Machine Learning 10-701

Tom M. Mitchell
Machine Learning Department
Carnegie Mellon University

March 24, 2011

Today:
• Non-linear regression
• Artificial neural networks
• Backpropagation
• Cognitive modeling
• Deep belief networks

Reading:
• Mitchell: Chapter 4
• Bishop: Chapter 5

Artificial Neural Networks to learn $f: X \rightarrow Y$

• $f$ might be non-linear function
• $X$ (vector of) continuous and/or discrete vars
• $Y$ (vector of) continuous and/or discrete vars

• Represent $f$ by network of logistic units
• Each unit is a logistic function
  
  \[
  \text{unit output} = \frac{1}{1 + \exp(w_0 + \sum_i w_i x_i)}
  \]

• MLE: train weights of all units to minimize sum of squared errors of predicted network outputs
• MAP: train to minimize sum of squared errors plus weight magnitudes
Multilayer Networks of Sigmoid Units
Connectionist Models

Consider humans:
- Neuron switching time $\sim .001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{1-5}$
- Scene recognition time $\sim .1$ second
- 100 inference steps doesn’t seem like enough
  $\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN’s):
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

Sigmoid Unit

$$\sigma(x) \text{ is the sigmoid function}$$

$$\frac{1}{1 + e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

We can derive gradient decent rules to train
- One sigmoid unit
- Multilayer networks of sigmoid units $\rightarrow$ Backpropagation
M(C)LE Training for Neural Networks

- Consider regression problem \( f: \mathbf{X} \rightarrow \mathbf{Y} \), for scalar \( Y \)
  \( y = f(x) + \varepsilon \)
  - assume noise \( N(0, \sigma) \), iid

- Let’s maximize the conditional data likelihood
  \[
  W \leftarrow \arg \max_W \ln \prod_l P(Y_l | X_l, W) 
  \]
  \[
  W \leftarrow \arg \min_W \sum_l (y_l - \bar{f}(x_l))^2 
  \]

MAP Training for Neural Networks

- Consider regression problem \( f: \mathbf{X} \rightarrow \mathbf{Y} \), for scalar \( Y \)
  \( y = f(x) + \varepsilon \)

- Gaussian \( P(W) = N(0, \sigma I) \)

- Let’s maximize the conditional data likelihood
  \[
  W \leftarrow \arg \max_W \ln P(W) \prod_l P(Y_l | X_l, W) 
  \]
  \[
  W \leftarrow \arg \min_W \left[ c \sum_i w_i^2 \right] + \left[ \sum_l (y_l - \bar{f}(x_l))^2 \right] 
  \]
  \[
  \ln P(W) \leftrightarrow c \sum_i w_i^2 
  \]
Error Gradient for a Sigmoid Unit

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
= \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
= \sum_{d \in D} (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right) \\
= -\sum_{d \in D} (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}
\]

But we know:

\[
\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d) \\
\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = \vec{x}_i,d
\]

So:

\[
\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d(1 - o_d) \vec{x}_i,d
\]

---

Gradient Descent

\[
\nabla E[\vec{w}] = \begin{bmatrix} \frac{\partial E}{\partial w_0} & \frac{\partial E}{\partial w_1} & \cdots & \frac{\partial E}{\partial w_n} \end{bmatrix}
\]

Training rule:

\[
\Delta \vec{w} = -\eta \nabla E[\vec{w}]
\]

i.e.,

\[
\Delta w_i = -\eta \frac{\partial E}{\partial w_i}
\]
Incremental (Stochastic) Gradient Descent

**Batch mode** Gradient Descent:
Do until satisfied
1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

**Incremental mode** Gradient Descent:
Do until satisfied
- For each training example $d$ in $D$
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if $\eta$ made small enough.

---

Backpropagation Algorithm (MLE)

- 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 $k$
      $$\delta_k = o_k(1 - o_k)(t_k - o_k)$$
    3. For each hidden unit $h$
      $$\delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$
    4. Update each network weight $w_{i,j}$
      $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
- where
  $$\Delta w_{i,j} = \eta \delta_j x_i$$
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)
- Often include weight momentum $\alpha$
  \[ \Delta w_{ij}(n) = \eta \frac{\partial \text{Error}}{\partial w_{ij}} + \alpha \Delta w_{ij}(n-1) \]
- Minimizes error over training examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\to$ slow!
- Using network after training is very fast

Overfitting in ANNs

[Graphs showing error versus weight updates for different examples]

\[ 1 + \exp(\theta x) \]
Dealing with Overfitting

Our learning algorithm involves a parameter

\[ n = \text{number of gradient descent iterations} \]

How do we choose \( n \) to optimize future error?

(note: similar issue for logistic regression, decision trees, …)

e.g. the \( n \) that minimizes error rate of neural net over future data

---

Dealing with Overfitting

Our learning algorithm involves a parameter

\[ n = \text{number of gradient descent iterations} \]

How do we choose \( n \) to optimize future error?

• Separate available data into training and validation set
• Use training to perform gradient descent
• \( n \leftarrow \text{number of iterations that optimizes validation set error} \)

→ gives unbiased estimate of optimal \( n \)
   (but a biased estimate of true error)
K-Fold Cross Validation

Idea: train multiple times, leaving out a disjoint subset of data each time for test. Average the test set accuracies.

Partition data into K disjoint subsets
For k=1 to K
  testData = kth subset
  h ← classifier trained* on all data except for testData
  accuracy(k) = accuracy of h on testData
end
FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps

Leave-One-Out Cross Validation

This is just k-fold cross validation leaving out one example each iteration

Partition data into K disjoint subsets, each containing one example
For k=1 to K
  testData = kth subset
  h ← classifier trained* on all data except for testData
  accuracy(k) = accuracy of h on testData
end
FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps
Expressive Capabilities 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

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].

Learning Hidden Layer Representations

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Can this be learned??
Learning Hidden Layer Representations

A network:

Learned hidden layer representation:

<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>.91 .11 .38</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

Sum of squared errors for each output unit
Neural Nets for Face Recognition

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

Learned Hidden Unit Weights

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