Chapter 4 (Part 2): Artificial Neural Networks

CS 536: Machine Learning
Littman (Wu, TA)

Administration

iCML–03: instructional Conference on Machine Learning
http://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/

Weka exercise
http://www.cs.rutgers.edu/~mlittman/courses/ml03/hw1.pdf
Grading Components

- HWs (not handed in): 0%
- Project paper (iCML): 25%
  - document (15%), review (5%), revision(5%)
- Project presentation: 20%
  - oral (10%), slides (10%)
- Midterm (take home): 20%
- Final: 35%

Artificial Neural Networks

[Read Ch. 4]
[Review exercises 4.1, 4.2, 4.5, 4.9, 4.11]
- Threshold units [ok]
- Gradient descent [today]
- Multilayer networks [today]
- Backpropagation [today?]
- Hidden layer representations
- Example: Face Recognition
- Advanced topics
Gradient Descent

To understand, consider simpler *linear unit*, where
\[ o = w_0 + w_1 x_1 + \ldots + w_n x_n \]

Let's learn \( w_i \)'s to minimize squared error

\[ E[w] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \]

Where \( D \) is set of training examples
Gradient Descent

Gradient
\[ E [w] = [\partial E/\partial w_0, \partial E/\partial w_1, \ldots, \partial E/\partial w_n] \]

Training rule:
\[ w = -\nabla E [w] \]
in other words:
\[ w_i = -\partial E/\partial w_i \]

Gradient of Error
\[ \partial E/\partial w_i \]
\[ = \partial / \partial w_i \ 1/2 \ \partial_d \ (t_d - o_d)^2 \]
\[ = 1/2 \ \partial_d \ \partial / \partial w_i (t_d - o_d)^2 \]
\[ = 1/2 \ \partial_d \ 2 \ (t_d - o_d) \ \partial / \partial w_i (t_d - o_d) \]
\[ = \partial_d \ (t_d - o_d) \ \partial / \partial w_i (t_d - w \ x_d) \]
\[ = \partial_d \ (t_d - o_d) \ (-x_{i,d}) \]
Gradient Descent Code

\[ \text{GRADIENT-DESCENT}(\text{training examples}, \mathbf{w}) \]

*Each training example is a pair of the form \( \langle x, t \rangle \), where \( x \) is the vector of input values, and \( t \) is the target output value. \( \mathbf{w} \) is the learning rate (e.g., .05).*

- Initialize each \( w_i \) to some small random value
- Until the termination condition is met, Do
  - Initialize each \( Dw_i \) to zero.
  - For each \( \langle x, t \rangle \) in training examples, Do
    - Input the instance \( x \) to the unit and compute the output \( o \)
    - For each linear unit weight \( w_i \), Do
      \[ Dw_i \leftarrow Dw_i + h(t-o)x_i \]
  - For each linear unit weight \( w_i \), Do
    \[ w_i \leftarrow w_i + Dw_i \]

Summary

Perceptron training rule will succeed if
- Training examples are linearly separable
- Sufficiently small learning rate

Linear unit training uses gradient descent
- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate
- Even when training data contains noise
- Even when training data not \( H \) separable
Stochastic Gradient Descent

**Batch mode** Gradient Descent:

Do until satisfied

1. Compute the gradient $\nabla E_D[w]$
2. $w \leftarrow w - \Delta E_D[w]$

**Incremental mode** Gradient Descent:

Do until satisfied

- For each training example $d$ in $D$
  1. Compute the gradient $\nabla E_d[w]$
  2. $w \leftarrow w - \Delta E_d[w]$

More Stochastic Grad. Desc.

$E_D[w] \equiv 1/2 \sum_{d \in D} (t_d - o_d)^2$

$E_d[w] \equiv 1/2 (t_d - o_d)^2$

*Incremental Gradient Descent* can approximate *Batch Gradient Descent* arbitrarily closely if $\Delta$ set small enough.
Multilayer Networks

Decision Boundaries
Sigmoid Unit

\[ s(x) = \frac{1}{1 + e^{-x}} \]

Derivatives of Sigmoids

Nice property:
\[ \frac{d}{dx} s(x) = s(x) (1 - s(x)) \]

We can derive gradient decent rules to train

- One sigmoid unit
- Multilayer networks of sigmoid units \( \Rightarrow \) Backpropagation
Error Gradient for Sigmoid

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} 1/2 \cdot \sigma_d (t_d - o_d)^2
\]

\[
= 1/2 \cdot \sigma_d \frac{\partial}{\partial w_i} (t_d - o_d)^2
\]

\[
= 1/2 \cdot \sigma_d 2 (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)
\]

\[
= \sigma_d (t_d - o_d) (-\frac{\partial o_d}{\partial w_i})
\]

\[
= - \sigma_d (t_d - o_d) (\frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i})
\]

Even more...

But we know:

\[
\frac{\partial o_d}{\partial \text{net}_d} = \sigma_d \frac{\partial \text{net}_d}{\partial \text{net}_d} = o_d (1- o_d)
\]

\[
\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}
\]

So:

\[
\frac{\partial E}{\partial w_i} = - \sigma_d (t_d - o_d) o_d (1- o_d) x_{i,d}
\]
Backpropagation Algorithm

Initialize all weights to small random numbers.
Until satisfied, Do
• For each training example, Do
  1. Input the training example to the network and compute the network outputs
  2. For each output unit $k$
     \[ d_k = o_k(1 - o_k)(t_k - o_k) \]
  3. For each hidden unit $h$
     \[ d_h = o_h(1 - o_h) \sum_{k \text{ in outputs}} w_{h,k} \]
  4. Update each network weight $w_{ij}$
     \[ w_{ij} \leftarrow w_{ij} + \eta \delta_i x_{ij} \]

More on Backpropagation

• Gradient descent over entire network weight vector
• Easily generalized to arbitrary directed graphs
• Will find a local, not necessarily global error minimum
  – In practice, often works well (can run multiple times)
More more

• Often include weight *momentum* 
  \[ \Delta w_{ij}(n) = \Delta_j x_{i,j} + \Delta w_{ij}(n-1) \]

• Minimizes error over training examples
  – Will it generalize well to subsequent examples?

• Training can take thousands of iterations ❗ slow!

• Using network after training is very fast

Hidden Layer Reps

<table>
<thead>
<tr>
<th>Simple target function:</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td></td>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td></td>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td></td>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td></td>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td></td>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td></td>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Autoencoder

Can the mapping be learned with this network??

Hidden Layer Rep.

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>.89 .04 .08</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>.01 .11 .88</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>.01 .97 .27</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>.99 .97 .71</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>.03 .05 .02</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>.22 .99 .99</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>.80 .01 .98</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>.60 .94 .01</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Training

Convergence of Backprop

Gradient descent to some local minimum

• Perhaps not global minimum...
• Add momentum
• Stochastic gradient descent
• Train multiple nets with different initial weights
More on Convergence

Nature of convergence
• Initialize weights near zero
• Therefore, initial networks are near-linear
• Increasingly non-linear functions possible as training progresses

Expressiveness of ANNs

Boolean functions:
• Every Boolean function can be represented by network with single hidden layer
• but might require exponential (in number of inputs) hidden units
Real-valued Functions

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

Overfitting in ANNs (Ex. 1)
Overfitting in ANNs (Ex. 2)

When do you stop training?

Face Recognition

left  strt  right  up

30x32 inputs
Typical input images

- 90% accurate learning head pose, and recognizing 1–of–20 faces

Learned Weights

http://www.cs.cmu.edu/tom/faces.html

Bias first?
Alternative Error Functions

Penalize large weights:
\[
E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \sum_{i,j} w_{ji}^2
\]

Train on target slopes as well as values:
\[
E(\tilde{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} \left( (t_{kd} - o_{kd})^2 + \mu \sum_{j \in \text{inputs}} \left( \frac{\partial t_{kd}}{\partial x_d^j} \frac{\partial x_d^j}{\partial x_d^j} \right)^2 \right)
\]

Tie together weights:
- e.g., in phoneme recognition network

Recurrent Networks

(a) Feedforward network
(b) Recurrent network
Unfolding: BPTT