Lecture 5
Neural Networks

Connectionist Models
Consider humans:
- Neuron switching time ~ .001 second
- Number of neurons ~ $10^{10}$
- Connections per neuron ~ $10^{4-5}$
- Scene recognition time ~ .1 second
- 100 inference steps doesn’t seem like enough
  ⇒ Much parallel computation

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

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

Sometimes we'll use simpler vector notation:

\[ o(f) = \begin{cases} 
  1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\
  -1 & \text{otherwise.}
\end{cases} \]

Decision Surface of a Perceptron

(a) Represents some useful functions

- What weights represent \( g(x_1, x_2) = AND(x_1, x_2) \)?
- But some functions not representable
  - All 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:

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

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

\[ E[\vec{o}] \equiv \frac{1}{2} \sum_{(x, y) \in D} (t - o)^2 \]

Where \( D \) is set of training examples
Gradient:
\[
\nabla E[w] = \begin{bmatrix}
\frac{\partial E}{\partial w_1}, & \frac{\partial E}{\partial w_2}, & \ldots, & \frac{\partial E}{\partial w_n}
\end{bmatrix}
\]

Training rule:
\[
\Delta w_i = -\eta \nabla E[u]
\]
I.e.:
\[
\Delta w_i = -\eta \frac{\partial E}{\partial w_i}
\]

Gradient Descent

**Gradient Descent**

**(training, examples, \eta)**

Initialize each \(w_i\) to some small random value

Until the termination condition is met, \(\Delta w_i\) to zero.

- For each \((x, t)\) in \(training, examples, \eta\), Do
  - For each linear unit weight \(w_i\), Do
    - \(w_i \leftarrow w_i + \Delta w_i + \eta (t - o_i) x_i\)

Batch vs. Incremental Gradient Descent

**Batch Mode** Gradient Descent:

Do until convergence

1. Compute the gradient \(\nabla E_D[u]\)
2. \(u \leftarrow u - \eta \nabla E_D[u]\)

**Incremental Mode** Gradient Descent:

Do until convergence

For each training example \(d\) in \(D\)

1. Compute the gradient \(\nabla E_D[u]\)
2. \(u \leftarrow u - \eta \nabla E_D[u]\)

\[
E_D[u] = \frac{1}{2} \sum_{i=1}^{n} (t_i - o_i)^2
\]

\[
E[u] = \frac{1}{2} \sum_{i=1}^{n} (t_i - o_i)^2
\]

**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 hypotheses with minimum squared error
- Given sufficiently small learning rate \(\eta\)
- Even when training data contains noise
- Even when training data not separable by \(H\)
Multilayer Networks of Sigmoid Units

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

Error Gradient for a Sigmoid Unit

But we know:
\[
\frac{\partial o_i}{\partial \text{net}_i} = \sigma(\text{net}_i) = o_i(1-o_i)
\]
\[
\frac{\partial \text{net}_i}{\partial o_i} = \frac{\partial (\text{net}_i \cdot x_i)}{\partial o_i} = x_{i,d}
\]
So:
\[
\frac{\partial E}{\partial o_i} = - \sum_{d \in D} (t_d - o_i) o_i (1-o_i) x_{i,d}
\]
Let:
\[
\delta_i = - \frac{\partial E}{\partial \text{net}_i}
\]
Backpropagation Algorithm

- Initialize all weights to small random numbers
- Until convergence, Do
  1. Input it to network and compute 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 $h$
     \[ \delta_h \leftarrow o_h(1 - o_h) \sum_{k:\text{output}} w_{h,k} \delta_k \]
  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>1000000</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>0010000</td>
</tr>
<tr>
<td>0001000</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
</tr>
</tbody>
</table>

Can this be learned?

Learned hidden layer representation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Values</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>.89 .04 .08</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>.01 .11 .88</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>.01 .97 .27</td>
<td>0010000</td>
</tr>
<tr>
<td>0001000</td>
<td>.99 .97 .71</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>.03 .05 .02</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>.22 .99 .99</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>.80 .01 .98</td>
<td>0000001</td>
</tr>
<tr>
<td>0000001</td>
<td>.90 .94 .01</td>
<td>0000001</td>
</tr>
</tbody>
</table>

Training

Sum of squared errors for each output 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

Expressiveness of Neural Nets

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
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers
Overfitting Avoidance

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

Train on target slopes as well as values:
\[ E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{input}} \left[ (t_{kd} - o_{kd})^2 + \mu \sum_{p \in \text{input}} \left( \frac{\partial t_{kd}}{\partial w_{i,p}} - \frac{\partial t_{kd}}{\partial w_{j,p}} \right)^2 \right] \]

Weight sharing
Early stopping

Neural Networks: Summary

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation