Learning Feature Hierarchies for Vision

Yann LeCun
Courant Institute of Mathematical Sciences and Center for Neural Science, New York University

Collaborators:
Rob Fergus,
Karol Gregor, Arthur Szlam, Graham Taylor
Y-Lan Boureau, Benoit Corda, Clément Farabet
Kevin Jarrett, Koray Kavukcuoglu, Pierre Sermanet
The Next Challenge for AI, Robotics, and Neuroscience

- How do we learn perception (e.g. vision)?
- How do we learn representations of the perceptual world?
- How do we learn visual categories from just a few examples?
The raw input is pre-processed through a hand-crafted feature extractor.

The features are not learned.

The trainable classifier is often generic (task independent), and “simple” (linear classifier, kernel machine, nearest neighbor,.....)

The most common Machine Learning architecture: the Kernel Machine.
...But the Mammalian Visual Cortex is Hierarchical. Why?

The ventral (recognition) pathway in the visual cortex has multiple stages:

Retina - LGN - V1 - V2 - V4 - PIT - AIT ....
Good Internal Representations are Hierarchical

- Low-level features - mid-level features - high-level features - categories
- Representations are increasingly abstract, global, and invariant.

In Vision: part-whole hierarchy
- Pixels->Edges->Textons->Parts->Objects->Scenes

In Language: hierarchy in syntax and semantics
- Words->Parts of Speech->Sentences->Text
- Objects, Actions, Attributes...-> Phrases -> Statements -> Stories
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 non-linear stages gets around the specificity/invariance dilemma [Mallat 2010]
Feature Transform = Filter Bank + Non-Linearity + Pooling

Biologically-inspired models of low-level feature extraction
- Inspired by [Hubel and Wiesel 1962]
- Many feature extraction methods are based on this
- SIFT, GIST, HoG, Convolutional networks.....
[Hubel & Wiesel 1962]:

- Simple cells detect local features
- Complex cells "pool" the outputs of simple cells within a retinotopic neighborhood.

"Simple cells"

"Complex cells"

Multiple convolutions

Pooling subsampling

Retinotopic Feature Maps
Vision: Multiple Stage of Feature Transform + Classifier

- Stacking multiple stages of [Filter Bank + Non-Linearity + Pooling].
- Learning the filter banks at every layers
- Creating a hierarchy of features
- Basic elements are inspired by models of the visual cortex
  - Simple Cell + Complex Cell model of [Hubel and Wiesel 1962]
  - Many “traditional” feature extraction methods are based on this
  - SIFT, GIST, HoG, Convolutional networks.....
Non-Linearity: tanh, absolute value, shrinkage function, local whitening, ...

Pooling: average, max, Lp norm, .....
Example of Architecture: Convolutional Network (ConvNet)

Input:
83x83 (raw: 91x91)

Stage 1:
- Filter Bank + Tanh + Gain
- 64 features 75x75
- 64 filters
- 9x9 kernels
- Abs + Contrast Norm + Pooling + Downsampling
- 64 features 14x14
- 5x5 subsampling
- 10x10 pooling

Stage 2:
- Filter Bank + Tanh + Gain
- 256 features 6x6
- 4096 filters
- 9x9 kernels
- Abs + Contrast Norm + Pooling + Downsampling
- 256 features 1x1
- 4x4 subsampling
- 6x6 pooling

Parzen Windows Classifier

Yann LeCun
Training all the filters in a multi-stage ConvNet architecture

- End-to-End Supervised Learning by Stochastic Gradient Descent (backprop)
- Layerwise Unsupervised Training with Sparse Coding (“deep learning”)
- Supervised Refinement after Unsupervised pre-Training.

Lots of people work on these architectures: Rob Fergus (NYU), Geoff Hinton (Toronto), Andrew Ng (Stanford), David Lowe (UBC), Tommy Poggio (MIT), Larry Carin (Duke), Thomas Serre (Brown), Stéphane Mallat (Polytechnique), Sebastian Seung (MIT),

Industry: Kai Yu, Ronan Collobert (NEC), T. Dean, J. Weston (Google), C. Garcia (France Telecom), P. Simard (Microsoft) + a number of startups...
Supervised Learning of Convolutional Nets
Supervised Learning of ConvNets

- Stochastic Gradient Descent
- Gradients computed using back-propagation (chain rule)
- Filters are initialized randomly
### Face Detection: Results

<table>
<thead>
<tr>
<th>Data Set-&gt;</th>
<th>TILTED</th>
<th>PROFILE</th>
<th>MIT+CMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>False positives per image-&gt;</td>
<td>4.42</td>
<td>0.47</td>
<td>0.5</td>
</tr>
<tr>
<td>Our Detector</td>
<td>90%</td>
<td>67%</td>
<td>83%</td>
</tr>
<tr>
<td>Jones &amp; Viola (tilted)</td>
<td>90%</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Jones &amp; Viola (profile)</td>
<td>x</td>
<td>70%</td>
<td>x</td>
</tr>
</tbody>
</table>

- MIT+CMU: Data Set
  - PROFILE
  - TILTED

- Jones & Viola (profile)
  - MIT+CMU: x

- Jones & Viola (tilted)
  - MIT+CMU: x

- Our Detector
  - MIT+CMU: 83%

- Jones & Viola (profile)
  - MIT+CMU: 88%
Face Detection and Pose Estimation: Results
Face Detection with a ConvNet

Demo produced with EBLearn open source package

http://eblearn.sf.net
Generic Object Detection and Recognition with Invariance to Pose and Illumination

- 50 toys belonging to 5 categories: animal, human figure, airplane, truck, car
- 10 instance per category: 5 instances used for training, 5 instances for testing
- Raw dataset: 972 stereo pair of each object instance. 48,600 image pairs total.

- For each instance:
  - 18 azimuths
    - 0 to 350 degrees every 20 degrees
  - 9 elevations
    - 30 to 70 degrees from horizontal every 5 degrees
  - 6 illuminations
    - on/off combinations of 4 lights
  - 2 cameras (stereo)
    - 7.5 cm apart
    - 40 cm from the object

Training instances        Test instances
Experiment 2: Jittered-Cluttered Dataset

- 291,600 training samples, 58,320 test samples
- SVM with Gaussian kernel: 43.3% error
- Convolutional Net with *binocular* input: 7.8% error
- Convolutional Net + SVM on top: 5.9% error
- Convolutional Net with *monocular* input: 20.8% error
- Smaller *mono* net (DEMO): 26.0% error
- Dataset available from [http://www.cs.nyu.edu/~yann](http://www.cs.nyu.edu/~yann)
Examples (Monocular Mode)
### GTSRB Road Sign Recognition Competition (phase 1)

- 32x32 images
- The 13 of the top 14 entries are ConvNets, 6 from NYU, 7 from IDSIA
- No 6 is humans!

<table>
<thead>
<tr>
<th>#</th>
<th>Team</th>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>197</td>
<td>IDSIA</td>
<td>cnn_hog3</td>
<td>98.98%</td>
</tr>
<tr>
<td>196</td>
<td>IDSIA</td>
<td>cnn_cnn_hog3</td>
<td>98.98%</td>
</tr>
<tr>
<td>195</td>
<td>IDSIA</td>
<td>cnn_cnn_hog3_haar</td>
<td>98.97%</td>
</tr>
<tr>
<td>187</td>
<td>sermanet</td>
<td>EBLearn 2LConvNet ms 108 feats + 100-feats CF classifier + No color</td>
<td>99.17%</td>
</tr>
<tr>
<td>178</td>
<td>sermanet</td>
<td>EBLearn 2LConvNet ms 108 feats</td>
<td>98.97%</td>
</tr>
<tr>
<td>199</td>
<td>INI-RTCV</td>
<td>Human performance</td>
<td>98.81%</td>
</tr>
<tr>
<td>170</td>
<td>IDSIA</td>
<td>CNN(IMG)_MLP(HOG3)</td>
<td>98.79%</td>
</tr>
</tbody>
</table>
Brain tissue reconstruction from slice images [Jain, Denk, Seung 2007]

- Sebastian Seung's lab at MIT.
- 3D convolutional net for **image segmentation**
- ConvNets Outperform MRF, Conditional Random Fields, Mean Shift, Diffusion, ...[ICCV'07]
Visual Navigation for a Mobile Robot

[LeCun et al. NIPS 2005]

- Mobile robot with two cameras
- The convolutional net is trained to emulate a human driver from recorded sequences of video + human-provided steering angles.
- The network maps stereo images to steering angles for obstacle avoidance
Industrial Applications of ConvNets

- **AT&T/Lucent/NCR**
  - Check reading, OCR, handwriting recognition (deployed 1996)

- **Vidient Inc**
  - Vidient Inc's "SmartCatch" system deployed in several airports and facilities around the US for detecting intrusions, tailgating, and abandoned objects (Vidient is a spin-off of NEC)

- **NEC Labs**
  - Cancer cell detection, automotive applications, kiosks

- **Google**
  - Face and license plate removal from StreetView

- **Microsoft**
  - OCR, handwriting recognition, speech detection

- **France Telecom**
  - Face detection, HCI, cell phone-based applications

- **Other projects: HRL (3D vision)....**
Embedded Hardware for Fast ConvNet
NeuFlow: a Dataflow Computer for Embedded Vision

**High Peak Performance**
- Current proven implementation: 92 GOP/sec on a Xilinx Virtex 6
- Beta version: 200 GOP/sec on a Xilinx Virtex 6
- Simulated version: 700 GOP/sec on an IBM 45nm process

**High Actual Performance with a Custom Dataflow Compiler**
- Takes a high-level description of a processing flow as an input (any typical image transform, as found in EbLearn/GbLearn/Torch)
- Performs a multi-step analysis and generates optimized bytecode for NeuFlow, by minimizing memory bandwidth usage
- A typical ConvNet is computed at an average of 80 to 90% (end-to-end) of the peak perf (GPU code rarely goes beyond 20/30%)
FPGA Custom Board: NYU ConvNet Processor

- Xilinx Virtex 4 FPGA, 8x5 cm board
  - Dual camera port, Fast dual QDR RAM,

- New version developed in collaboration with Eugenio Culurciello (Yale)
  - Version for Virtex 6 FPGA development board (operational!)

[Farabet et al. ISCAS 2009]
NeuFlow: Dataflow architecture for ConvNet/Vision

Reconfigurable Dataflow Architecture: grid of processor tiles

[Farabet et al. ISCAS 2010]
xFlow & LuaFlow: Language/Compiler for NeuFlow

- Algorithm described as a graph of computing nodes (dataflow graph)
- Compiler generates instructions for NeuFlow processor (or CUDA)

flow-graph model

\[ \text{graph parsing} = \{ \text{node reordering, node merging} \} \]

memory map & sequence of transforms

\[ \text{device mapping} = \{ \text{static scheduling, reconfiguration sequencing} \} \]

sequence of reconfigurations & memory transfers

\[ \text{compilation} = \{ \text{machine code generation} \} \]

binary code
LuaFlow program example

- 16 convolutions, 9x9 kernels - > tanh - > fully-connected layer

- initializing neuFlow:
  ```lua
nueFlow = NeuFlow{mode='runtime'}
```

- describing a neural net:
  ```lua
  input_host = torch.Tensor(100,100)
  net = nn.Sequential()
  net:add(nn.SpatialConvolution(1,16,9,9))
  net:add(nn.Tanh())
  net:add(nn.SpatialLinear(16,4))
  ```

- elaborating the code for neuFlow:
  ```lua
  neuFlow:beginLoop('main')
      input_nf = neuFlow:copyFromHost(input_host)
      output_nf = neuFlow:compile(net, input_nf)
      output_host = neuFlow:copyToHost(output_nf)
  neuFlow:endLoop('main')
  ```
LuaFlow program example

16 convolutions, 9x9 kernels - > tanh - > fully-connected layer

- loading the bytecode on neuFlow:
  ```lua
  neuFlow:loadBytecode()
  -- at this point, neuFlow executes its new code
  ```

- now simply describe the host code:
  ```lua
  while true do
    input_host = camera:getFrame()
    neuFlow:copyToDev(input_host)
    neuFlow:copyFromDev(output_host)
    result = soft_classifier:forward(output_host)
  end
  ```

- at this point the code is running in a loop, neuFlow is computing the neural net, while the host computes a simple linear classifier on the results
FPGA Performance

Seconds per frame for a robot vision task (log scale)  [Farabet et al. 2010]

- X86 Core2 Duo: 3s
- Nvidia 9400M GPU: 25ms
- Virtex 4 custom board: 6ms
- Nvidia Tesla C1060: 6ms
- Virtex 6 dev board: 6ms
NeuFlow: Performance with the LAGR ConvNet

Example: a typical ConvNet trained for obstacle detection (LAGR)

Software on Intel x86: 1 frame per second.

NeuFlow Virtex 6: 30 frames per second.
**NeuFlow ASIC**

- Collaboration with e-Lab (Yale)
- Design in progress
- 45 nm technology
- 700 Gop/s
- < 3 Watts.
- 5x3 mm
## NeuFlow: Performance Comparison

### Example: a typical ConvNet

<table>
<thead>
<tr>
<th></th>
<th>CPU Intel 2 cores</th>
<th>GPU NVidia GT335m</th>
<th>GPU NVidia GS1070</th>
<th>FPGA NeuFlow Virtex-4</th>
<th>FPGA NeuFlow Virtex-6</th>
<th>ASIC NeuFlow IBM 45nm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak Gop/sec</strong></td>
<td>10</td>
<td>182</td>
<td>1000</td>
<td>40</td>
<td>160</td>
<td>700</td>
</tr>
<tr>
<td><strong>Actual Gop/sec</strong></td>
<td>1.1</td>
<td>54</td>
<td>290</td>
<td>37</td>
<td>147</td>
<td>99</td>
</tr>
<tr>
<td><strong>FPS</strong></td>
<td>1.4</td>
<td>67</td>
<td>360</td>
<td>46</td>
<td>182</td>
<td>700</td>
</tr>
<tr>
<td><strong>Power (W)</strong></td>
<td>30</td>
<td>30</td>
<td>220</td>
<td>10</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td><strong>Gop/s/W</strong></td>
<td>0.037</td>
<td>1.8</td>
<td>1.32</td>
<td>3.7</td>
<td>9.8</td>
<td>200</td>
</tr>
</tbody>
</table>
Computer vision

Eye robot

Poor eyesight remains one of the main obstacles to letting robots loose among humans. But it is improving, in part by aping natural vision

Oct 21st 2010

A ConvNet begins by swiping a number of software filters, each several pixels across, over the image, pixel by pixel. Like the brain’s primary visual cortex, these filters look for simple features such as edges. The upshot is a set of feature maps, one for each filter, showing which patches of the original image contain the sought-after element. A series of transformations is then performed on each map in order to enhance it and improve the contrast. Next, the maps are swiped again, but this time rather than stopping at each pixel, the filter takes a snapshot every few pixels. That produces a new set of maps of lower resolution. These highlight the salient features while reining in computing power. The whole process is then repeated, with several hundred filters probing for more elaborate shapes rather than just a few scouring for simple ones. The resulting array of feature maps is run through one final set of filters. These classify objects into general categories, such as pedestrians or cars.

Many state-of-the-art computer-vision systems work along similar lines. The uniqueness of ConvNets lies in where they get their filters. Traditionally, these were simply plugged in one by one, in a laborious manual process that required an expert human eye to tell the machine what features to look for, in future, at each level. That made systems which relied on them good at spotting narrow classes of objects but inept at discerning anything else.

Dr LeCun’s artificial visual cortex, by contrast, lights on the appropriate filters automatically as it is taught to distinguish the different types of object. When an
Object Recognition with ConvNets

Filter Bank  Non-Linearity  feature Pooling  Filter Bank  Non-Linearity  feature Pooling  Classifier

Local Divisive Normalization

Convolutions w/ filter bank: 20x7x7 kernels

Pooling: 20x4x4 kernels

Convs: 100x7x7 kernels

Pooling: 20x4x4 kernels

Convs: 800x7x7 kernels

Linear Classifier

Object Categories / Positions

Input Image: 1x500x500

Normalized Image: 1x500x500

C1: 20x494x494

S2: 20x123x123

C3: 20x117x117

C4: 20x29x29

C5: 200x23x23

{ } at (x,y)
Problem: supervised ConvNets don't work with few labeled samples

On recognition tasks with few labeled samples, deep supervised architectures don't do so well.

Example: Caltech-101 Object Recognition Dataset
- 101 categories of objects (gathered from the web)
- Only 30 training samples per category!

Recognition rates (OUCH!):
- Supervised ConvNet: 29.0%
- SIFT features + Pyramid Match Kernel SVM: 64.6%
  [Lazebnik et al. 2006]

When learning the features, there are simply too many parameters to learn in purely supervised mode (or so we thought).

Yann LeCun
New York University
“Conventional” vision systems are similar to ConvNets

Fixed low-level Features + unsupervised mid-level features + simple classifier

Example:
- SIFT + K-means + Pyramid pooling + SVM intersection kernel: \( >65\% \)  
  [Lazebnik et al. CVPR 2006]
- SIFT + Sparse coding + Pyramid pooling + SVM: \( >73\% \)  
  [Yang et al. CVPR 2009, Boureau et al. CVPR 2010]
Unsupervised Deep Learning: Leveraging Unlabeled Data

Unlabeled data is usually available in large quantity

A lot can be learned about the world by just looking at it

Unsupervised learning captures underlying regularities about the data

The best way to capture underlying regularities is to learn good representations of the data

The main idea of Unsupervised Deep Learning
  ▶ Learn each layer one at a time in unsupervised mode
  ▶ Stick a supervised classifier on top
  ▶ Optionally: refine the entire system in supervised mode

Unsupervised Learning view as Energy-Based Learning

[Hinton 05, Bengio 06, LeCun 06, Ng 07]
Find a dictionary of basis functions such that any input can be reconstructed of a sparse linear combination of them.

Energy: \[ E(Y^i, Z; W_d) = ||Y^i - W_d Z||^2 + \lambda \sum_j |z_j| \]

Optimal Code: \[ Z^i = \arg\min_z E(Y^i, z; W_d) \]

Free Energy: \[ F(Y^i; W_d) = F(Z^i) = \min_z E(Y^i, z; W_d) \]
Unsupervised Feature Learning with Sparse Coding

The learning algorithm minimizes the loss function:

\[ L(W_d) = \sum_i F(Y^i; W_d) = \sum_i \left( \min_Z E(Y^i, Z; W_d) \right) \]

The columns of \( W_d \) are normalized.

**Energy:**

\[ E(Y^i, Z; W_d) = \|Y^i - W_dZ\|^2 + \lambda \sum_j |z_j| \]

**Free Energy:**

\[ F(Y^i; W_d) = F(Z^i) = \min_z E(Y^i, z; W_d) \]
Problem with Sparse Coding: Inference is slow

Inference: find $Z$ that minimizes the energy for a given $Y$

$$
E(Y^i, Z^i; W_d) = \| Y^i - W_d Z^i \|^2 + \lambda \sum_j |z^i_j|
$$

$$
Z^i = \arg\min_z E(Y^i, z; W_d)
$$

- For each new $Y$, an optimization algorithm must be run to find the corresponding optimal $Z$
- This would be very slow for large scale vision tasks
- Also, the optimal $Z$ are very unstable:
  - A small change in $Y$ can cause a large change in the optimal $Z$
Solution: Predictive Sparse Decomposition (PSD)

Prediction the optimal code with a trained encoder

Energy = reconstruction_error + code_prediction_error + code_sparsity

\[ E(Y^i, Z) = \|Y^i - W_d Z\|^2 + \|Z - g_e(W_e, Y^i)\|^2 + \lambda \sum_j |z_j| \]

\[ g_e(W_e, Y^i) = D \tanh(W_e Y) \]
PSD: Inference

Inference by gradient descent starting from the encoder output

\[ E(Y^i, Z) = \|Y^i - W_d Z\|^2 + \|Z - g_e(W_e, Y^i)\|^2 + \lambda \sum_j |z_j| \]

\[ Z^i = \text{argmin}_z E(Y^i, z; W) \]
Learning by minimizing the average energy of the training data with respect to $W_d$ and $W_e$.

**Loss function:**

$$L(W_d, W_e) = \sum_i F(Y^i; W_d, W_e)$$

$$F(Y^i; W_d, W_e) = \min_z E(Y^i, z; W_d, W_e)$$
PSD: Learning Algorithm

1. Initialize $Z = \text{Encoder}(Y)$
2. Find $Z$ that minimizes the energy function
3. Update the Decoder basis functions to reduce reconstruction error
4. Update Encoder parameters to reduce prediction error

Repeat with next training sample
PSD: Encoder Architectures

Simple: $D \cdot \tanh(We.Y)$

Sophisticated/Iterative: $Z(t) = \text{shrink}( We.Y + S.Z(t-1) )$

For a discussion of best encoders, see [Gregor & LeCun, ICML 2010]
Encoder Architecture inspired by ISTA [Gregor, LeCun ICML 2010] (Iterative Shrinkage and Thresholding Algorithm)

ISTA/FISTA: iterative algorithm that converges to optimal sparse code

INPUT

\[ Z(t) = sh[We.Y + S.Z(t-1)] \]

Time-Unfolded version of ISTA/FISTA. Idea: learn the We and S matrices

Yann LeCun
Learning ISTA (LISTA) vs ISTA/FISTA

- FISTA (4x)
- FISTA (1x)
- LISTA (4x)
- LISTA (1x)

error

iter
LISTA with partial mutual inhibition matrix

![Graph with error vs. cf for different methods: dim reduction (4x), elements removal (4x), dim reduction (1x), elements removal (1x).]

- **dim reduction (4x)**
- **elements removal (4x)**
- **dim reduction (1x)**
- **elements removal (1x)**
Learning Coordinate Descent (LcoD): faster than LISTA

error

iter

- CoD (4x)
- CoD (1x)
- LCoD (4x)
- LCoD (1x)
PSD trained on handwritten digits: decoder filters are “parts” (strokes).

- Any digit can be reconstructed as a linear combination of a small number of these “parts”.

![MNIST handwritten digits](image-url)
Basis functions are like Gabor filters (like receptive fields in V1 neurons)

256 filters of size 12x12

Trained on natural image patches from the Berkeley dataset

Encoder is linear-tanh-diagonal
Learned Features on natural patches: V1-like receptive fields
Phase 1: train first layer using PSD

\[ \|Y^i - \hat{Y}\|^2 \]

\[ W_d Z \]

\[ \|Z - \hat{Z}\|^2 \]

\[ g_e(W_e, Y^i) \]
Using PSD to Train a Hierarchy of Features

- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
Using PSD to Train a Hierarchy of Features

- **Phase 1:** train first layer using PSD
- **Phase 2:** use encoder + absolute value as feature extractor
- **Phase 3:** train the second layer using PSD
Using PSD to Train a Hierarchy of Features

- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
- Phase 3: train the second layer using PSD
- Phase 4: use encoder + absolute value as 2\textsuperscript{nd} feature extractor

\[ g_e(W_e, Y^i) \rightarrow |z_j| \rightarrow g_e(W_e, Y^i) \rightarrow |z_j| \]
Using PSD to Train a Hierarchy of Features

- Phase 1: train first layer using PSD
- Phase 2: use encoder + absolute value as feature extractor
- Phase 3: train the second layer using PSD
- Phase 4: use encoder + absolute value as 2\textsuperscript{nd} feature extractor
- Phase 5: train a supervised classifier on top
- Phase 6 (optional): train the entire system with supervised back-propagation
A General View of Unsupervised Learning
The “deep learning” method was popularized by Hinton for training “deep belief networks”.

DBN use a special kind of encoder-decoder architecture called Restricted Boltzmann Machines (RBM)

1. Train each layer in an unsupervised fashion, layer by layer

2. Stick a supervised classifier on top, and refine the entire system with gradient descent (back-prop) on a supervised criterion.
Unsupervised Learning: Capturing Dependencies Between Variables

**Energy function: viewed as a negative log probability density**

**Probabilistic View:**
- Produce a probability density function that:
  - has high value in regions of high sample density
  - has low value everywhere else (integral = 1).

**Energy-Based View:**
- produce an energy function $E(Y,W)$ that:
  - has low value in regions of high sample density
  - has high(er) value everywhere else
Unsupervised Learning: Capturing Dependencies Between Variables

Energy function viewed as a negative log density

Example: $y = x^2$
\[
P(Y|W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}
\]

\[
E(Y, W) \propto - \log P(Y|W)
\]
Training an Energy-Based Model

- Make the energy around training samples low
- Make the energy everywhere else higher

\[
P(Y, W) = \frac{e^{-\beta E(Y, W)}}{\int_y e^{-\beta E(y, W)}}
\]
Training an Energy-Based Model to Approximate a Density

Maximizing $P(Y|W)$ on training samples

$$P(Y|W) = \frac{e^{-\beta E(Y,W)}}{\int_y e^{-\beta E(y,W)}}$$

Minimizing $-\log P(Y,W)$ on training samples

$$L(Y, W) = E(Y, W) + \frac{1}{\beta} \log \int_y e^{-\beta E(y,W)}$$
Training an Energy-Based Model with Gradient Descent

Gradient of the negative log-likelihood loss for one sample $Y$:

\[
\frac{\partial L(Y, W)}{\partial W} = \frac{\partial E(Y, W)}{\partial W} - \int_y P(y|W) \frac{\partial E(y, W)}{\partial W}
\]

Gradient descent:

\[
W \leftarrow W - \eta \frac{\partial L(Y, W)}{\partial W}
\]

- Pushes down on the energy of the samples
- Pulls up on the energy of low-energy $Y$'s

\[
W \leftarrow W - \eta \frac{\partial E(Y, W)}{\partial W} + \eta \int_y P(y|W) \frac{\partial E(y, W)}{\partial W}
\]
How do we push up on the energy of everything else?

**Solution 1: contrastive divergence [Hinton 2000]**
- Move away from a training sample a bit
- Push up on that

**Solution 2: score matching**
- On the training samples: minimize the gradient of the energy, and maximize the trace of its Hessian.

**Solution 3: denoising auto-encoder (not really energy-based)**
- Train the inference dynamics to map noisy samples to clean samples

**Solution 4: MAIN INSIGHT! [Ranzato, ..., LeCun AI-Stat 2007]**
- **Restrict the information content of the code (features) Z**
  - If the code Z can only take a few different configurations, only a correspondingly small number of Ys can be perfectly reconstructed
- Idea: impose a sparsity prior on Z
- This is reminiscent of sparse coding [Olshausen & Field 1997]
Restricted Boltzmann Machines

- Y and Z are binary
- Enc and Dec are linear
- Distance is negative dot product

\[ E(Y, Z) = \text{Dist}[Y, \text{Dec}(Z)] + \text{Dist}[Z, \text{Enc}(Y)] \]

\[ \text{Enc}(Y) = -W.Y \quad \text{Dist}(Z, W.Y) = -\frac{1}{2}Z^T W.Y \]

\[ \text{Dec}(Y) = -W^T.Z \quad \text{Dist}(Y, E^T.Z) = -\frac{1}{2}Y^T W^T.Z \]

\[ E(Y, Z) = -Z^T W.Y \quad F(Y) = -\log \sum_z e^{Z^T W.Y} \]

[Hinton & Salakhutdinov 2005]
If the information content of the feature vector is limited (e.g. by imposing sparsity constraints), the energy MUST be large in most of the space.

- pulling down on the energy of the training samples will necessarily make a groove

The volume of the space over which the energy is low is limited by the entropy of the feature vector

- Input vectors are reconstructed from feature vectors.
- If few feature configurations are possible, few input vectors can be reconstructed properly.
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

*Training based on minimizing the reconstruction error over the training set*
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

*BAD: machine does not learn structure from training data!!*

*It just copies the data.*
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

IDEA: reduce number of available codes.
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

**IDEA:** reduce number of available codes.
Why Limit the Information Content of the Code?

- Training sample
- Input vector which is NOT a training sample
- Feature vector

*IDEA: reduce number of available codes.*
We are going to impose a sparsity penalty on the code to restrict its information content.

We will allow the code to have higher dimension than the input.

Categories are more easily separable in high-dim sparse feature spaces.
- This is a trick that SVM use: they have one dimension per sample.

Sparse features are optimal when an active feature costs more than an inactive one (zero).
- e.g. neurons that spike consume more energy.
- The brain is about 2% active on average.
- **2 dimensional toy dataset**
- **Mixture of 3 Cauchy distrib.**
- **Visualizing energy surface**
  (black = low, white = high)

[Ranzato's PhD thesis 2009]

<table>
<thead>
<tr>
<th></th>
<th>PCA</th>
<th>autoencoder</th>
<th>sparse coding</th>
<th>K-Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoder</td>
<td>(W'Y)</td>
<td>(\sigma(W_eY))</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>decoder</td>
<td>(WZ)</td>
<td>(W_dZ)</td>
<td>(WZ)</td>
<td>(WZ)</td>
</tr>
<tr>
<td>energy</td>
<td>(|Y - WZ|^2)</td>
<td>(|Y - WZ|^2)</td>
<td>(|Y - WZ|^2 + \lambda</td>
<td>Z</td>
</tr>
<tr>
<td>loss</td>
<td>(F(Y))</td>
<td>(F(Y) + \log \Gamma)</td>
<td>(F(Y))</td>
<td>(F(Y))</td>
</tr>
<tr>
<td>pull-up</td>
<td>dimens.</td>
<td>part. func.</td>
<td>sparsity</td>
<td>1-of-N code</td>
</tr>
</tbody>
</table>
Energies Surfaces for Various Loss Functions

- PCA
- energy loss
- neg-log-likel.
- margin loss
- sparse cod.
- kmeans

CD1
Back To Object Recognition
1. Train stage-1 filters with PSD on patches from natural images
2. Compute stage-1 features on training set
3. Train state-2 filters with PSD on stage-1 feature patches
4. Compute stage-2 features on training set
5. Train linear classifier on stage-2 features
6. Refine entire network with supervised gradient descent

What are the effects of the non-linearities and unsupervised pretraining?
Using PSD Features for Object Recognition

- 64 filters on 9x9 patches trained with PSD
  - with Linear-Sigmoid-Diagonal Encoder

weights = -0.2828 - 0.3043
Adding Rectification and Normalization Modules

- **C** Convolutions (filter bank)
- **Tanh+Abs** tanh (sigmoid) + Absolute Value Rectification
- **N** Subtractive and Divisive Local Normalization
- **P** Pooling down-sampling layer: average or max?

---

**C**

Convolutions

**Tanh + Abs**

**N**

Subtractive and Divisive Normalization

**P**

Pooling, sub-sampling

---

THIS IS ONE STAGE OF THE CONVNET
Local Contrast Normalization

- Performed on the state of every layer, including the input

**Subtractive Local Contrast Normalization**
- Subtracts from every value in a feature a Gaussian-weighted average of its neighbors (high-pass filter)

**Divisive Local Contrast Normalization**
- Divides every value in a layer by the standard deviation of its neighbors over space and over all feature maps

**Subtractive + Divisive LCN performs a kind of approximate whitening.**
Training Protocol

Training

- Logistic Regression on Random Features: $R$
- Logistic Regression on PSD features: $U$
- Refinement of whole net from random with backprop: $R^+$
- Refinement of whole net starting from PSD filters: $U^+$

Classifier

- Multinomial Logistic Regression or Pyramid Match Kernel SVM
Using PSD Features for Recognition

\[
[64.F_{CSG}^{9 \times 9} - R/N/P^{5 \times 5}] - \text{log\_reg}
\]

<table>
<thead>
<tr>
<th>R/N/P</th>
<th>(R_{abs} - N - P_A)</th>
<th>(R_{abs} - P_A)</th>
<th>(N - P_M)</th>
<th>(N - P_A)</th>
<th>(P_A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(U^+)</td>
<td>54.2%</td>
<td>50.0%</td>
<td>44.3%</td>
<td>18.5%</td>
<td>14.5%</td>
</tr>
<tr>
<td>(R^+)</td>
<td>54.8%</td>
<td>47.0%</td>
<td>38.0%</td>
<td>16.3%</td>
<td>14.3%</td>
</tr>
<tr>
<td>(U)</td>
<td>52.2%</td>
<td>43.3(±1.6)%</td>
<td>44.0%</td>
<td>17.2%</td>
<td>13.4%</td>
</tr>
<tr>
<td>(R)</td>
<td>53.3%</td>
<td>31.7%</td>
<td>32.1%</td>
<td>15.3%</td>
<td>12.1(±2.2)%</td>
</tr>
</tbody>
</table>

\[
[64.F_{CSG}^{9 \times 9} - R/N/P^{5 \times 5}] - \text{PMK}
\]

| U | 65.0% |

\[
[96.F_{CSG}^{9 \times 9} - R/N/P^{5 \times 5}] - \text{PCA - lin\_svm}
\]

| U | 58.0% |

96.Gabor - PCA - lin\_svm (Pinto and DiCarlo 2006)

| Gabors | 59.0% |

SIFT - PMK (Lazebnik et al. CVPR 2006)

| Gabors | 64.6% |
Rectification makes a huge difference:
- 14.5% -> 50.0%, without normalization
- 44.3% -> 54.2% with normalization

Normalization makes a difference:
- 50.0 → 54.2

Unsupervised pretraining makes small difference

PSD works just as well as SIFT

Random filters work as well as anything!
- If rectification/normalization is present

PMK_SVM classifier works a lot better than multinomial log_reg on low-level features
- 52.2% → 65.0%
Multistage Hubel-Wiesel Architecture

**Image Preprocessing:**
- High-pass filter, local contrast normalization (divisive)

**First Stage:**
- Filters: 64 9x9 kernels producing 64 feature maps
- Pooling: 10x10 averaging with 5x5 subsampling

**Second Stage:**
- Filters: 4096 9x9 kernels producing 256 feature maps
- Pooling: 6x6 averaging with 3x3 subsampling
- Features: 256 feature maps of size 4x4 (4096 features)

**Classifier Stage:**
- Multinomial logistic regression

**Number of parameters:**
- Roughly 750,000
### Multistage Hubel-Wiesel Architecture on Caltech-101

#### Single Stage System: \([64.F_{\text{CSG}}^{9 \times 9} - R/N/P_{5 \times 5}] - \log \_\text{reg}\)

<table>
<thead>
<tr>
<th>R/N/P</th>
<th>(R_{\text{abs}} - N - P_A)</th>
<th>(R_{\text{abs}} - P_A)</th>
<th>(N - P_M)</th>
<th>(N - P_A)</th>
<th>(P_A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(U^+)</td>
<td>54.2%</td>
<td>50.0%</td>
<td>44.3%</td>
<td>18.5%</td>
<td>14.5%</td>
</tr>
<tr>
<td>(R^+)</td>
<td>54.8%</td>
<td>47.0%</td>
<td>38.0%</td>
<td>16.3%</td>
<td>14.3%</td>
</tr>
<tr>
<td>(U)</td>
<td>52.2%</td>
<td>43.3%(±1.6)</td>
<td>44.0%</td>
<td>17.2%</td>
<td>13.4%</td>
</tr>
<tr>
<td>(R)</td>
<td>53.3%</td>
<td>31.7%</td>
<td>32.1%</td>
<td>15.3%</td>
<td>12.1%(±2.2)</td>
</tr>
<tr>
<td>(G)</td>
<td>52.3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Two Stage System: \([64.F_{\text{CSG}}^{9 \times 9} - R/N/P_{5 \times 5}] - \log \_\text{reg}\) – \([256.F_{\text{CSG}}^{9 \times 9} - R/N/P_{4 \times 4}] - \log \_\text{reg}\)

<table>
<thead>
<tr>
<th>R/N/P</th>
<th>(R_{\text{abs}} - N - P_A)</th>
<th>(R_{\text{abs}} - P_A)</th>
<th>(N - P_M)</th>
<th>(N - P_A)</th>
<th>(P_A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(U^+U^+)</td>
<td>65.5%</td>
<td>60.5%</td>
<td>61.0%</td>
<td>34.0%</td>
<td>32.0%</td>
</tr>
<tr>
<td>(R^+R^+)</td>
<td>64.7%</td>
<td>59.5%</td>
<td>60.0%</td>
<td>31.0%</td>
<td>29.7%</td>
</tr>
<tr>
<td>(UU)</td>
<td>63.7%</td>
<td>46.7%</td>
<td>56.0%</td>
<td>23.1%</td>
<td>9.1%</td>
</tr>
<tr>
<td>(RR)</td>
<td>62.9%</td>
<td>33.7%(±1.5)</td>
<td>37.6%(±1.9)</td>
<td>19.6%</td>
<td>8.8%</td>
</tr>
<tr>
<td>(GT)</td>
<td>55.8%</td>
<td></td>
<td></td>
<td></td>
<td>like HMAX model</td>
</tr>
</tbody>
</table>

#### Single Stage: \([64.F_{\text{CSG}}^{9 \times 9} - R/N/P_{5 \times 5}] - \text{PMK-SVM}\)

| \(U\) | 64.0\% | |

#### Two Stages: \([64.F_{\text{CSG}}^{9 \times 9} - R/N/P_{5 \times 5}] - \log \_\text{reg}\) – \([256.F_{\text{CSG}}^{9 \times 9} - R/N]\) - \text{PMK-SVM}

| \(UU\) | 52.8\% |
Using more ideas from biology

- **Pyramid Pooling**
  - Multi-scale pooling at the last stage

- **Threshold/Shrinkage Response Function + Lateral Inhibition Matrix**
  - Filter Bank - Shrinkage - Inhibition - Shrinkage

- **Discriminative term during pre-training (using label information)**
  - $E(x, y, z, D, \theta) = C(y, l(z, \theta)) + \|x - Dz\|^2_2 + \lambda_1 \|z\|_1$
  - [Mairal NIPS 09], [Boureau CVPR 10]
Using a few more tricks...

- Pyramid pooling on last layer: 1% improvement over regular pooling
- Shrinkage non-linearity + lateral inhibition: 1.6% improvement over tanh
- Discriminative term in sparse coding: 2.8% improvement

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Protocol</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $F_{tanh} - R_{abs} - N - P^\text{pyr}_A$</td>
<td>$R^+R^+$</td>
<td>65.4 ± 1.0</td>
</tr>
<tr>
<td>(2) $F_{tanh} - R_{abs} - N - P^\text{pyr}_A$</td>
<td>$U^+U^+$</td>
<td>66.2 ± 1.0</td>
</tr>
<tr>
<td>(3) $F_{si} - R_{abs} - N - P_A$</td>
<td>$R^+R^+$</td>
<td>63.3 ± 1.0</td>
</tr>
<tr>
<td>(4) $F_{si} - R_{abs} - N - P_A$</td>
<td>$UU$</td>
<td>60.4 ± 0.6</td>
</tr>
<tr>
<td>(5) $F_{si} - R_{abs} - N - P_A$</td>
<td>$U^+U^+$</td>
<td>66.4 ± 0.5</td>
</tr>
<tr>
<td>(6) $F_{si} - R_{abs} - N - P^\text{pyr}_A$</td>
<td>$U^+U^+$</td>
<td>67.8 ± 0.4</td>
</tr>
<tr>
<td>(7) $F_{si} - R_{abs} - N - P_A$</td>
<td>$DD$</td>
<td>66.0 ± 0.3</td>
</tr>
<tr>
<td>(8) $F_{si} - R_{abs} - N - P_A$</td>
<td>$D^+D^+$</td>
<td>68.7 ± 0.2</td>
</tr>
<tr>
<td>(9) $F_{si} - R_{abs} - N - P^\text{pyr}_A$</td>
<td>$D^+D^+$</td>
<td>70.6 ± 0.3</td>
</tr>
</tbody>
</table>
Latest Results and Analysis

Latest result on C-101: 70.8% correct

- Multi-scale pooling at the last layer (pyramid pooling)
- Discriminative term in the sparse coding unsupervised learning
- Different encoder architecture, with shrinkage function. And different sparse coding inference method (ISTA) [Gregor ICML 2010]

Second Stage + logistic regression = PMK_SVM

Unsupervised pre-training doesn't help much :-(

Random filters work amazingly well with normalization

Supervised global refinement helps a bit

The best system is really cheap

Either use rectification and average pooling or no rectification and max pooling.
Multistage Hubel-Wiesel Architecture: Filters

Stage 1
- After PSD
  - Weights: [-0.232, 0.2075]

Stage 2
- After supervised refinement
  - Weights: [-0.2828, 0.3043]
  - Weights: [-0.0778, 0.064]
  - Weights: [-0.0929, 0.0784]
Demo: real-time learning of visual categories

Input high-pass filtered contrast-normalized 83x83 (raw: 91x91)

STAGE 1
Filter Bank + Tanh + Gain
64 features 75x75
64 filters 9x9 kernels
Abs + Contrast Norm + Pooling + Downsampling
64 features 14x14
5x5 subsampling 10x10 pooling

STAGE 2
Filter Bank + Tanh + Gain
256 features 6x6
4096 filters 9x9 kernels
Abs + Contrast Norm + Pooling + Downsampling
256 features 1x1
4x4 subsampling 6x6 pooling

Parzen Windows Classifier

Yann LeCun
Why Random Filters Work?
Small NORB dataset

- 5 classes and up to 24,300 training samples per class
Small NORB dataset

Two-stage system: error rate versus number of labeled training samples

- No normalization
- Random filters
- Unsup filters
- Sup filters
- Unsup+Sup filters

Graph showing error rate versus number of labeled training samples per class.
ConvNets and “Conventional” Vision Architectures are Similar

Can't we use the same tricks as ConvNets to train the second stage of a “conventional vision architecture?

Stage 1: SIFT

Stage 2: discriminative sparse coding over neighborhoods + normalization + pooling
Using DL/ConvNet ideas in “conventional” recognition systems

Adapting insights from ConvNets:
- Jointly encoding spatial neighborhoods instead of single points: increase spatial receptive fields for higher-level features
- Use max pooling instead of average pooling
- Train supervised dictionary for sparse coding

This yields state-of-the-art results:
- 75.7% on Caltech-101 (+/-1.1%): record for single system
- 85.6% on 15-Scenes (+/- 0.2): record!

[Boureau et al. CVPR 2010]
## The Competition: SIFT + Sparse-Coding + PMK-SVM

### Replacing K-means with Sparse Coding

- [Yang 2008] [Boureau, Bach, Ponce, LeCun 2010]

<table>
<thead>
<tr>
<th>Method</th>
<th>Caltech 15</th>
<th>Caltech 30</th>
<th>Scenes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boiman et al. [1]</td>
<td>Nearest neighbor + spatial correspondence</td>
<td>65.00 ± 1.14</td>
<td>70.40</td>
</tr>
<tr>
<td>Jain et al. [8]</td>
<td>Fast image search for learned metrics</td>
<td>61.00</td>
<td>69.60</td>
</tr>
<tr>
<td>Lazebnik et al. [12]</td>
<td>Spatial Pyramid + hard quantization + kernel SVM</td>
<td>56.40</td>
<td>64.40 ± 0.80</td>
</tr>
<tr>
<td>van Gemert et al. [24]</td>
<td>Spatial Pyramid + soft quantization + kernel SVM</td>
<td>-</td>
<td>64.14 ± 1.18</td>
</tr>
<tr>
<td>Yang et al. [26]</td>
<td>SP + sparse codes + max pooling + linear</td>
<td><strong>67.00±0.45</strong></td>
<td><strong>73.2±0.54</strong></td>
</tr>
<tr>
<td>Zhang et al. [27]</td>
<td>kNN-SVM</td>
<td>59.10 ± 0.60</td>
<td>66.20 ± 0.50</td>
</tr>
<tr>
<td>Zhou et al. [29]</td>
<td>SP + Gaussian mixture</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Baseline:</strong></td>
<td><strong>SP + hard quantization + avg pool + kernel SVM</strong></td>
<td>56.74 ± 1.31</td>
<td>64.19 ± 0.94</td>
</tr>
<tr>
<td><strong>Unsupervised coding</strong></td>
<td><strong>SP + soft quantization + avg pool + kernel SVM</strong></td>
<td>59.12 ± 1.51</td>
<td>66.42 ± 1.26</td>
</tr>
<tr>
<td><strong>1×1 features</strong></td>
<td><strong>SP + soft quantization + max pool + kernel SVM</strong></td>
<td>63.61 ± 0.88</td>
<td>-</td>
</tr>
<tr>
<td><strong>8 pixel grid resolution</strong></td>
<td><strong>SP + sparse codes + avg pool + kernel SVM</strong></td>
<td>62.85 ± 1.22</td>
<td>70.27 ± 1.29</td>
</tr>
<tr>
<td></td>
<td><strong>SP + sparse codes + max pool + kernel SVM</strong></td>
<td>64.62 ± 0.94</td>
<td><strong>71.81±0.96</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SP + sparse codes + max pool + linear</strong></td>
<td><strong>64.71 ± 1.05</strong></td>
<td><strong>71.52 ± 1.13</strong></td>
</tr>
</tbody>
</table>

**Macrofeatures +**

<table>
<thead>
<tr>
<th>Method</th>
<th>Caltech 15</th>
<th>Caltech 30</th>
<th>Scenes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Finer grid resolution</strong></td>
<td><strong>SP + sparse codes + max pool + kernel SVM</strong></td>
<td><strong>69.03±1.17</strong></td>
<td><strong>75.72±1.06</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SP + sparse codes + max pool + linear</strong></td>
<td><strong>68.78 ± 1.09</strong></td>
<td><strong>75.14 ± 0.86</strong></td>
</tr>
</tbody>
</table>
Splitting the Sparse Coding into Clusters

[Boureau, et al. 2011]

<table>
<thead>
<tr>
<th>$p$</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>$1+4$</th>
<th>$1+16$</th>
<th>$1+64$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Caltech-101

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
$k = 256$ & 70.5 ± 0.8 & 72.6 ± 1.0 & 74.0 ± 1.0 & 75.0 ± 0.8 & 72.5 ± 1.0 & 74.2 ± 1.1 & 75.6 ± 0.6 \\
\hline
$k = 1024$ & 75.6 ± 0.9 & 76.0 ± 1.2 & 76.3 ± 1.1 & 76.2 ± 0.8 & 76.3 ± 1.2 & 76.9 ± 1.0 & 77.3 ± 0.6 \\
\hline
\end{tabular}

Scenes

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
$k = 256$ & 78.8 ± 0.6 & 80.9 ± 0.7 & 81.5 ± 0.8 & 81.1 ± 0.5 & 80.8 ± 0.8 & 81.5 ± 0.8 & 81.9 ± 0.7 \\
\hline
$k = 1024$ & 82.7 ± 0.7 & 83.0 ± 0.7 & 82.7 ± 0.9 & 81.4 ± 0.7 & 83.3 ± 0.8 & 83.3 ± 1.0 & 83.1 ± 0.7 \\
\hline
\end{tabular}

Table 2. Results on Caltech 101 (30 training examples per class), and 15-Scenes (100 training examples per class) as a function of $k$: size of the codebook for sparse coding, and $p$: number of clusters extracted on the input data. Macrofeatures extracted every 4 pixels for Caltech, every 8 pixels for the Scenes.
[Kavukcuoglu et al. NIPS 2010]: convolutional PSD

[Zeiler, Krishnan, Taylor, Fergus, CVPR 2010]: Deconvolutional Network
[Lee, Gross, Ranganath, Ng, ICML 2009]: Convolutional Boltzmann Machine
[Norouzi, Ranjbar, Mori, CVPR 2009]: Convolutional Boltzmann Machine
[Chen, Sapiro, Dunson, Carin, Preprint 2010]: Deconvolutional Network with automatic adjustment of code dimension.
**Problem:**

- With patch-level training, the learning algorithm must reconstruct the entire patch with a single feature vector.
- But when the filters are used convolutionally, neighboring feature vectors will be highly redundant.

Patch-level training produces lots of filters that are shifted versions of each other.
Convolutional Sparse Coding

- Replace the dot products with dictionary element by convolutions.
  - Input $Y$ is a full image
  - Each code component $Z_k$ is a feature map (an image)
  - Each dictionary element is a convolution kernel

Regular sparse coding

$$E(Y, Z) = \|Y - \sum_k W_k Z_k\|^2 + \alpha \sum_k |Z_k|$$

Convolutional S.C.

$$E(Y, Z) = \|Y - \sum_k W_k * Z_k\|^2 + \alpha \sum_k |Z_k|$$

"deconvolutional networks" [Zeiler, Taylor, Fergus CVPR 2010]
**Convolutional Formulation**

- Extend sparse coding from **PATCH** to **IMAGE**

\[
\mathcal{L}(x, z, D) = \frac{1}{2} \| x - \sum_{k=1}^{K} D_k * z_k \|_2^2 + \sum_{k=1}^{K} \| z_k - f(W_k^k * x) \|_2^2 + |z|_1
\]

- **PATCH** based learning
- **CONVOLUTIONAL** learning
Convolutional Formulation

- Efficient Training using 2\textsuperscript{nd} order derivative approximation
- Especially important for training an encoder function
- Encoder with smooth shrinkage non-linearity
Filters and Basis Functions obtained with 16, 32, and 64 filters.
- Smooth shrinkage encoder, coordinate gradient descent inference
Convolutional PSD: Second Stage

- Second Stage Filters (encoder) and Basis Functions (decoder)

- **ENCODER**

- **DECODER**
**Performance on Caltech-101**

- Significant Improvement on 1\textsuperscript{st} layer from *Patch* to *Convolutional*
- 1\textsuperscript{st} layer is closest to input and convolutional training is most effective

<table>
<thead>
<tr>
<th></th>
<th>Patch</th>
<th>Convolutional</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 stage, unsp: U</td>
<td>52.2%</td>
<td>57.1%</td>
</tr>
<tr>
<td>1 stage, unsp+sup: U(^+)</td>
<td>54.2%</td>
<td>57.6%</td>
</tr>
<tr>
<td>2 stages, unsp: UU</td>
<td>63.7%</td>
<td>65.3%</td>
</tr>
<tr>
<td>2 stages, unsp+sup: U(^+)U(^+)</td>
<td>65.5%</td>
<td>66.3%</td>
</tr>
</tbody>
</table>
Cifar-10 Dataset

- **Dataset of tiny images**
  - Images are 32x32 color images
  - 10 object categories with 50000 training and 10000 testing

- **Example Images**
Architecture of Network

First Stage:
- Filters Y: 96 7x7 kernels: 64 Y, 16 U, and 16 V
- Pooling: 4x4 averaging with 2x2 subsample
- Features: 64 feature maps size 12x12
- Filters are learned convolutionally with DPSD

Second Stage:
- Filters: 2048 7x7 kernels
- Pooling: 3x3 averaging no downsampling
- Features: 128 feature maps of size 4x4
- Filters are learned convolutionally with DPSD

Chrominance Filters (Cr, Cb)
## Comparative Results on Cifar-10 Dataset

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Protocol</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 linear combinations [25]</td>
<td></td>
<td>36.0%</td>
</tr>
<tr>
<td>10k GRBM, 1 layer with fine-tuning [10]</td>
<td></td>
<td>64.8%</td>
</tr>
<tr>
<td>mcRBM-DBN(11025-8192-8192) [25]</td>
<td></td>
<td>71.0%</td>
</tr>
<tr>
<td>PCA(512)-iLCC(4096)-SVM [30]</td>
<td></td>
<td>74.5%</td>
</tr>
<tr>
<td>(1) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>RR</td>
<td>47.5%</td>
</tr>
<tr>
<td>(2) $F_{tanh} - R_{abs} - P_{M} - N$</td>
<td>R$^+$$R^+$</td>
<td>70.0%</td>
</tr>
<tr>
<td>(3) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>R$^+$$R^+$</td>
<td>70.5%</td>
</tr>
<tr>
<td>(4) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>$D_c^+$</td>
<td>59.6%</td>
</tr>
<tr>
<td>(5) $F_{si} - R_{abs} - N - P_A$</td>
<td>$D_c^+$</td>
<td>60.0%</td>
</tr>
<tr>
<td>(6) $F_{tanh} - R_{abs} - P_{M} - N$</td>
<td>$U_c^+$$U^+$</td>
<td>74.7%</td>
</tr>
<tr>
<td>(7) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>$U_c^+$$U^+$</td>
<td>74.8%</td>
</tr>
<tr>
<td>(8) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>$D^+$$D^+$</td>
<td>74.4%</td>
</tr>
<tr>
<td>(9) $F_{si} - R_{abs} - N - P_A$</td>
<td>$D_c^+$$D^+$</td>
<td>75.0%</td>
</tr>
<tr>
<td>(10) $F_{si} - R_{abs} - P_{M} - N$</td>
<td>$D_c^+$$D^+$</td>
<td>77.6%</td>
</tr>
</tbody>
</table>


**Ranzato and Hinton. Modeling pixel means and covariances using a factorized third order boltzmann machine. CVPR 2010

Yann LeCun
Pedestrian Detection (INRIA Dataset)

[Kavukcuoglu et al. NIPS 2010]
Pedestrian Detection: Examples

Kavukcuoglu et al. NIPS 2010
Learning Complex Cells with Invariance Properties Using Group Sparsity

[Kavukcuoglu et al. CVPR 2008]
Unsupervised PSD ignores the spatial pooling step.

Could we devise a similar method that learns the pooling layer as well?

Idea [Hyvarinen & Hoyer 2001]: group sparsity on pools of features
  - Minimum number of pools must be non-zero
  - Number of features that are on within a pool doesn't matter
  - Pools tend to regroup similar features

\[
\begin{align*}
\|Y^i - \tilde{Y}\|^2 \\
W_d Z \\
g_e(W_e, Y^i) \\
\|Z - \tilde{Z}\|^2 \\
\sqrt{\sum_{(k \in P_j)} Z_k^2} \\
\lambda \sum_j .
\end{align*}
\]
Learning the filters and the pools

Using an idea from Hyvarinen: topographic square pooling (subspace ICA)

1. Apply filters on a patch (with suitable non-linearity)
2. Arrange filter outputs on a 2D plane
3. Square filter outputs
4. Minimize sqrt of sum of blocks of squared filter outputs

Units in the code Z

Define pools and enforce sparsity across pools
Learning the filters and the pools

- The filters arrange themselves spontaneously so that similar filters enter the same pool.
- The pooling units can be seen as complex cells.
- They are invariant to local transformations of the input:
  - For some it's translations, for others rotations, or other transformations.
Pinwheels?
Invariance Properties Compared to SIFT

- Measure distance between feature vectors (128 dimensions) of 16x16 patches from natural images
  - Left: normalized distance as a function of translation
  - Right: normalized distance as a function of translation when one patch is rotated 25 degrees.

- Topographic PSD features are more invariant than SIFT
Learning Invariant Features

**Recognition Architecture**
- \( \rightarrow \text{HPF/LCN} \rightarrow \text{filters} \rightarrow \text{tanh} \rightarrow \text{sqr} \rightarrow \text{pooling} \rightarrow \text{sqr} \rightarrow \text{Classifier} \)
- Block pooling plays the same role as rectification

---

Yann LeCun

New York University
Recognition Accuracy on Caltech 101

- A/B Comparison with SIFT (128x34x34 descriptors)
- 32x16 topographic map with 16x16 filters
- Pooling performed over 6x6 with 2x2 subsampling
- 128 dimensional feature vector per 16x16 patch
- Feature vector computed every 4x4 pixels (128x34x34 feature maps)
- Resulting feature maps are spatially smoothed

<table>
<thead>
<tr>
<th>Method</th>
<th>Av. Accuracy/Class (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local norm_{5x5} + boxcar_{5x5} + PCA_{3060} + linear SVM</td>
<td></td>
</tr>
<tr>
<td>IPSD (24x24)</td>
<td>50.9</td>
</tr>
<tr>
<td>SIFT (24x24) (non rot. inv.)</td>
<td>51.2</td>
</tr>
<tr>
<td>SIFT (24x24) (rot. inv.)</td>
<td>45.2</td>
</tr>
<tr>
<td>Serre et al. features [25]</td>
<td>47.1</td>
</tr>
<tr>
<td>local norm_{9x9} + Spatial Pyramid Match Kernel SVM</td>
<td></td>
</tr>
<tr>
<td>SIFT [11]</td>
<td>64.6</td>
</tr>
<tr>
<td>IPSD (34x34)</td>
<td>59.6</td>
</tr>
<tr>
<td>IPSD (56x56)</td>
<td>62.6</td>
</tr>
<tr>
<td>IPSD (120x120)</td>
<td>65.5</td>
</tr>
</tbody>
</table>
## Recognition Accuracy on Tiny Images & MNIST

- A/B Comparison with SIFT (128x5x5 descriptors)
- 32x16 topographic map with 16x16 filters.

### Performance on Tiny Images Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPSD (5x5)</td>
<td>54</td>
</tr>
<tr>
<td>SIFT (5x5) (non rot. inv.)</td>
<td>53</td>
</tr>
</tbody>
</table>

### Performance on MNIST Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Error Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPSD (5x5)</td>
<td>1.0</td>
</tr>
<tr>
<td>SIFT (5x5) (non rot. inv.)</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Learning fields of Simple Cells and Complex Cells

[Gregor and LeCun, arXiv.org 2010]
Training Simple Cells with Local Receptive Fields over Large Input Images

Training on 115x115 images. Kernels are 15x15
Simple Cells + Complex Cells with Sparsity Penalty: Pinwheels

Training on 115x115 images. Kernels are 15x15
119x119 Image Input
100x100 Code
20x20 Receptive field size
sigma=5

Michael C. Crair, et. al. The Journal of Neurophysiology
Vol. 77 No. 6 June 1997, pp. 3381-3385 (Cat)

K Obermayer and GG Blasdel, Journal of Neuroscience, Vol 13, 4114-4129 (Monkey)
Same Method, with Training at the Image Level (vs patch)

Color indicates orientation (by fitting Gabors)
Recognizing Activities
In Videos

[Taylor, Fergus, LeCun, Bregler ECCV 2010]
**Convolutional Gated RBM**: takes two successive frames as input and automatically learns motion features

- Feature encode the transformation from the first frame to the second frame
- Trained in unsupervised mode to predict the second frame

**The rest is a 3D (spatio-temporal) convolutional network**
- Trained in supervised mode with sparse coding
Gated RBM

\[ E(y, z; x) = - \sum_{ijk} W_{ijk} x_i y_j z_k \]

\[ p(y, z|x) = \frac{1}{Z(x)} \exp(-E(y, z; x)) \]

\[ E(y, z; x) = \frac{1}{2\sigma^2} \sum_j (y_j - a_j)^2 - \frac{1}{\sigma} \sum_{ijk} W_{ijk} x_i y_j z_k - \sum_k b_k z_k \]

In practice, we use this energy (real valued outputs, biases)

\[ p(z_k = 1|x, y) = \frac{1}{1 + \exp\left(-\sum_{ij} W_{ijk} x_i y_j\right)} \]
Convolutional Gated RBM (ConvGRBM)

Parameters (filter weights) are a 4D tensor

As in convolutional RBM (Lee et. al 2009)
KTH Action Dataset

- Available since 2004
- 160x120 greyscale
- 25 subjects performing 6 actions: walking, jogging, running, boxing, hand waving and hand clapping
- 4 scenarios: outdoors, outdoors with scale variation, outdoors with different clothes and indoors
- 2 popular evaluation schemes
Features Learned by ConvGRBM on KTH

Time →

Feature ($\mathbf{z}^k$)

Hand clapping

Walking

Boxing

Jogging
Hollywood 2 Dataset

[Laptev 2008]

- 12 classes of human actions, 10 class of scenes
- 3669 video clips (30s-2m); 20.1h total video
- Captured from 69 Hollywood movies
- Samples may contain instances of several actions
- Contains an “automatic” and “human” labeled training sets

<table>
<thead>
<tr>
<th>Actions</th>
<th>Training subset (clean)</th>
<th>Training subset (automatic)</th>
<th>Test subset (clean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnswerPhone</td>
<td>66</td>
<td>59</td>
<td>64</td>
</tr>
<tr>
<td>DriveCar</td>
<td>85</td>
<td>90</td>
<td>102</td>
</tr>
<tr>
<td>Eat</td>
<td>40</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>FightPerson</td>
<td>54</td>
<td>33</td>
<td>70</td>
</tr>
<tr>
<td>GetOutCar</td>
<td>51</td>
<td>40</td>
<td>57</td>
</tr>
<tr>
<td>HandShake</td>
<td>32</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td>HugPerson</td>
<td>64</td>
<td>27</td>
<td>66</td>
</tr>
<tr>
<td>Kiss</td>
<td>114</td>
<td>125</td>
<td>103</td>
</tr>
<tr>
<td>Run</td>
<td>135</td>
<td>187</td>
<td>141</td>
</tr>
<tr>
<td>SitDown</td>
<td>104</td>
<td>87</td>
<td>108</td>
</tr>
<tr>
<td>SitUp</td>
<td>24</td>
<td>26</td>
<td>37</td>
</tr>
<tr>
<td>StandUp</td>
<td>132</td>
<td>133</td>
<td>146</td>
</tr>
<tr>
<td>All Samples</td>
<td>823</td>
<td>810</td>
<td>884</td>
</tr>
</tbody>
</table>
Hollywood 2 Architecture

- ConvGRBM
- Sparse coding
- Max pooling
- SVM
## Results

Compared to other methods using dense sampling (no interest points):

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOG3D-KM-SVM</td>
<td>85.3</td>
</tr>
<tr>
<td>HOG/HOF-KM-SVM</td>
<td>86.1</td>
</tr>
<tr>
<td>HOG-KM-SVM</td>
<td>79.0</td>
</tr>
<tr>
<td>HOF-KM-SVM</td>
<td>88.0</td>
</tr>
<tr>
<td>32GRBM-KM-SVM</td>
<td>88.3</td>
</tr>
<tr>
<td>32GRBM-SC-SVM</td>
<td>89.1</td>
</tr>
<tr>
<td>32convGRBM-3Dconvnet-LR</td>
<td>88.9</td>
</tr>
<tr>
<td>32convGRBM-3Dconvnet-MLP</td>
<td>90.0</td>
</tr>
</tbody>
</table>

State-of-the-art: 91.8% (Laptev et al. 2008)
Uses explicit interest-point detection

Dense sampling actually works better:

<table>
<thead>
<tr>
<th>Method</th>
<th>Mean AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOG3D-KM-SVM</td>
<td>45.3</td>
</tr>
<tr>
<td>HOG/HOF-KM-SVM</td>
<td>47.4</td>
</tr>
<tr>
<td>HOG-KM-SVM</td>
<td>39.4</td>
</tr>
<tr>
<td>HOF-KM-SVM</td>
<td>45.5</td>
</tr>
<tr>
<td>convGRBM+SC+SVM</td>
<td>46.8</td>
</tr>
</tbody>
</table>

We also outperform Cuboids (45.0%) and Willems et al (38.2%)
Deep Learning for Mobile Robot Vision
Getting a robot to drive autonomously in unknown terrain solely from vision (camera input).

Our team (NYU/Net-Scale Technologies Inc.) was one of 8 participants funded by DARPA.

All teams received identical robots and can only modify the software (not the hardware).

The robot is given the GPS coordinates of a goal, and must drive to the goal as fast as possible. The terrain is unknown in advance. The robot is run 3 times through the same course.

Long-Range Obstacle Detection with online, self-trained ConvNet

Uses temporal consistency!
Obstacle Detection

Obstacles overlaid with camera image

Camera image

Detected obstacles (red)
Navigating to a goal is hard...

especially in a snowstorm.

corrected by stereo perspective

corrected by human perspective

especially in a snowstorm.
Self-Supervised Learning

- Stereo vision tells us what nearby obstacles look like
- Use the labels (obstacle/traversable) produced by stereo vision to train a monocular neural network
- Self-supervised “near to far” learning
Long Range Vision: Distance Normalization

Pre-processing (125 ms)
- Ground plane estimation
- Horizon leveling
- Conversion to YUV + local contrast normalization
- Scale invariant pyramid of distance-normalized image “bands”
Convolutional Net Architecture

Operates on 12x25 YUV windows from the pyramid

100 features per
3x12x25 input window

Convolutions with 6x5 kernels

Pooling/subsampling with 1x4 kernels

Convolutions with 7x6 kernels

YUV image band
20-36 pixels tall,
36-500 pixels wide

Logistic regression 100 features -> 5 classes

100 features per
100x1x1 input window

20x6x5 input window

20x6x20 input window

3x12x25 input window
Convolutional Net Architecture

YUV input

<table>
<thead>
<tr>
<th>YUV input</th>
</tr>
</thead>
<tbody>
<tr>
<td>3@36x484</td>
</tr>
<tr>
<td>20@30x484</td>
</tr>
<tr>
<td>20@30x125</td>
</tr>
<tr>
<td>100@25x121</td>
</tr>
</tbody>
</table>

CONVOLUTIONS (6x5)

MAX SUBSAMPLING (1x4)

CONVOLUTIONS (7x6)
Long Range Vision: 5 categories

Online Learning (52 ms)

- Label windows using stereo information – 5 classes

![Diagram showing categories super-ground, ground, footline, obstacle, super-obstacle]
**Trainable Feature Extraction**

- “Deep belief net” approach to unsupervised feature learning

- Two stages are trained in sequence
  - Each stage has a layer of convolutional filters and a layer of horizontal feature pooling.
  - Naturally shift invariant in the horizontal direction

- Filters of the convolutional net are trained so that the input can be reconstructed from the features
  - 20 filters at the first stage (layers 1 and 2)
  - 300 filters at the second stage (layers 3 and 4)

- Scale invariance comes from pyramid.
  - For near-to-far generalization
Long Range Vision Results
Long Range Vision Results

Input image
Stereo Labels
Classifier Output

Input image
Stereo Labels
Classifier Output
Long Range Vision Results
Vehicle Map (Hyperbolic Polar map)

Legend
Goal
Path Planning
Trajectories
Traversable
Uncertain
Quasi-Lethal
Lethal
Bumper/Stuck
Unseen

Cost Map
(FastOD & FarOD)

RGB Map
(FarOD Only)

FarOD Neural Network Labels

FarOD Stereo: Input labels to Neural Network
Video Results

Vehicle Map (Hyperbolic Polar map)

Legend
- Goal
- Path Planning
- Traversable
- Uncertain
- Quasi-Lethal
- Lethal
- Bumper/Stuck
- Unseen

Cost Map
- 25m
- 50m
- 100m
- 200m

RGB Map
- Goal
- FastOD
- FarOD Only

FarOD Neural Network Labels

FarOD Stereo: Input labels to Neural Network
Feature Learning for traversability prediction (LAGR)

Comparing
- purely supervised
- stacked, invariant auto-encoders
- DrLIM invariant learning

Testing on hand-labeled groundtruth frames – binary labels

Comparison of Feature Extractors on Groundtruth Data

<table>
<thead>
<tr>
<th>Feature Extractor</th>
<th>Error Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbf</td>
<td>25</td>
</tr>
<tr>
<td>supervised</td>
<td>22.5</td>
</tr>
<tr>
<td>autoencoder</td>
<td>20</td>
</tr>
<tr>
<td>autoenc + sup</td>
<td>21</td>
</tr>
<tr>
<td>DrLIM</td>
<td>15</td>
</tr>
<tr>
<td>DrLIM + sup</td>
<td>12.5</td>
</tr>
<tr>
<td>No learning</td>
<td>10</td>
</tr>
</tbody>
</table>

AVERAGE: 12.5
The End