CMSC726 Spring 2006: Neural Networks

readings: Mitchell ch. 4
souces: course slides are based on material from a variety of sources, including Tom Dietterich, Carlos Guestrin, Terran Lane, Rich Maclin, Ray Mooney, Andrew Moore, Andrew Ng, Jude Shavlik, and others.

Artificial Neural Networks

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face recognition
- Advanced topics
| Connectionist Models |

Consider humans
- Neuron switching time $\sim 0.001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim .1$ second
- 100 inference step does not seem like enough

*must use lots of parallel computation!*

Properties of artificial neural nets (ANNs):
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

| When to Consider Neural Networks |

- Input is high-dimensional discrete or real-valued (e.g., raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is *unimportant*

Examples:
- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction
ALVINN drives 70 mph on highways

30x32 Sensor Input Retina

Perceptron

$$\sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
-1 & \text{otherwise}
\end{cases}$$

$$o(x_1, \ldots, x_n) = \begin{cases} 
1 & \text{if } w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\
-1 & \text{otherwise}
\end{cases}$$

Sometimes we will use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 
1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\
-1 & \text{otherwise}
\end{cases}$$
Decision Surface of Perceptron

Represents some useful functions
- What weights represent \( g(x_1, x_2) = \text{AND}(x_1, x_2) \)?
But some functions not representable
- e.g., not linearly separable
- therefore, we will want networks of these ...

Perceptron Training Rule

\[
\Delta w_i = \eta (t - o)x_i
\]
where

\[ w_i \leftarrow w_i + \Delta w_i \]

- \( t = c(\bar{x}) \) is target value
- \( o \) is perceptron output
- \( \eta \) is small constant (e.g., \( 0.1 \)) called learning rate

Can prove it will converge
- If training data is linearly separable
- and \( \eta \) is sufficiently small
| Linear Threshold Unit |

To understand, consider simple linear unit, where

\[ o = w_0 + w_1 x_1 + \ldots + w_n x_n \]

Idea: learn \( w_i \)'s that minimize the squared error

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

Where \( D \) is the set of training examples

---

| Gradient Descent |

---
Gradient Descent

Gradient \( \nabla E[\hat{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \)

Training rule: \( \Delta w_i = -\eta \nabla E[\hat{w}] \)

i.e., \( \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \)

\[
\begin{align*}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \hat{w} \cdot \hat{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{align*}
\]
Gradient Descent

GRADIENT-DESCENT(training_examples, \( \eta \))

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

- Initialize each \( w \) to some small random value
- Until the termination condition is met, do
  - Initialize each \( \Delta w \) to zero.
  - For each \( \langle \mathbf{x}, t \rangle \) in training_examples, do
    * Input the instance \( \mathbf{x} \) and compute output \( o \)
    * For each linear unit weight \( w \), do
      \[ \Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i \]
  - For each linear unit weight \( w \), do
    \[ w_i \leftarrow w_i + \Delta w_i \]

Summary

Perceptron training rule guaranteed to succeed if
- Training examples are linearly separable
- Sufficiently small learning rate \( \eta \)

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

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

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

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

$E_d[\tilde{w}] \equiv \frac{1}{2} (t_d - o_d)^2$

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

---

**Multilayer Networks of Sigmoid Units**

[Diagram of a multilayer network with sigmoid units]
Multilayer Decision Space

Sigmoid Unit

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

\[ o = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}} \]

Recall: 
\[ \frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)) \]

We can derive gradient descent rules to train:
- One sigmoid unit
- Multilayer networks of sigmoid units → Backpropagation
The Sigmoid Function

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

Sort of a rounded step function
Unlike step function, can take derivative (makes learning possible)

Error Gradient for a Sigmoid Unit

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

\[ = \frac{1}{2} \sum_{d} \frac{\partial}{\partial w_i} \left( t_d - o_d \right)^2 \]

\[ = \frac{1}{2} \sum_{d} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \]

\[ = \sum_{d} (t_d - o_d) \left( - \frac{\partial o_d}{\partial w_i} \right) \]

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

But we know:

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

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

So:

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

Initialize all weights to small random numbers. Until satisfied, do

- For each training example, do
  1. Input the training example and compute the outputs
  2. For each output unit $k$
     \[ \delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \]
  3. For each hidden unit $h$
     \[ \delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k \]
  4. Update each network weight $w_{i,j}$
     \[ w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j} \]
     where
     \[ \Delta w_{i,j} = \eta \cdot \delta_j x_{i,j} \]

**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 $\alpha$
  \[ \Delta w_{j,i}(n) = \eta \cdot \delta_j x_{j,i} + \alpha \cdot \Delta w_{j,i}(n - 1) \]
- Minimizes error over training examples
- Will it generalize well to subsequent examples?
- Training can take thousands of iterations -- slow!
  - Using network after training is fast
Learning Hidden Layer Representations

Inputs

Outputs

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000 → 10000000</td>
<td>.89</td>
<td>04.08</td>
</tr>
<tr>
<td>01000000 → 01000000</td>
<td>.01</td>
<td>11.88</td>
</tr>
<tr>
<td>00100000 → 00100000</td>
<td>.97</td>
<td>27</td>
</tr>
<tr>
<td>00010000 → 00010000</td>
<td>.99</td>
<td>97.71</td>
</tr>
<tr>
<td>00001000 → 00001000</td>
<td>.03</td>
<td>05.02</td>
</tr>
<tr>
<td>00000100 → 00000100</td>
<td>.22</td>
<td>99.99</td>
</tr>
<tr>
<td>00000010 → 00000010</td>
<td>.80</td>
<td>01.98</td>
</tr>
<tr>
<td>00000001 → 00000001</td>
<td>.60</td>
<td>94.01</td>
</tr>
</tbody>
</table>

Learning Hidden Layer Representations

Inputs

Outputs

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000 → 10000000</td>
<td>.01</td>
<td>.94</td>
</tr>
<tr>
<td>01000000 → 01000000</td>
<td>.98</td>
<td>.01</td>
</tr>
<tr>
<td>00100000 → 00100000</td>
<td>.99</td>
<td>.99</td>
</tr>
<tr>
<td>00010000 → 00010000</td>
<td>.02</td>
<td>.05</td>
</tr>
<tr>
<td>00001000 → 00001000</td>
<td>.71</td>
<td>.97</td>
</tr>
<tr>
<td>00000100 → 00000100</td>
<td>.27</td>
<td>.97</td>
</tr>
<tr>
<td>00000010 → 00000010</td>
<td>.88</td>
<td>.11</td>
</tr>
<tr>
<td>00000001 → 00000001</td>
<td>.08</td>
<td>.04</td>
</tr>
</tbody>
</table>
Output Unit Error during Training

Sum of squared errors for each output unit

Hidden Unit Encoding

Hidden unit encoding for one input
### Input to Hidden Weights

Weights from inputs to one hidden unit

### Convergence of Backpropagation

- Gradient descent to some local minimum
  - Perhaps not global minimum
  - Momentum can cause quicker convergence
  - Stochastic gradient descent also results in faster convergence
  - Can train multiple networks and get different results (using different initial weights)

**Nature of convergence**
- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions as training progresses
Expressive Capabilities of ANNs

Boolean functions:
- Every Boolean function can be represented by network with a single hidden layer
- But that might require an exponential (in the number of inputs) hidden units

Continuous functions:
- Every bounded continuous function can be approximated with arbitrarily small error by a 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

![Error versus weight updates (example 1)](image)
Overfitting in ANNs

Error versus weight updates (Example 2)

- Error on Training set and Validation set over number of weight updates.
- Overfitting occurs when the error on the validation set increases.

Neural Nets for Face Recognition

- Typical Input Images: 90% accurate learning, head pose, and recognizing 1-of-20 faces.
- 30x32 inputs
Learned Network Weights

Typical Input Images

30x32 inputs

Learned Weights

Alternative Error Functions

Penalize large weights:

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

Train on target slopes as well as values:

\[ E(\tilde{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \text{ outputs}} \left( (t_{kd} - o_{kd})^2 + \mu \left( \frac{\partial t_{kd}}{\partial x_d^1} - \frac{\partial o_{kd}}{\partial x_d^1} \right)^2 \right) \]

Tie together weights:

- e.g., in phoneme recognition
Recurrent Networks

What you should know

- ANNs are practical method for learning real-valued and vector-valued functions over continuous and discrete inputs
- Backpropagation can be used to find weights for multi-layer ANNs
- Overfitting is an issue for ANNs
- Many, many variants that we have not covered