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 $\sim 0.1$ 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
- 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
Perceptron

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

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

Sometimes we’ll 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 a Perceptron

(a) + + + + -- --
(b) + + - -

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’ll want networks of these...
Perceptron training rule

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

where

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

Where:

- \( t = c(\vec{x}) \) is target value
- \( o \) is perceptron output
- \( \eta \) is small constant (e.g., .1) called learning rate
Perceptron training rule

Can prove it will converge

- If training data is linearly separable
- and \( \eta \) sufficiently small
Gradient Descent

To understand, consider simpler linear unit, where

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

Let’s learn \( w_i \)’s that minimize the squared error

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

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

Gradient

$$\nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots \frac{\partial E}{\partial w_n} \right]$$

Training rule:

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

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$
Gradient Descent

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

**Gradient-Descent**(training examples, $\eta$)

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

- Initialize each $w_i$ to some small random value
- Until the termination condition is met, Do
  - Initialize each $\Delta w_i$ to zero.
  - For each $\langle \vec{x}, t \rangle$ in training examples, Do
    * Input the instance $\vec{x}$ to the unit and compute the output $o$
    * For each linear unit weight $w_i$, Do
      \[ \Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \]
  - For each linear unit weight $w_i$, 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 $\nabla E_D[\vec{w}]$
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
Multilayer Networks of Sigmoid Units
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 → Backpropagation
Error Gradient for a Sigmoid Unit

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

But we know:

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

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

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 to the network and compute the network 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

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

Sum of squared errors for each output unit
Training
Training

Weights from inputs to one hidden unit
Convergence of Backpropagation

Gradient descent to some local minimum

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

Nature of convergence

• Initialize weights near zero
• Therefore, initial networks near-linear
• Increasingly non-linear functions possible as training progresses
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].
Overfitting in ANNs

Error versus weight updates (example 1)

Error versus weight updates (example 2)
Neural Nets for Face Recognition

Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces
Learned Hidden Unit Weights

left strt rght up

Learned Weights

Typical input images

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

Penalize large weights:

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

Train on target slopes as well as values:

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

Tie together weights:

- e.g., in phoneme recognition network
Recurrent Networks

(a) Feedforward network

(b) Recurrent network

(c) Recurrent network unfolded in time