Deep Learning for Vision
BMVC 2013

Adam Coates
Stanford University
(Visiting Scholar: Indiana University, Bloomington)
What do we want ML to do?

• Given image, predict complex high-level patterns:

  “Cat”

Object recognition
What do we want ML to do?

• Given image, predict complex high-level patterns:

“Cat”

Object recognition

Detection
What do we want ML to do?

• Given image, predict complex high-level patterns:

  “Cat”

  Object recognition

  Detection

  Segmentation

  [Martin et al., 2001]
How is ML done?

- Machine learning often uses common pipeline with hand-designed feature extraction.
  - Final ML algorithm learns to make decisions starting from the higher-level representation.
  - Sometimes layers of increasingly high-level abstractions.  
    - Constructed using prior knowledge about problem domain.

```
Feature Extraction → Machine Learning Algorithm

Prior Knowledge, Experience
```

“Cat”?
“Deep Learning”

- Deep Learning
  - Train *multiple layers* of features/abstractions from data.
  - Try to discover *representation* that makes decisions easy.

Deep Learning: train layers of features so that classifier works well.

“Cat”?
“Deep Learning”

• Why do we want “deep learning”? 
  – Some decisions require many stages of processing. 
    • Easy to invent cases where a “deep” model is compact but a shallow model is very large / inefficient.
  – We already, intuitively, hand-engineer “layers” of representation.
    • Let’s replace this with something automated!
  – Algorithms scale well with data and computing power.
    • In practice, one of the most consistently successful ways to get good results in ML.
    • Can try to take advantage of unlabeled data to learn representations before the task.
Have we been here before?

➤ Yes.
  – Basic ideas common to past ML and neural networks research.
    • Supervised learning is straight-forward.
    • Standard ML development strategies still relevant.
    • Some knowledge carried over from problem domains.

➤ No.
  – Faster computers; more data.
  – Better optimizers; better initialization schemes.
    • “Unsupervised pre-training” trick
      [Hinton et al. 2006; Bengio et al. 2006]
  – Lots of empirical evidence about what works.
    • Made useful by ability to “mix and match” components.
      [See, e.g., Jarrett et al., ICCV 2009]
Real impact

• DL systems are high performers in many tasks over many domains.

Image recognition  [E.g., Krizhevsky et al., 2012]

Speech recognition  [E.g., Heigold et al., 2013]

NLP  [E.g., Socher et al., ICML 2011; Collobert & Weston, ICML 2008]
Outline

• ML refresher / crash course
  – Logistic regression
  – Optimization
  – Features

• Supervised deep learning
  – Neural network models
  – Back-propagation
  – Training procedures

• Supervised DL for images
  – Neural network architectures for images.
  – Application to Image-Net

• Debugging

• Unsupervised DL

• References / Resources
Outline

• ML refresher / crash course
• Supervised deep learning
• Supervised DL for images
• Debugging

• Unsupervised DL
  – Representation learning, unsupervised feature learning.
  – Greedy layer-wise training.
  – Example: sparse auto-encoders.
  – Other unsupervised learning algorithms.

• References / Resources
Supervised Learning

- Given *labeled* training examples:
  \[ \mathcal{X} = \{(x^{(i)}, y^{(i)}): i = 1, \ldots, m\} \]

- For instance: \(x^{(i)}\) = vector of pixel intensities.
  \(y^{(i)}\) = object class ID.

- Goal: find \(f(x)\) to predict \(y\) from \(x\) on training data.
  - Hopefully: learned predictor works on “test” data.
Logistic Regression

• Simple binary classification algorithm
  – Start with a function of the form:
    \[ f(x; \theta) \equiv \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)} \]
  – Interpretation: \( f(x) \) is probability that \( y = 1 \).
  • Sigmoid “nonlinearity” squashes linear function to [0,1].

\[ \begin{align*}
    \mathcal{L}(\theta) &= -\sum_{i=1}^{m} \left( 1\{y^{(i)} = 1\} \log(f(x^{(i)}; \theta)) + \mathbb{P}(y^{(i)} = 1|x^{(i)}) \right. \\
    &\quad \left. - 1\{y^{(i)} = 0\} \log(1 - f(x^{(i)}; \theta)) + \mathbb{P}(y^{(i)} = 0|x^{(i)}) \right)
\end{align*} \]
Optimization

• How do we tune $\theta$ to minimize $\mathcal{L}(\theta)$?
• One algorithm: gradient descent
  – Compute gradient:
    \[
    \nabla_\theta \mathcal{L}(\theta) = \sum_{i}^{m} x^{(i)} \cdot (y^{(i)} - f(x^{(i)}; \theta))
    \]
  – Follow gradient “downhill”:
    \[
    \theta := \theta - \eta \nabla_\theta \mathcal{L}(\theta)
    \]
• Stochastic Gradient Descent (SGD): take step using gradient from only small batch of examples.
  – Scales to larger datasets.  [Bottou & LeCun, 2005]
Is this enough?

• Loss is convex $\Rightarrow$ we always find minimum.
• Works for simple problems:
  – Classify digits as 0 or 1 using pixel intensity.
  – Certain pixels are highly informative --- e.g., center pixel.
  ![00000 1111]
• Fails for even slightly harder problems.
  – Is this a coffee mug?
Why is vision so hard?

“Coffee Mug”

Pixel Intensity

Pixel intensity is a very poor representation.
Why is vision so hard?

Pixel 1

[72 160] Pixel Intensity

Pixel 2

+ Coffee Mug

- Not Coffee Mug
Why is vision so hard?

+ Coffee Mug

- Not Coffee Mug
Why is vision so hard?

Learning Algorithm

Is this a Coffee Mug?

+ Coffee Mug

- Not Coffee Mug
Features

handle?  cylinder?
Features

handle?  cylinder?

cylinder?

handle?

+ Coffee Mug

- Not Coffee Mug
Features

Is this a Coffee Mug?

Learning Algorithm

+ Coffee Mug

- Not Coffee Mug
Features

• Features are usually hard-wired transformations built into the system.
  – Formally, a function that maps raw input to a “higher level” representation.
    \[ \Phi(x) : \mathbb{R}^n \rightarrow \mathbb{R}^K \]
  – Completely static --- so just substitute \( \varphi(x) \) for \( x \) and do logistic regression like before.
Features

• Features are usually hard-wired transformations built into the system.
  – Formally, a function that maps raw input to a “higher level” representation.
    \[ \Phi(x) : \mathbb{R}^n \to \mathbb{R}^K \]
  – Completely static --- so just substitute \( \varphi(x) \) for \( x \) and do logistic regression like before.

Where do we get good features?
Features

• Huge investment devoted to building application-specific feature representations.
  – Find higher-level patterns so that final decision is easy to learn with ML algorithm.

Object Bank [Li et al., 2010]

Super-pixels
[Gould et al., 2008, Ren & Malik, 2003]

SIFT [Lowe, 1999]

Spin Images [Johnson & Hebert, 1999]
SUPERVISED DEEP LEARNING

Extension to neural networks
Basic idea

• We saw how to do supervised learning when the “features” $\phi(x)$ are fixed.
Basic idea

• We saw how to do supervised learning when the “features” \( \phi(x) \) are fixed.
  
  – Let’s extend to case where features are given by tunable functions with their own parameters.

\[
P(y = 1|x) = f(x; \theta, W) = \sigma(\theta^\top \sigma(Wx))
\]
Basic idea

- We saw how to do supervised learning when the “features” $\phi(x)$ are fixed.
- Let’s extend to case where features are given by tunable functions with their own parameters.

$$P(y = 1|x) = f(x; \theta, W) = \sigma(\theta^\top \sigma(Wx))$$

Outer part of function is same as logistic regression.

Inputs are “features”---one feature for each row of $W$:

$$\begin{bmatrix}
\sigma(w_1 x) \\
\sigma(w_2 x) \\
\vdots \\
\sigma(w_K x)
\end{bmatrix}$$
Basic idea

• To do supervised learning for two-class classification, minimize:

\[ \mathcal{L}(\theta, W) = -\sum_{i=1}^{m} 1\{y^{(i)} = 1\} \log(f(x^{(i)}; \theta, W)) + 1\{y^{(i)} = 0\} \log(1 - f(x^{(i)}; \theta, W)) \]

• Same as logistic regression, but now \( f(x) \) has multiple stages (“layers”, “modules”):

\[ f(x; \theta, W) = \sigma(\theta^\top \sigma(Wx)) \]
Basic idea

• To do supervised learning for two-class classification, minimize:

\[
\mathcal{L}(\theta, W) = - \sum_{i} 1\{y^{(i)} = 1\} \log(f(x^{(i)}; \theta, W)) + 1\{y^{(i)} = 0\} \log(1 - f(x^{(i)}; \theta, W))
\]

• Same as logistic regression, but now \( f(x) \) has multiple stages ("layers", "modules"):

\[
f(x; \theta, W) = \sigma(\theta^\top \sigma(Wx))
\]

![Diagram showing intermediate representation and prediction for \( \mathbb{P}(y = 1|x) \)]
Neural network

• This model is a sigmoid “neural network”:

\[ \mathcal{L}(\theta, W) \]

\[ y \]
Neural network

• This model is a sigmoid “neural network”:

\[
\mathcal{L}(\theta, W) \rightarrow y
\]
Neural network

• This model is a sigmoid “neural network”:

\[ \mathcal{L}(\theta, W) \]

Flow of computation.
“Forward prop”
Neural network

• This model is a sigmoid “neural network”:

Flow of computation. “Forward prop”
Neural network

• This model is a sigmoid “neural network”:

Flow of computation. “Forward prop”
Neural network

- Can stack up several layers:

Must learn multiple stages of internal “representation”.

\[
\begin{align*}
&h'_1 & \quad h'_2 & \quad \cdots & \quad h'_K \\
&h_1 & \quad h_2 & \cdots & \quad h_K \\
x_1 & \quad x_2 & \cdots & \quad x_n
\end{align*}
\]
Neural network

• Can stack up several layers:

\[ f(x; \theta, W_1, W_2) = \sigma(\theta^T \sigma(W_2 \sigma(W_1 x))) \]

Must learn multiple stages of internal "representation".
Neural network

- Can stack up several layers:

Must learn multiple stages of internal “representation”.

\[ x \xrightarrow{\sigma(W_1 x)} h \xrightarrow{\sigma(W_2 h)} h' \xrightarrow{\sigma(\theta^\top h')} f \]
Back-propagation

• Minimize:

\[ \mathcal{L}(\theta, W) = - \sum_{i}^{m} 1\{y^{(i)} = 1\} \log(f(x^{(i)}; \theta, W)) + 1\{y^{(i)} = 0\} \log(1 - f(x^{(i)}; \theta, W)) \]

• To minimize \( \mathcal{L}(\theta, W) \) we need gradients:

\[ \nabla_{\theta} \mathcal{L}(\theta, W) \text{ and } \nabla_{W} \mathcal{L}(\theta, W) \]

– Then use gradient descent algorithm as before.

• Formula for \( \nabla_{\theta} \mathcal{L}(\theta, W) \) can be found by hand (same as before); but what about \( W \)?
The Chain Rule

- Suppose we have a module that looks like:

\[
\begin{align*}
&\xrightarrow{z} \quad h(z; W) \quad \xrightarrow{h} \\
&\xrightarrow{W} \\
\end{align*}
\]

\[\nabla_h \mathcal{L} \bigg|_j = \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \cdot \frac{\partial h_j}{\partial z_k} \]

.
The Chain Rule

- Suppose we have a module that looks like:

```
\[ z \rightarrow h(z; W) \rightarrow h \]
```

- And we know \( [\nabla_h \mathcal{L}]_j = \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \) and \( \frac{\partial h_j}{\partial z_k} \), chain rule gives:

\[
\frac{\partial \mathcal{L}(\theta, W)}{\partial z_k} = \sum_j \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \frac{\partial h_j}{\partial z_k} \implies \nabla_z \mathcal{L} = J_{h,z}(\nabla_h \mathcal{L})
\]

Jacobian matrix.
The Chain Rule

• Suppose we have a module that looks like:

\[ \begin{array}{c}
  \downarrow \ \\
  W \rightarrow h(z; W) \rightarrow h \\
  \downarrow \ \\
  z
\end{array} \]

• And we know \( \nabla_h \mathcal{L} \) and \( \frac{\partial h_j}{\partial z_k} \), chain rule gives:

\[
\frac{\partial \mathcal{L}(\theta, W)}{\partial z_k} = \sum_j \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \frac{\partial h_j}{\partial z_k} \Rightarrow \nabla_z \mathcal{L} = J_{h,z}(\nabla_h \mathcal{L})
\]

Similarly for \( W \):

\[
\frac{\partial \mathcal{L}(\theta, W)}{\partial W_{kl}} = \sum_j \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \frac{\partial h_j}{\partial W_{kl}} \Rightarrow \nabla_W \mathcal{L} = J_{h,W}(\nabla_h \mathcal{L})
\]
The Chain Rule

• Suppose we have a module that looks like:

![Diagram of module](image)

And we know \[ \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \] and \[ \frac{\partial h_j}{\partial z_k} \], chain rule gives:

\[
\frac{\partial \mathcal{L}(\theta, W)}{\partial z_k} = \sum_j \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \frac{\partial h_j}{\partial z_k} \implies \nabla_z \mathcal{L} = J_{h,z}(\nabla_h \mathcal{L})
\]

Similarly for \( W \):

\[
\frac{\partial \mathcal{L}(\theta, W)}{\partial W_{kl}} = \sum_j \frac{\partial \mathcal{L}(\theta, W)}{\partial h_j} \frac{\partial h_j}{\partial W_{kl}} \implies \nabla_W \mathcal{L} = J_{h,W}(\nabla_h \mathcal{L})
\]

- Given gradient with respect to output, we can build a new “module” that finds gradient with respect to inputs.
The Chain Rule

- Easy to build toolkit of known rules to compute gradients given $\delta \equiv \nabla_h \mathcal{L}$

<table>
<thead>
<tr>
<th>Function</th>
<th>Gradient w.r.t. input</th>
<th>Gradient w.r.t. parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(z)$</td>
<td>$\nabla_z \mathcal{L}$</td>
<td>$\nabla_W \mathcal{L}$</td>
</tr>
<tr>
<td>$h = Wz$</td>
<td>$W^\top \delta$</td>
<td>$\delta z^\top$</td>
</tr>
<tr>
<td>$h = \sigma(z)$</td>
<td>$\delta \odot \sigma(z) \odot (1 - \sigma(z))$</td>
<td></td>
</tr>
<tr>
<td>$h = \sqrt{Wz^2}$</td>
<td>$\left(W^\top \frac{\delta}{\delta h}\right) \odot z$</td>
<td>$\frac{\delta}{2h} (z^2)^\top$</td>
</tr>
<tr>
<td>$h = \max_j {z_j}$</td>
<td>$1{z_j = h} \delta$</td>
<td></td>
</tr>
</tbody>
</table>
## The Chain Rule

- Easy to build toolkit of known rules to compute gradients given $\delta \equiv \nabla_h \mathcal{L}$
  
  - Automated differentiation! E.g., Theano [Bergstra et al., 2010]

<table>
<thead>
<tr>
<th>Function</th>
<th>Gradient w.r.t. input</th>
<th>Gradient w.r.t. parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(z)$</td>
<td>$\nabla_z \mathcal{L}$</td>
<td>$\nabla_W \mathcal{L}$</td>
</tr>
<tr>
<td>$h = Wz$</td>
<td>$W^\top \delta$</td>
<td>$\delta \quad z^\top$</td>
</tr>
<tr>
<td>$h = \sigma(z)$</td>
<td>$\delta \circ \sigma(z) \circ (1 - \sigma(z))$</td>
<td></td>
</tr>
<tr>
<td>$h = \sqrt{Wz^2}$</td>
<td>$(W^\top \frac{\delta}{h}) \circ z$</td>
<td>$\frac{\delta}{2h} (z^2)^\top$</td>
</tr>
<tr>
<td>$h = \max_j {z_j}$</td>
<td>$1{z_j = h} \delta$</td>
<td></td>
</tr>
</tbody>
</table>
Back-propagation

• Can re-apply chain rule to get gradients for all intermediate values and parameters.

\[
x \xrightarrow{W} \sigma(Wx) \xrightarrow{h} \sigma(\theta^\top h) \xrightarrow{f} L(\theta, W)
\]

\[
\nabla_x L \xleftarrow{J_h} \nabla_h L \xleftarrow{J_f} \nabla_f L
\]

“Backward” modules for each forward stage.
Example

• Given $\nabla_f \mathcal{L}$, compute $\nabla_W \mathcal{L}$:

Using several items from our table:

$$\nabla_h \mathcal{L} = \theta[f(1 - f)(\nabla_f \mathcal{L})]$$

$$\nabla_W \mathcal{L} = [h \odot (1 - h) \odot (\nabla_h \mathcal{L})] x^\top$$
Training Procedure

• Collect labeled training data
  – For SGD: Randomly shuffle after each epoch!
    \[ \mathcal{X} = \{(x^{(i)}, y^{(i)}) : i = 1, \ldots, m\} \]

• For a batch of examples:
  – Compute gradient w.r.t. all parameters in network.
    \[ \Delta_\theta := \nabla_\theta \mathcal{L}(\theta, W) \]
    \[ \Delta_W := \nabla_W \mathcal{L}(\theta, W) \]
  – Make a small update to parameters.
    \[ \theta := \theta - \eta_\theta \Delta_\theta \]
    \[ W := W - \eta_W \Delta_W \]
  – Repeat until convergence.
Training Procedure

• Historically, this has not worked so easily.
  – Non-convex: Local minima; convergence criteria.
  – Optimization becomes difficult with many stages.
    • “Vanishing gradient problem”
  – Hard to diagnose and debug malfunctions.
Training Procedure

• Historically, this has not worked so easily.
  – Non-convex: Local minima; convergence criteria.
  – Optimization becomes difficult with many stages.
    • “Vanishing gradient problem”
  – Hard to diagnose and debug malfunctions.

• Many things turn out to matter:
  – Choice of nonlinearities.
  – Initialization of parameters.
  – Optimizer parameters: step size, schedule.
Nonlinearities

• Choice of functions inside network matters.
  – Sigmoid function turns out to be difficult.
  – Some other choices often used:

  \[
  \text{tanh}(z), \quad \text{abs}(z), \quad \text{ReLu}(z) = \max\{0, z\}
  \]

  “Rectified Linear Unit” → Increasingly popular.

  [Nair & Hinton, 2010]
Initialization

- Usually small random values.
  - Try to choose so that typical input to a neuron avoids saturating / non-differentiable areas.
  - Occasionally inspect units for saturation / blowup.
  - Larger values may give faster convergence, but worse models!

- Initialization schemes for particular units:
  - tanh units: Unif[-r, r]; sigmoid: Unif[-4r, 4r].

\[
r = \sqrt{6/(\text{fan-in} + \text{fan-out})}
\]

See [Glorot et al., AISTATS 2010]

- Later in this tutorial: unsupervised pre-training.
Optimization: Step sizes

- Choose SGD step size carefully.
  - Up to factor $\sim 2$ can make a difference.
- Strategies:
  - Brute-force: try many; pick one with best result.
  - Choose so that typical “update” to a weight is roughly $1/1000$ times weight magnitude. [Look at histograms.]
    - Smaller if fan-in to neurons is large.
  - Racing: pick size with best error on validation data after $T$ steps.
    - Not always accurate if $T$ is too small.
- Step size schedule:
  - Simple $1/t$ schedule:
    $$ \eta_t = \frac{\eta_0 \tau}{\max\{\tau, t\}} $$
  - Or: fixed step size. But if little progress is made on objective after $T$ steps, cut step size in half.

Optimization: Momentum

• “Smooth” estimate of gradient from several steps of SGD:

\[ v := \mu v + \epsilon_t \nabla \theta \mathcal{L}(\theta) \]

\[ \theta := \theta + v \]

Optimization: Momentum

• “Smooth” estimate of gradient from several steps of SGD:

\[ v := \mu v + \epsilon_t \nabla_{\theta} L(\theta) \]

\[ \theta := \theta + v \]

• A little bit like second-order information.
  – High-curvature directions cancel out.
  – Low-curvature directions “add up” and accelerate.

Optimization: Momentum

• “Smooth” estimate of gradient from several steps of SGD:

\[ v := \mu v + \epsilon_t \nabla_\theta \mathcal{L}(\theta) \]

\[ \theta := \theta + v \]

– Start out with \( \mu = 0.5 \); gradually increase to 0.9, or 0.99 after learning is proceeding smoothly.
– Large momentum appears to help with hard training tasks.
– “Nesterov accelerated gradient” is similar; yields some improvement.

[Sutskever et al., ICML 2013]
Other factors

• “Weight decay” penalty can help.
  – Add small penalty for squared weight magnitude.

• For modest datasets, LBFGS or second-order methods are easier than SGD.
  – See, e.g.: Martens & Sutskever, ICML 2011.
  – Can crudely extend to mini-batch case if batches are large. [Le et al., ICML 2011]
SUPERVISED DL FOR VISION
Working with images

• Major factors:
  – Choose functional form of network to roughly match the computations we need to represent.
    • E.g., “selective” features and “invariant” features.
  – Try to exploit knowledge of images to accelerate training or improve performance.

• Generally try to avoid wiring detailed visual knowledge into system --- prefer to learn.
Local connectivity

• Neural network view of single neuron:

Extremely large number of connections.
→ More parameters to learn from.
→ Higher computational expense.
→ Turn out not to be helpful in practice.
Local connectivity

• Reduce parameters with local connections.
  – Weight vector is a spatially localized “filter”.

![Diagram showing local connectivity](image)
Local connectivity

- Sometimes think of neurons as viewing small adjacent windows.
  - Specify connectivity by the size ("receptive field" size) and spacing ("step" or "stride") of windows.
    - Typical RF size = 5 to 20
    - Typical step size = 1 pixel up to RF size.

Rows of $W$ are sparse. Only weights connecting to inputs in the window are non-zero.
Local connectivity

- Spatial organization of filters means output features can also be organized like an image.
  - X,Y dimensions correspond to X,Y position of neuron window.
  - “Channels” are different features extracted from same spatial location. (Also called “feature maps”, or “maps”.)

1-dimensional example:
Local connectivity

- We can treat output of a layer like an image and re-use the same tricks.

1-dimensional example:

X spatial location

“Channel” or “map” index

1D input
Weight-Tying

- Even with local connections, may still have too many weights.
  - Trick: constrain some weights to be equal if we know that some parts of input should learn same kinds of features.
  - Images tend to be “stationary”: different patches tend to have similar low-level structure.
    - Constrain weights used at different spatial positions to be the equal.
Weight-Tying

- Before, could have neurons with different weights at different locations. But can reduce parameters by making them equal.

1-dimensional example:

- Sometimes called a “convolutional” network. Each unique filter is spatially convolved with the input to produce responses for each map.
  [LeCun et al., 1989; LeCun et al., 2004]
Weight-Tying

- Before, could have neurons with different weights at different locations. But can reduce parameters by making them equal.

1-dimensional example:

- Sometimes called a “convolutional” network. Each unique filter is spatially convolved with the input to produce responses for each map.  
[LeCun et al., 1989; LeCun et al., 2004]
Pooling

• Functional layers designed to represent invariant features.
• Usually locally connected with specific nonlinearities.
  – Combined with convolution, corresponds to hard-wired translation invariance.
• Usually fix weights to local box or gaussian filter.
  – Easy to represent max-, average-, or 2-norm pooling.

\[ h = \left( W \mathcal{Z}^a \right)^{1/a} \]

[Scherer et al., ICANN 2010]
[Boureau et al., ICML 2010]
Contrast Normalization

• Empirically useful to soft-normalize magnitude of groups of neurons.
  – Sometimes we subtract out the local mean first.

\[ h = \frac{z}{\sqrt{W z^2 + \epsilon}} \]

[Jarrett et al., ICCV 2009]
Application: Image-Net

- System from Krizhevsky et al., NIPS 2012:
  - Convolutional neural network.
  - Max-pooling.
  - Rectified linear units (ReLu).
  - Contrast normalization.
  - Local connectivity.
Application: Image-Net

- Top result in LSVRC 2012: ~85%, Top-5 accuracy.
Application: Image-Net

- Top result in LSVRC 2012: ~85%, Top-5 accuracy.
More applications

• Segmentation: predict classes of pixels / super-pixels.
  
  Farabet et al., ICML 2012 →  
  Ciresan et al., NIPS 2012

• Detection: combine classifiers with sliding-window architecture.  
  – Economical when used with convolutional nets.

  Pierre Sermanet (2010) →

• Robotic grasping.  [Lenz et al., RSS 2013]

http://www.youtube.com/watch?v=f9CuzqI1SkE
DEBUGGING TIPS

YMMV
Getting the code right

• Numerical gradient check.

• Verify that objective function decreases on a small training set.
  – Sometimes reasonable to expect 100% classifier accuracy on small datasets with big model. If you can’t reach this, why not?

• Use off-the-shelf optimizer (e.g., LBFGS) with small model and small dataset to verify that your own optimizer reaches good solutions.
Bias vs. Variance

• After training, performance on test data is poor. What is wrong?
  – Training accuracy is an upper bound on expected test accuracy.
    • If gap is small, try to improve training accuracy:
      – A bigger model. (More features!)
      – Run optimizer longer or reduce step size to try to lower objective.
    • If gap is large, try to improve generalization:
      – More data.
      – Regularization.
      – Smaller model.
UNSUPERVISED DL
Representation Learning

• In supervised learning, train “features” to accomplish top-level objective.

But what if we have too few labels to train all these parameters?
Representation Learning

• Can we train the “representation” without using top-down supervision?

Learn a “good” representation directly?
Representation Learning

• What makes a good representation?
  – Distributed: roughly, $K$ features represents more than $K$ types of patterns.
    • E.g., $K$ binary features that can vary independently to represent $2^K$ patterns.
  – Invariant: robust to local changes of input; more abstract.
    • E.g., pooled edge features: detect edge at several locations.
  – Disentangling factors: put separate concepts (e.g., color, edge orientation) in separate features.

Bengio, Courville, and Vincent (2012)
Unsupervised Feature Learning

• Train representations with unlabeled data.
  – Minimize an *unsupervised* training loss.
    • Often based on generic priors about characteristics of good features (e.g., sparsity).
    • Usually train 1 layer of features at a time.
  – Then, e.g., train supervised classifier on top.

AKA “Self-taught learning” [Raina et al., ICML 2007]
Greedy layer-wise training

• Train representations with unlabeled data.
  – Start by training bottom layer alone.
Greedy layer-wise training

• Train representations with unlabeled data.
  – When complete, train a new layer on top using inputs from below as a new training set.

Forward pass only.
UFL Example

• Simple priors for good features:
  – Reconstruction: recreate input from features.
    \[ \mathcal{L}_{\text{recons}}(W_2, W_1) = \sum_{i} \|W_2 h(W_1 x^{(i)}) - x^{(i)}\|^2_2 \]
  – Sparsity: explain the input with as few features as possible.
    \[ \mathcal{L}_{\text{sparse}}(W_1) = \sum_{i} \|h(W_1 x^{(i)})\|_1 \]
Sparse auto-encoder

- Train two-layer neural network by minimizing:

\[
\minimize_{W_1, W_2} \sum_i \|W_2 h(W_1 x^{(i)}) - x^{(i)} \|_2^2 + \lambda \|h(W_1 x^{(i)})\|_1
\]

\[
h(z) = \text{ReLu}(z)
\]

- Remove “decoder” and use learned features (h).

[Ranzato et al., NIPS 2006]
Sparse auto-encoder

• Train two-layer neural network by minimizing:

\[
\minimize_{W_1, W_2} \sum_i \|W_2 h(W_1 x^{(i)}) - x^{(i)}\|^2_2 + \lambda \|h(W_1 x^{(i)})\|_1
\]

\[h(z) = \text{ReLU}(z)\]

• Remove “decoder” and use learned features (h).

[Ranzato et al., NIPS 2006]
What features are learned?

• Applied to image patches, well-known result:

- Sparse auto-encoder
  [Ranzato et al., 2007]

- Sparse coding
  [Olshausen & Field, 1996]

- Sparse RBM
  [Lee et al., 2007]
Pre-processing

• Unsupervised algorithms more sensitive to pre-processing.
  – Whiten your data. E.g., ZCA whitening:
    \[ [V, D] := \text{eig}(\text{cov}(X)) \]
    \[ \hat{x}^{(i)} := V (D + I \epsilon)^{-1/2} V^\top (x^{(i)} - \text{mean}(X)) \]
  – Contrast normalization often useful.
    \[ \hat{x} = \frac{x}{\sqrt{W x^2 + \epsilon}} \]
  – Do these before unsupervised learning at each layer.

[See, e.g., Coates et al., AISTATS 2011; Code at www.stanford.edu/~acoates/]
Group-sparsity

• Simple priors for good features:
  – Group-sparsity:

\[
\mathcal{L}_{\text{group-sparse}}(W_1) = \sum_i \sqrt{V[h(W_1x^{(i)})^2]}
\]

– V chosen to have a “neighborhood” structure. Typically in 2D grid.

\[
V_{ij} = \begin{cases} 
1 & \text{if neuron } i \text{ adjacent to } j. \\
0 & \text{otherwise}
\end{cases}
\]

Hyvärinen et al., Neural Comp. 2001
Ranzato et al., NIPS 2006
Kavukcuoglu et al., CVPR 2009
Garrigues & Olshausen, NIPS 2010
What features are learned?

• Applied to image patches:
  – Pool over adjacent neurons to create invariant features.
  – These are *learned* invariances without video.

Predictive Sparse Decomposition
[Kavukcuoglu et al., CVPR 2009]

Works with auto-encoders too.
[See, e.g., Le et al. NIPS 2011]
High-level features?

• Quite difficult to learn 2 or 3 levels of features that perform better than 1 level on supervised tasks.
  – Increasingly abstract features, but unclear how much abstraction to allow or what information to leave out.
Unsupervised Pre-training

• Use as initialization for supervised learning!
  – Features may not be perfect for task, but probably a good starting point.
  – AKA “supervised fine-tuning”.

• Procedure:
  – Train each layer of features greedily unsupervised.
  – Add supervised classifier on top.
  – Optimize entire network with back-propagation.

➤ Major impetus for renewed interest in deep learning.
[Hinton et al., Neural Comp. 2006]
[Bengio et al., NIPS 2006]
Unsupervised Pre-training

- Pre-training not always useful --- but sometimes gives better results than random initialization.

Results from [Le et al., ICML 2011]:

<table>
<thead>
<tr>
<th>Image-Net Version</th>
<th>9M images, 10K classes</th>
<th>14M images, 22K classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without pre-training</td>
<td>16.1%</td>
<td>13.6%</td>
</tr>
<tr>
<td>With pre-training</td>
<td><strong>19.2%</strong></td>
<td><strong>15.8%</strong></td>
</tr>
</tbody>
</table>

Notes: exact classification (not top-5). Random guessing = 0.01%.

See also [Erhan et al., JMLR 2010]
High-level features

• Recent work [Le et al., 2012; Coates et al., 2012] suggests high-level features can learn non-trivial concepts.
  – E.g., able to find single features that respond strongly to cats, faces:
Other Unsupervised Criteria

• Neural networks with other unsupervised training criteria.
  – Denoising, in-painting. [Vincent et al., 2008]
  – “Contraction” [Rifai et al., ICML 2011].
  – Temporal coherence [Zou et al., NIPS 2012]
    [Mobahi et al., ICML 2009]
RBMs

• Restricted Boltzmann Machine
  – Similar to auto-encoder, but probabilistic.
  – Bipartite, binary MRF.
  – Pretraining of RBMs used to initialize “deep belief network” [Hinton et al., 2006] and “deep boltzmann machine” [Salakhutdinov & Hinton, AISTATS 2009].

– Intractable
  • Gibbs sampling.
  • Train with contrastive divergence [Hinton, Neural Comp. 2002]
Sparse Coding

• Another class of models frequently used in UFL
  – Neuron responses are free variables.

\[
\text{minimize}_{W,h} \sum_i \|W h^{(i)} - x^{(i)}\|^2 + \lambda \|h^{(i)}\|_1
\]

[Olshausen & Field, 1996]

– Solve by alternating optimization over \(W\) and responses \(h\).

– Like sparse auto-encoder, but “encoder” to compute \(h\) is now a convex optimization algorithm.
  • Can replace encoder with a deep neural network. [Gregor & LeCun, ICML 2010]
  • Highly optimized implementations [Mairal, JMLR 2010]
Summary

• Supervised deep-learning
  – Practical and highly successful in practice. A general-purpose extension to existing ML.
  – Optimization, initialization, architecture matter!

• Unsupervised deep-learning
  – Pre-training often useful in practice.
  – Difficult to train many layers of features without labels.
  – Some evidence that useful high-level patterns are captured by top-level features.
Resources

*Tutorials*

Stanford Deep Learning tutorial:  
http://ufldl.stanford.edu/wiki

Deep Learning tutorials list:  
http://deeplearning.net/tutorials

IPAM DL/UFL Summer School:  
http://www.ipam.ucla.edu/programs/gss2012/

ICML 2012 Representation Learning Tutorial  
References


Overviews:

Yoshua Bengio,
“Practical Recommendations for Gradient-Based Training of Deep Architectures”

Yoshua Bengio & Yann LeCun,
“Scaling Learning Algorithms towards AI”

Yoshua Bengio, Aaron Courville & Pascal Vincent,
“Representation Learning: A Review and New Perspectives”

Software:

Theano GPU library: http://deeplearning.net/software/theano
SPAMS toolkit: http://spams-devel.gforge.inria.fr/