Deep Learning for Vision

Xiaoming Liu

Slides adapted from Adam Coates
What do we want ML to do?

- Given image, predict complex high-level patterns:
  - 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.
“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

• 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.

\[
\begin{pmatrix}
255 \\
98 \\
93 \\
87 \\
\vdots
\end{pmatrix}
\rightarrow
f(x)
\rightarrow
y = 1 \quad ("Cat")
\]

• 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 \([0,1]\).

  \[ \begin{array}{c}
  1 \\
  \end{array} \begin{array}{c}
  1 \\
  \end{array} \begin{array}{c}
  1 \\
  \end{array} \begin{array}{c}
  1 \\
  \end{array} \begin{array}{c}
  1 \\
  \end{array} \]

  – Find choice of \( \theta \) that minimizes objective:

  \[ \mathcal{L}(\theta) = - \sum_{i} \mathbb{1}\{y(i) = 1\} \log(f(x(i); \theta)) + \mathbb{P}(y(i) = 0|x(i)) \log(1 - f(x(i); \theta)) \]
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.

• 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?

Learning Algorithm

Is this a Coffee Mug?

+ 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.

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.
  – Let’s extend to case where features are given by tunable functions with their own parameters.

$$
\mathbb{P}(y = 1|x) = f(x; \theta, W) = \sigma(\theta^T \sigma(Wx))
$$

Outer part of function is same as logistic regression.

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

$$
\begin{bmatrix}
\sigma(w_1x) \\
\sigma(w_2x) \\
\vdots \\
\sigma(w_Kx)
\end{bmatrix}
$$
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](image.png)

Intermediate representation ("features")

Prediction for \( \mathbb{P}(y = 1|x) \)
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”.

\[ x \xrightarrow[\sigma(W_1x)]{} h \xrightarrow[\sigma(W_2h)]{} h' \xrightarrow[\sigma(\theta^T h')]{} f \]
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:

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

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

• 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:

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

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 $\delta \equiv \nabla_h \mathcal{L}$
  
  - 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>$\delta Z^\top$</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>$(W^\top \frac{\delta}{h}) \odot z$</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>
Back-propagation

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

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

\[ \nabla_x \mathcal{L} \xrightarrow{J_h} \nabla_h \mathcal{L} \xrightarrow{J_f} \nabla_f \mathcal{L} \]

“Backward” modules for each forward stage.
Example

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

\[
x \rightarrow \sigma(Wx) \rightarrow h \rightarrow \sigma(\theta^\top h) \rightarrow f
\]

\[
\nabla_W \mathcal{L} \quad \rightarrow \quad J_{h,W} \quad \rightarrow \quad \nabla_h \mathcal{L} \quad \rightarrow \quad J_{f,h} \quad \rightarrow \quad \nabla_f \mathcal{L}
\]

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.

• 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.
  – Has to be random otherwise every neuron will be equal!
  – 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]
Optimization: Step sizes

• Choose SGD step size carefully.
  – Up to factor ~2 can make a difference.

• Strategies:
  – Brute-force: try many; pick one with best result.
  – Racing: pick size with best error on validation data after T steps.
    • Not always accurate if T is too small.

• Step size schedule:
  – Fixed: same step size
  – Step
  – Multi-Step
  – Inverse

Optimization: Momentum

• “Smooth” estimate of gradient from several steps of SGD:
  \[ v := \mu v + \epsilon_t \nabla_{\theta} \mathcal{L}(\theta) \]
  \[ \theta := \theta + v \]

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

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]
Application

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 train.
- 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”.
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 = 3 to 20
    • Typical step size = 1 pixel up to RF size.
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”.)
Local connectivity

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

1-dimensional example:
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]
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.

\[ h = (W z^a)^{1/a} \]

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

- Reduce covariance shift during training
- Spatially over all samples in each batch

\[
\mu_B \leftarrow \frac{1}{W \times H \times m} \sum_{w,h,i} x_{w,h,i}
\]

\[
\sigma^2_B \leftarrow \frac{1}{W \times H \times m} \sum_{w,h,i} (x_{w,h,i} - \mu_B)^2
\]

\[
\hat{x}_{w,h,i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \epsilon}}
\]

\[
y_i \leftarrow \gamma \hat{x}_{w,h,i} + \beta \equiv BN_{\gamma,\beta}(x_{w,h,i})
\]
**Application: Image-Net**

- System from Krizhevsky et al., NIPS 2012:
  - Convolutional neural network.
  - Max-pooling.
  - Rectified linear units (ReLu).
  - Local response normalization.
  - Local connectivity.
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
Go
AlphaGo beat best human (Lee Sedol) 3/2016
(win-win-win-loss-win)
Games are available at
https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol
How does AlphaGo work?
(For all the details, see their paper: www.nature.com/nature/journal/v529/n7587/pdf/nature16961.pdf)

• Combines two techniques:
  – Monte Carlo Tree Search (MCTS)
    • Historical advantages of MCTS over minimax-search approaches:
      – Does not require an evaluation function
      – Typically works better with large branching factors
    • Improved Go programs by ~10 kyu around 2006
  – Deep Learning
    • ‘Value networks’ to evaluate board positions and
    • ‘Policy networks’ to select moves
AlphaGo hardware

- Evaluating policy and value networks requires several orders of magnitude more computation than traditional search heuristics
- AlphaGo uses an asynchronous multi-threaded search that executes simulations on CPUs, and computes policy and value networks in parallel on GPUs
- Final version of AlphaGo used 40 search threads, 48 CPUs, and 8 GPUs
- (They also implemented a distributed version)
Speech Recognition


Input: acoustic features (spectrograms)

Output: characters (and space) + null

No phoneme and lexicon (No OOV problem)
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/