$ possible states: that is, $y_{k,t}$ (the $k$th element of $y_t$) is the network's estimate for the probability of observing state $k$ at time $t$ given $x$. Given a length $T$ state target sequence $z$ the network is trained to minimise the negative log-probability of the target sequence given the input sequence:

$$\log \Pr(z|x) = \sum_{t=1}^{T} \log y_{z,t}(13)$$

Which leads to the following error derivatives at the output layer:

$$\frac{\partial \log \Pr(z|x)}{\partial \hat{y}_{k,t}} = y_{k,t}$$

where $\hat{y}_t$ is the vector of output activations before they have been normalised with the softmax function. These derivatives are then fed back through the network using backpropagation through time to determine the weight gradient.

When training deep networks in hybrid systems with stochastic gradient descent it has been found advantageous to select minibatches of frames randomly from the whole training set, rather than using whole utterances as batches. This is impossible with RNN-HMM hybrids because the weight gradients are a function of the entire utterance.

Another difference is that hybrid deep networks are trained with an acoustic context window of frames to either side of the one being classified. This is not necessary for DBLSTM, since it is as able to store past and future context internally, and the data was therefore presented a single frame at a time.

For some of the experiments Gaussian noise was added to the network weights during training [15]. The noise was added once per training sequence, rather than at every timestep. Weight noise tends to 'simplify' neural networks, in the sense of reducing the amount of information required to transmit the parameters [16, 17], which improves generalisation.

4. TIMIT EXPERIMENTS

The first set of experiments were carried out on the TIMIT [18] speech corpus. Their purpose was to see how hybrid training for deep bidirectional LSTM compared with the end-to-end training methods described in [1]. To this end, we ensured that the data preparation, network architecture and training parameters were consistent with those in the previous work. To allow us to test for significance, we also carried out repeated runs of the previous experiments (which were only run once in the original paper). In addition, we ran hybrid experiments using a deep bidirectional RNN with $tanh$ hidden units instead of LSTM.

The standard 462 speaker set with all SA records removed was used for training, and a separate development set of 50 speakers was used for early stopping. Results are reported for the 24-speaker core test set. The audio data was preprocessed using a Fourier-transform-based filterbank with 40 coefficients (plus energy) distributed on a mel-scale, together with their first and second temporal derivatives. Each input
CONVOLUTIONAL NEURAL NETS
Expressive Capabilities of ANNs

- **Boolean functions:**
  - Every Boolean function can be represented by a network with a single hidden layer.
  - But might require exponential (in number of inputs) hidden units.

- **Continuous functions:**
  - Every bounded continuous function can be approximated with arbitrary small error, by a network with one hidden layer [Cybenko 1989; Hornik et al 1989].
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].
Using ANN to hierarchical representation

Good Representations are hierarchical

- **In Language:** hierarchy in syntax and semantics
  - Words -> Parts of Speech -> Sentences -> Text
  - Objects, Actions, Attributes... -> Phrases -> Statements -> Stories
- **In Vision:** part-whole hierarchy
  - Pixels -> Edges -> Textons -> Parts -> Objects -> Scenes

© Eric Xing @ CMU, 2006-2011
“Deep” learning: learning hierarchical representations

- Deep Learning: learning a hierarchy of internal representations
- From low-level features to mid-level invariant representations, to object identities
- Representations are increasingly invariant as we go up the layers
- using multiple stages gets around the specificity/invariance dilemma
Filtering + NonLinearity + Pooling = 1 stage of a Convolutional Net

- [Hubel & Wiesel 1962]:
  - **simple cells** detect local features
  - **complex cells** “pool” the outputs of simple cells within a retinotopic neighborhood.

```
Multiple convolutions
Retinotopic Feature Maps

pooling subsampling
```

“Simple cells”

“Complex cells”
Convolutional Network: Multi-Stage Trainable Architecture

Hierarchical Architecture
- Representations are more global, more invariant, and more abstract as we go up the layers

Alternated Layers of Filtering and Spatial Pooling
- Filtering detects conjunctions of features
- Pooling computes local disjunctions of features

Fully Trainable
- All the layers are trainable
Convolutional Net Architecture for Hand-writing recognition

Convolutional net for handwriting recognition (400,000 synapses)

- Convolutional layers (simple cells): all units in a feature plane share the same weights
- Pooling/subsampling layers (complex cells): for invariance to small distortions.
- Supervised gradient-descent learning using back-propagation
- The entire network is trained end-to-end. All the layers are trained simultaneously.
- [LeCun et al. Proc IEEE, 1998]
How to train?

To compute all the derivatives, we use a backward sweep called the **back-propagation algorithm** that uses the recurrence equation for $\frac{\partial E}{\partial X_i}$:

- $\frac{\partial E}{\partial X_n} = \frac{\partial C(X_n, Y)}{\partial X_n}$
- $\frac{\partial E}{\partial X_{n-1}} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial X_{n-1}}$
- $\frac{\partial E}{\partial W_n} = \frac{\partial E}{\partial X_n} \frac{\partial F_n(X_{n-1}, W_n)}{\partial W_n}$
- $\frac{\partial E}{\partial X_{n-2}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial X_{n-2}}$
- $\frac{\partial E}{\partial W_{n-1}} = \frac{\partial E}{\partial X_{n-1}} \frac{\partial F_{n-1}(X_{n-2}, W_{n-1})}{\partial W_{n-1}}$

...etc, until we reach the first module.

We now have all the $\frac{\partial E}{\partial W_i}$ for $i \in [1, n]$. 
Handwritten Digit Dataset MNIST: 60,000 training samples, 10,000 test samples

Application: MNIST Handwritten Digit Dataset
## Results on MNIST Handwritten Digits

<table>
<thead>
<tr>
<th>CLASSIFIER</th>
<th>DEFORMATION</th>
<th>PREPROCESSING</th>
<th>ERROR (%)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear classifier (1-layer NN)</td>
<td>none</td>
<td>12.00</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>linear classifier (1-layer NN)</td>
<td>deskewing</td>
<td>8.40</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>pairwise linear classifier</td>
<td>deskewing</td>
<td>7.60</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>K-nearest-neighbors, (L2)</td>
<td>none</td>
<td>3.09</td>
<td></td>
<td>Kenneth Wilder, U. Chicago</td>
</tr>
<tr>
<td>K-nearest-neighbors, (L2)</td>
<td>deskewing</td>
<td>2.40</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>K-nearest-neighbors, (L2)</td>
<td>deskew, clean, blur</td>
<td>1.80</td>
<td></td>
<td>Kenneth Wilder, U. Chicago</td>
</tr>
<tr>
<td>K-NN L3, 2 pixel jitter</td>
<td>deskew, clean, blur</td>
<td>1.22</td>
<td></td>
<td>Kenneth Wilder, U. Chicago</td>
</tr>
<tr>
<td><strong>K-NN, shape context matching</strong></td>
<td>shape context feature</td>
<td>0.63</td>
<td></td>
<td>Belongie et al. IEEE PAMI 2002</td>
</tr>
<tr>
<td>40 PCA + quadratic classifier</td>
<td>none</td>
<td>11.10</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>1000 RBF + linear classifier</td>
<td>none</td>
<td>3.30</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>K-NN, Tangent Distance</td>
<td>subsamp 16x16 pixels</td>
<td>1.10</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>SVM, Gaussian Kernel</td>
<td>none</td>
<td>1.40</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>SVM deg 4 polynomial</td>
<td>deskewing</td>
<td>1.10</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Reduced Set SVM deg 5 poly</td>
<td>deskewing</td>
<td>1.00</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Virtual SVM deg-9 poly</td>
<td>Affine</td>
<td>0.80</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>V-SVM, 2-pixel jittered</td>
<td>none</td>
<td>0.68</td>
<td></td>
<td>DeCoste and Scholkopf, MLJ 2002</td>
</tr>
<tr>
<td>V-SVM, 2-pixel jittered</td>
<td>deskewing</td>
<td>0.56</td>
<td></td>
<td>DeCoste and Scholkopf, MLJ 2002</td>
</tr>
<tr>
<td>2-layer NN, 300 HU, MSE</td>
<td>none</td>
<td>4.70</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>2-layer NN, 300 HU, MSE,</td>
<td>Affine</td>
<td>3.60</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>2-layer NN, 300 HU</td>
<td>deskewing</td>
<td>1.60</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>3-layer NN, 500+ 150 HU</td>
<td>none</td>
<td>2.95</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>3-layer NN, 500+ 150 HU</td>
<td>Affine</td>
<td>2.45</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>3-layer NN, 500+ 300 HU, CE, reg</td>
<td>none</td>
<td>1.53</td>
<td></td>
<td>Hinton, unpublished, 2005</td>
</tr>
<tr>
<td>2-layer NN, 800 HU, CE</td>
<td>none</td>
<td>1.60</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
<tr>
<td>2-layer NN, 800 HU, CE</td>
<td>Affine</td>
<td>1.10</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
<tr>
<td>2-layer NN, 800 HU, MSE</td>
<td>Elastic</td>
<td>0.90</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
<tr>
<td>2-layer NN, 800 HU, CE</td>
<td>Elastic</td>
<td>0.70</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
<tr>
<td>Convolutional net LeNet-1</td>
<td>subsamp 16x16 pixels</td>
<td>1.70</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Convolutional net LeNet-4</td>
<td>none</td>
<td>1.10</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Convolutional net LeNet-5,</td>
<td>none</td>
<td>0.95</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td><strong>Conv. net LeNet-5,</strong></td>
<td>Affine</td>
<td>0.80</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Boosted LeNet-4</td>
<td>Affine</td>
<td>0.70</td>
<td></td>
<td>LeCun et al. 1998</td>
</tr>
<tr>
<td>Conv. net, CE</td>
<td>Affine</td>
<td>0.60</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
<tr>
<td>Conv. net, CE</td>
<td>Elastic</td>
<td>0.40</td>
<td></td>
<td>Simard et al., ICDAR 2003</td>
</tr>
</tbody>
</table>
Application: ANN for Face Reco.

- The model

- The learned hidden unit weights

Typical input images

http://www.cs.cmu.edu/~tom/faces.html
Face Detection with a Convolutional Net
Computer vision features

SIFT

Spin image

Drawbacks of feature engineering
1. Needs expert knowledge
2. Time consuming hand-tuning
Sparse coding on images

Natural Images

Learned bases: “Edges”

New example

\[ x = 0.8 * b_{36} + 0.3 * b_{42} + 0.5 * b_{65} \]

[0, 0, ..., 0.8, ..., 0.3, ..., 0.5, ...] = coefficients (feature representation)

Courtesy: Lee and Ng
Basis (or features) can be learned by Optimization

Given input data \( \{x^{(1)}, \ldots, x^{(m)}\} \), we want to find good bases \( \{b_1, \ldots, b_n\} \):

\[
\min_{b,a} \sum_i \|x^{(i)} - \sum_j a_j^{(i)} b_j\|_2^2 + \beta \sum_i \|a^{(i)}\|_1
\]

- **Reconstruction error**
- **Sparsity penalty**
- **Normalization constraint**

\( \forall j : \|b_j\| \leq 1 \)

Solve by alternating minimization:

-- Keep \( b \) fixed, find optimal \( a \).
-- Keep \( a \) fixed, find optimal \( b \).
Learning Feature Hierarchy

Higher layer
(Combinations of edges)

“Sparse coding”
(edges)

Input image (pixels)

DBN (Hinton et al., 2006) with additional sparseness constraint.

[Related work: Hinton, Bengio, LeCun, and others.]
Convolutional architectures

- Weight sharing by convolution (e.g., [Lecun et al., 1989])
- “Max-pooling”
  - Invariance
  - Computational efficiency
  - Deterministic and feed-forward
- One can develop convolutional Restricted Boltzmann machine (CRBM).
- One can define *probabilistic max-pooling* that combine bottom-up and top-down information.

Courtesy: Lee and Ng
Convolutional Deep Belief Networks

- Bottom-up (greedy), layer-wise training
  - Train one layer (convolutional RBM) at a time.

- Inference (approximate)
  - Undirected connections for all layers (Markov net)
    [Related work: Salakhutdinov and Hinton, 2009]
  - Block Gibbs sampling or mean-field
  - Hierarchical probabilistic inference

© Eric Xing @ CMU, 2006-2011
Unsupervised learning of object-parts

Faces

Cars

Elephants

Chairs

Courtesy: Lee and Ng
Weaknesses & Criticisms

- Learning everything. Better to encode prior knowledge about structure of images.
  
  A: Compare with machine learning vs. linguists debate in NLP.

- Results not yet competitive with best engineered systems.
  
  A: Agreed. True for some domains.
Tutorials

• LSTMs
  – Christopher Olah’s blog
  – http://colah.github.io/posts/2015-08-Understanding-LSTMs/

• Convolutional Neural Networks
  – Andrej Karpathy, CS231n Notes