Neural Networks

Slides adapted from Pedro Domingos and Vibhav Gogate

Connectionist Models

Consider humans:
- Neuron switching time $\sim .001$ second
- Number of neurons $\sim 10^{10}$
- Connections per neuron $\sim 10^{4-5}$
- Scene recognition time $\sim .1$ second
- 100 inference steps doesn’t seem like enough

$\Rightarrow$ 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:

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

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

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

\[ \nabla E[\bar{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

Training rule:

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

I.e.:

\[ \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]

---

**Gradient Descent**

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

GRADIENT-DESCENT($training\_examples, \eta$)

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 $(\vec{x}, t)$ in $training\_examples$, Do
  - Input instance $\vec{x}$ to unit and compute 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 $H$
Batch vs. Incremental Gradient Descent

**Batch Mode** Gradient Descent:
Do until convergence
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 convergence
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

\[ out(x) = g \left( w_0 + \sum_k w_k g \left( w_0^k + \sum_i w_i^k x_i \right) \right) \]
\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\( \sigma(x) \) is the sigmoid function

\[ \frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)) \]
We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

---

**Backpropagation Algorithm**

Initialize all weights to small random numbers

Until convergence, Do

For each training example, 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 $h$
   \[
   \delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k
   \]
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 $\alpha$
  \[
  \Delta w_{i,j}(n) = \eta \delta_i x_{i,j} + \alpha \Delta w_{i,j}(n - 1)
  \]
- 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?

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>
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 in Neural Nets

Error versus weight updates (example 1)

Training set error
Validation set error
Overfitting Avoidance

Penalize large weights:

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

Weight sharing
Early stopping
Neural Networks: Summary

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation