Introduction

A brief history of computers

<table>
<thead>
<tr>
<th></th>
<th>1970s</th>
<th>1980s</th>
<th>1990s</th>
<th>2000s</th>
<th>2010s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^5$</td>
<td>$10^8$</td>
<td>$10^{11}$</td>
</tr>
<tr>
<td>RAM</td>
<td>?</td>
<td>1MB</td>
<td>100MB</td>
<td>10GB</td>
<td>1TB</td>
</tr>
<tr>
<td>CPU</td>
<td>?</td>
<td>10MF</td>
<td>1GF</td>
<td>100GF</td>
<td>1PF GPU</td>
</tr>
</tbody>
</table>

- Data grows at higher exponent
- Moore’s law (silicon) vs. Kryder’s law (disks)
- Early algorithms data bound, now CPU/RAM bound

- Introduction to deep learning
- Deep Convolutional Networks
- Tips for optimizing deep networks

Slides from Alex Smola, Dhruv Batra, Yisong Yue, Geoffrey Hinton, Raquel Urtasun
Perceptron

\[ f(x) = \sum_i w_i x_i = \langle w, x \rangle \]

Not linearly separable data

- Some datasets are not **linearly separable**!
  - e.g. XOR problem
- Nonlinear separation is trivial

Addressing non-linearly separable data

- Two options:
  - Option 1: Non-linear functions
  - Option 2: Non-linear classifiers

Option 1 — Non-linear features

- Choose non-linear features, e.g.,
  - Typical linear features: \( w_0 + \sum_i w_i x_i \)
  - Example of non-linear features:
    - Degree 2 polynomials, \( w_0 + \sum_i w_i x_i + \sum_{ij} w_{ij} x_i x_j \)
- Classifier \( h_w(x) \) still linear in parameters \( w \)
  - As easy to learn
  - Data is linearly separable in higher dimensional spaces
  - Express via kernels
Option 2 — Non-linear classifiers

- Choose a classifier $h_w(x)$ that is non-linear in parameters $w$, e.g.,
  - Decision trees, neural networks,...

- More general than linear classifiers
- But, can often be harder to learn (non-convex optimization required)
- Often very useful (outperforms linear classifiers)
- In a way, both ideas are related

Recall: The Neuron Metaphor

- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node

Biological Neurons

- Soma (CPU)
  Cell body - combines signals
- Dendrite (input bus)
  Combines the inputs from several other nerve cells
- Synapse (interface)
  Interface and parameter store between neurons
- Axon (cable)
  May be up to 1m long and will transport the activation signal to neurons at different locations

Types of Neuron

- Potentially more. Requires a convex loss function for gradient descent training.

Linear Neuron
$$z = \theta_0 + \sum_i x_i \theta_i$$
$$y = \frac{1}{1 + e^{-z}}$$

Logistic Neuron
$$z = \theta_0 + \sum_i x_i \theta_i$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron
$$z = \theta_0 + \sum_i x_i \theta_i$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$
Limitation

- A single “neuron” is still a linear decision boundary
- What to do?
- Idea: Stack a bunch of them together!

Nonlinearities via Layers

- Cascade Neurons together
- The output from one layer is the input to the next
- Each Layer has its own sets of weights

Universal Function Approximators

- Theorem
  - 3-layer network with linear outputs can uniformly approximate any continuous function to arbitrary accuracy, given enough hidden units [Funahashi ’89]
A simple example

- Consider a neural network with two layers of neurons.
  - Neurons in the top layer represent known shapes.
  - Neurons in the bottom layer represent pixel intensities.
- A pixel gets to vote if it has ink on it.
  - Each inked pixel can vote for several different shapes.
- The shape that gets the most votes wins.

How to display the weights

Give each output unit its own “map” of the input image and display the weight coming from each pixel in the location of that pixel in the map.
Use a black or white blob with the area representing the magnitude of the weight and the color representing the sign.

How to learn the weights

Show the network an image and increment the weights from active pixels to the correct class.
Then decrement the weights from active pixels to whatever class the network guesses.
The learned weights

The details of the learning algorithm will be explained later.

Why insufficient

• A two layer network with a single winner in the top layer is equivalent to having a rigid template for each shape.
  - The winner is the template that has the biggest overlap with the ink.

• The ways in which hand-written digits vary are much too complicated to be captured by simple template matches of whole shapes.
  - To capture all the allowable variations of a digit we need to learn the features that it is composed of.

Multilayer Perceptron

• Layer Representation
  \[ y_i = W_i x_i \]
  \[ x_{i+1} = \sigma(y_i) \]

• (typically) iterate between linear mapping Wx and nonlinear function

• Loss function \( l(y, y_i) \) to measure quality of estimate so far

Backpropagation

• Layer Representation
  \[ y_i = W_i x_i \]
  \[ x_{i+1} = \sigma(y_i) \]

• Compute change in objective
  \[ g_j = \partial_{W_j} l(y, y_i) \]

• Chain rule
  \[ \partial_x [f_2 \circ f_1] (x) = [\partial_{f_1} f_2 \circ f_1 (x)] [\partial_x f_1] (x) \]
Backpropagation

- Layer Representation
  \[ y_i = W_i x_i \]
  \[ x_{i+1} = \sigma(y_i) \]

- Gradients
  \[ \frac{\partial z_i}{\partial y_i} = W_i \]
  \[ \frac{\partial W_i}{\partial y_i} = x_i \]
  \[ \frac{\partial y_i}{\partial x_{i+1}} = \sigma'(y_i) \]

  \[ \Rightarrow \frac{\partial z_i}{\partial x_{i+1}} = \sigma'(y_i) W_i^T \]

- Backprop
  \[ g_n = \frac{\partial l}{\partial y_n} \]
  \[ g_i = \frac{\partial l}{\partial y_n} = g_{i+1} \frac{\partial x_i}{\partial x_{i+1}} \]
  \[ \frac{\partial w_i}{\partial y_n} = g_{i+1} \sigma'(y_i) x_i^T \]

Example application

- Consider trying to classify image of handwritten digit: 32x32 pixels
- Single output units – it is a 4 (one vs. all)?
- Use the sigmoid output function:
  \[ o_i = \frac{1}{1 + \exp(-z_k)} \]
  \[ z_k = (w_{k0} + \sum_{j=1}^{J} h_j(x)v_{kj}) \]

  - Can train the network, that is, adjust all the parameters \( w \), to optimize the training objective, but this is a complicated function of the parameters

The idea behind backpropagation

- We don’t know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.

- We can compute error derivatives for all the hidden units efficiently at the same time.
  - Once we have the error derivatives for the hidden activities, it’s easy to get the error derivatives for the weights going into a hidden unit.
Non-linear neurons with smooth derivatives

- For backpropagation, we need neurons (units) that have well-behaved derivatives
  - Typically they use the logistic function
  - The output is a smooth function of the inputs and the weights.

\[
x_j = b_j + \sum_i y_i w_{ij}
\]

\[
y_j = \frac{1}{1 + e^{-x_j}}
\]

\[
\frac{\partial x_j}{\partial w_{ij}} = y_i \quad \frac{\partial x_j}{\partial y_j} = w_{ij}
\]

\[
\frac{dy_j}{dx_j} = y_j (1 - y_j)
\]

Back-propagation: Sketch on one training case

1. Convert the discrepancy between each output and its target value into an error derivative.

\[
E = \frac{1}{2} \sum_k (o_k - t_k)^2
\]

\[
\frac{\partial E}{\partial o_k} = o_k - t_k
\]

2. Compute error derivatives in each hidden layer from error derivatives in the layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]

3. Use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights.

The derivatives

\[
\frac{\partial E}{\partial x_j} = \frac{dy_j}{dx_j} \frac{\partial E}{\partial y_j} = y_j (1 - y_j) \frac{\partial E}{\partial y_j}
\]

\[
\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}
\]

\[
\frac{\partial E}{\partial y_i} = \sum_j \frac{dx_j}{dy_i} \frac{\partial E}{\partial x_j} = \sum j w_{ij} \frac{\partial E}{\partial x_j}
\]

Optimization

- Layer Representation
  \[y_i = W_i x_i\]
  \[x_{i+1} = \sigma(y_i)\]

- Gradient descent
  \[W_i \leftarrow W_i - \eta \partial W_i l(y, y_n)\]

- Second order method (use higher derivatives)

- Stochastic gradient descent (use only one sample)

- Minibatch (small subset)
Layers

### Fully Connected

- Forward mapping
  \[ y_i = W_i x_i \]
  \[ x_{i+1} = \sigma(y_i) \]
  with subsequent nonlinearity
- Backprop gradients
  \[ \partial_x, x_{i+1} = \sigma'(y_i) W_i^T \]
  \[ \partial_W, x_{i+1} = \sigma'(y_i) x_i^T \]
- General purpose layer

### Rectified Linear Unit (ReLU)

- Forward mapping
  \[ y_i = W_i x_i \]
  \[ x_{i+1} = \sigma(y_i) \]
  with subsequent nonlinearity
- Gradients vanish at tails
- Solution - replace by \[\max(0, x)\]
  - Derivative is in \[\{0; 1\}\]
  - Sparsity of signal
  (Nair & Hinton, machinelearning.wustl.edu/mlpapers/paper_files/icml2010_NairH10.pdf)

Where is Wally
Convolutions

- Images typically have invariant patterns
  - E.g., directional gradients are translational invariant:

- Apply convolution to local sliding windows

http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

Convolution Filters

- Applies to an image patch \( x \)
  - Converts local window into single value
  - Slide across image

\[
x \otimes W = \sum_{ij} W_{ij} x_{ij}
\]

Local Image Patch

\[
\otimes W = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}
\]

Gabor Filters

- Most common low-level convolutions for computer vision

Example \( W \):

http://en.wikipedia.org/wiki/Gabor_filter
Gaussian Blur Filters

- Weights decay according to Gaussian Distribution
  - Variance term controls radius

Example W:
Apply per RGB Channel

- Black = 0
- White = Positive

http://en.wikipedia.org/wiki/Gaussian_blur

Convolutional Layers

- Images have translation invariance (to some extent)
- Low level is mostly edge and feature detectors
- Usually via convolution (plus nonlinearity)

Subsampling & MaxPooling

- Multiple convolutions blow up dimensionality
- Subsampling - average over patches (this works decently)
- MaxPooling - pick the maximum over patches (often non overlapping ones)
Max Pooling

- Assume Convolution Layer is eye detector
- How to make detector more robust to the exact location of the eye?

```
```

Alternative: L2 Pooling

- L2 norm of a neighborhood of convolutional layer outputs
- Softer version of max pooling
- Harder to differentiate

```
\[ \sqrt{\sum_{i,j} f_{ij}^2} \]
```

```
```

Max Pooling

- Maximum response from a neighborhood of convolutional layer outputs
- i.e., an OR gate!

```
```

Depth vs. Width

- Longer range effects
- many narrow convolutions
- few wide convolutions
- More nonlinearities work better (same number of parameters)

```
Simonyan and Zisserman
arxiv.org/pdf/1409.1556v6.pdf
```

```
```
Fancy structures

- Compute different filters
- Compose one big vector from all of them
- Layer this iteratively

Szegedy et al.
arxiv.org/pdf/1409.4842v1.pdf

Whole system training

- Whole system training
- Layers need not be ‘neural networks’
- Rankers
- Segmenters
- Finite state automata
- Jointly train a full OCR system

Le Cun, Bottou, Bengio, Haffner, 2001
yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

Online Demo

- http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html
ImageNET

- Object recognition competition (2012)
  - 1.5 Million Labeled Training Examples
  - ≈1000 classes

Deep Convolutional Net for ImageNET

- 7 Hidden Layers
  - 5 Convolutional
  - 2 Regular

- Multiclass Logistic Regression at top

- Trained using stochastic gradient descent
  - And a lot of tricks

- Won the 2012 ImageNET competition

http://www.image-net.org/challenges/LSVRC/2012/results.html

Visualizing CNN (Layer 1)

Figure Credit: [Zeiler & Fergus ECCV14]
Objectives

- Stack hidden layer activations as new feature representation
- Train an SVM =)
- Generalize to other datasets

Use Hidden Layers as Features

Failure Cases

Classification

- Binary classification
  \[ \log(1 + \exp(-yy_n)) \]
  Binary exponential model

- Multiclass classification (softmax)
  Multinomial exponential model
  \[ -\log p(y|y_n) = -\log \frac{e^{y_n[y]}}{\sum_{y'} e^{y_n[y']}} = \log \sum_{y'} e^{y_n[y']} - y_n[y] \]

- Pretty much anything else we did so far in BBM 406

Use Hidden Layers as Features

Input Image
RGB Input Image 224 x 224 x 3

7x7x3 Convolution 3x3 Max Pooling Down Sample 4x 55 x 55 x 96
5x5x96 Convolution 3x3 Max Pooling Down Sample 4x 13 x 13 x 256
3x3x256 Convolution 13 x 13 x 354

Logistic Regression
4096 Units

Standard
4096 Units

3x3x354 Convolution 3x3 Max Pooling Down Sample 2x 6 x 6 x 256
3x3x354 Convolution 13 x 13 x 354

Correctly
Incorrectly
Predicts
Predicts

Figure 5: Adversarial examples generated for AlexNet [49].

Figure 6: Adversarial examples for QuocNet [60].

Regression

- Least mean squares
  \[ \frac{1}{2} \| y - y_n \|^2 \]
  this works for vectors, too

- Applications
  - Stock market prediction (more on this later)
  - Image superresolution
    (regress from lower dimensional to higher dimensional image)
  - Recommendation and rating (Netflix)

Autoencoder

- Regress from observation to itself \( (y_n = x_1) \)
- Lower-dimensional layer is bottleneck
- Often trained iteratively
- Extracts approximate sufficient statistic of data
- Special case - PCA
  - linear mapping
  - only single layer

Optimization
Stochastic Gradient Descent

- Update parameters according to
  \[ W_{ij} \leftarrow W_{ij} - \eta_{ij}(t)g_{ij} \]
- Rate of decay
- Adjust each layer
- Adjust each parameter individually
- Minibatch size
- Momentum terms
- Lots of things that can (should) be adjusted (via Bayesian optimization, e.g. Spearmint, MOE)

Learning rate decay

- **Constant**
  (requires schedule for piecewise constant, tricky)
- **Polynomial decay**
  \[ \eta(t) = \frac{\alpha}{(\beta + t)^{\gamma}} \]
  Recall exponent of 0.5 for conventional SGD, 1 for strong convexity. Bottou picks 0.75
- **Exponential decay**
  \[ \eta(t) = \alpha e^{-\beta t} \]
  risky since decay could be too aggressive

Minibatch

- Update parameters according to
- Aggregate gradients before applying
- Reduces variance in gradients
- Better for vectorization (GPUs)
  vector, vector < vector, matrix < matrix, matrix
- Large minibatch may need large memory (and slow updates).
- Magic numbers are 64 to 256 on GPUs

AdaGrad

- Adaptive learning rate (preconditioner)
  \[ \eta_{ij}(t) = \frac{\eta_t}{\sqrt{K + \sum_t g_{ij}^2(t)}} \]
  \( t \)
- For directions with large gradient, decrease learning rate aggressively to avoid instability
- If gradients start vanishing, learning rate decrease reduces, too
- Local variant
  \[ \eta_{ij}(t) = \frac{\eta_t}{\sqrt{K + \sum_{t'=t-\tau}^t g_{ij}^2(t')}} \]

Duchi, Hazan, Singer, 2010
http://www.magicbroom.info/Papers/DuchiHaSi10.pdf
Momentum

- Average over recent gradients
- Helps with local minima
- Flat (noisy) gradients
  \[ m_t = (1 - \lambda)m_{t-1} + \lambda g_t \]
  \[ w_t \leftarrow w_t - \eta_t g_t - \tilde{\eta}tm_t \]
- Can lead to oscillations for large momentum
- Nesterov’s accelerated gradient
  \[ m_{t+1} = \mu m_t + \epsilon g(w_t - \mu m_t) \]
  \[ w_{t+1} = w_t - m_{t+1} \]

Capacity control

- Minimizing loss can lead to overfitting
- Weight decay
  \[ w_t \leftarrow w_t - \eta_t g_t \]
  \[ w_t \leftarrow (1 - \lambda)w_t - \eta_t g_t \]
- Parameter clipping
- Overheated GPU
- Numerical instabilities

Dropout

- Avoid parameter sensitivity
  (small changes in value shouldn’t change result)
- Distributed representation
  (information carried by more than 1 dimension)
- Randomized sparsification
  \[ y_{ti} = \xi_{ti}y_{ti} \text{ where } \begin{cases} P(X_{ti} = \pi^{-1}) = \pi \\ P(X_{ti} = 0) = 1 - \pi \end{cases} \]
- Same trick works for matrix W, too: DropConnect slightly better performance ...
  \[ \text{http://cs.nyu.edu/~wanli/dropc/} \]

Srivastava, Hinton, Krizhevski, Sutskever, Salakhutdinov

\[ \text{http://jmlr.org/papers/v15/srivastava14a.html} \]

Dropout & DropConnect

Regular

Dropout

DropConnect

\[ \text{http://cs.nyu.edu/~wanli/dropc/} \]
Toolkits

  Efficient for convolutional models / images
- Torch [http://torch.ch/](http://torch.ch/)
  Very efficient. But you must LIKE Lua ...
  Google and Facebook love it
- Theano [http://deeplearning.net/software/theano/](http://deeplearning.net/software/theano/)
  Compiled from Python. Not as efficient as Torch
- Minerva [https://github.com/dmlc/minerva](https://github.com/dmlc/minerva)
  Compiler layout of execution on machines
- CXXNet [https://github.com/dmlc/cxxnet](https://github.com/dmlc/cxxnet)
  Simpler than Caffe. More efficient
- Parameter Server bindings to [https://github.com/dmlc/Minerva, Caffe, CXXNet, ...](https://github.com/dmlc/Minerva, Caffe, CXXNet, ...)

![Graph showing comparison between With and Without dropout](image)