Brief History

• Try to create artificial intelligence based on natural intelligence

• The brain
  – massively interconnected neurons
Natural Neural Networks

- Signals “move” via electrochemical signals
- The synapses release a chemical transmitter
  - the sum of which can cause a threshold to be reached, causing the neuron to “fire”
- Synapses can be inhibitory or excitatory
- We are born with about 100 billion neurons
- A neuron may connect to as many as 100,000 other neurons
Real Neural Learning

- Synapses change size and strength with experience.
- Hebb rule
  - When two connected neurons are firing at the same time, the strength of the synapse between them increases.
- “Neurons that fire together, wire together.”
Artificial Neural Networks

- Analogy to biological neural systems
- Attempt to understand natural biological systems through computational modeling
- Intelligent behavior as an “emergent” property of large number of simple units rather than from explicitly encoded symbolic rules and algorithms
Modeling a Neuron

\[ a_i = g(in_i) \]

\[ in_i = \sum_j W_{j,i} a_j \]

- \( a_j \): Activation value of unit \( j \)
- \( W_{j,i} \): Weight on link from unit \( j \) to unit \( i \)
- \( in_i \): Weighted sum of inputs to unit \( i \)
- \( a_i \): Activation value of unit \( i \)
- \( g \): Activation function
Activation Functions

\begin{itemize}
  \item \textbf{Step}_t(x) = 1 \quad \text{if } x \geq t, \text{ else } 0 \quad \text{threshold}=t
  \item \text{Sign}(x) = +1 \quad \text{if } x \geq 0, \text{ else } -1 \quad \text{threshold}=0
  \item \text{Sigmoid}(x)= \frac{1}{1+e^{-x}}
\end{itemize}
McCollough and Pitts (1943) showed how such model neurons could compute logical functions and be used to construct finite-state machines.
The First Neural Networks

AND Function

\[ \begin{array}{c|c|c|c}
\hline
X_1 & X_2 & Y \\
\hline
1 & 1 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0 \\
\hline
\end{array} \]

Threshold(\(Y\)) = 2
The First Neural Networks

**OR Function**

$$\text{Threshold}(Y) = 2$$

<table>
<thead>
<tr>
<th>OR</th>
<th>X1</th>
<th>X2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The First Neural Networks

AND NOT Function

Threshold($Y$) = 2
Perceptron network

Perceptron Network

Single Perceptron
Perceptron Learning

• Assume supervised training examples giving the desired output for a unit given a set of known input activations.
• Learn synaptic weights so that unit produces the correct output for each example.
• Perceptron uses iterative update algorithm to learn a correct set of weights.
Perceptron Learning Rule

• Update weights by:

\[ w_{ji} = w_{ji} + \eta(t_i - o_i)I_j \]

where \( \eta \) is the “learning rate”
\( I_j \) is the \( j \)th input;
\( O_j \) is the predicted output, and \( t_i \) is the teacher specified output for output \( i \).

• Equivalent to rules:
  – If output is correct do nothing.
  – If output is high, lower weights on active inputs
  – If output is low, increase weights on active inputs

• Also adjust threshold to compensate:

\[ T_i = T_i - \eta(t_i - o_i) \]
### Perceptron Learning Algorithm

- **Iteratively update weights until convergence.**

  Initialize weights to random values  
  Until outputs of all training examples are correct  
  For each training pair, E, do:  
  - Compute current output $o_i$ for E given its inputs  
  - Compare current output to target value, $t_i$, for E  
  - Update synaptic weights and threshold using learning rule

- **Each execution of the outer loop is typically called an *epoch*.**
Perceptron as a Linear Separator

- Since perceptron uses linear threshold function, it is searching for a linear separator that discriminates the classes.

Or hyperplane in \( n \)-dimensional space

\[
\begin{align*}
    w_{12}o_2 + w_{13}o_3 & > T_1 \\
    o_3 & > -\frac{w_{12}}{w_{13}}o_2 + \frac{T_1}{w_{13}}
\end{align*}
\]
Perceptron as Hill Climbing

- Perceptron effectively does hill-climbing (gradient descent) in this space, changing the weights and threshold a small amount at each time to decrease training set error.
Perceptron Performance

• Linear threshold functions are restrictive (high bias) but still reasonably expressive.
• In practice, converges fairly quickly for linearly separable data.
• Experimentally, Perceptron does quite well on many benchmark data sets.
**Perceptron Limits**

- System obviously cannot learn concepts it cannot represent.
- Minsky and Papert (1969) wrote a book analyzing the perceptron and demonstrating many functions it could not learn.
- These results discouraged further research on neural nets; and symbolic AI became the dominate paradigm.
Concept Perceptron Cannot Learn

• Cannot learn exclusive-or, or parity function in general.
Multi-Layer Networks

• Multi-layer networks can represent arbitrary functions, but an effective learning algorithm for such networks was thought to be difficult.

• A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.

• The weights determine the function computed. Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer.
XOR network

XOR Function

\[
\begin{align*}
X_1 \text{ XOR } X_2 &= (X_1 \text{ AND NOT } X_2) \text{ OR } (X_2 \text{ AND NOT } X_1) \\
\text{Threshold}(Y) &= 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>XOR</th>
<th>X1</th>
<th>X2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Hill-Climbing in Multi-Layer Nets

- To do hill climbing (gradient descent), we need the output of a unit to be a differentiable function of its input and weights.
- Standard linear threshold function is not differentiable at the threshold.
Differentiable Output Function

- Need non-linear output function to move beyond linear functions.
  - A multi-layer linear network is still linear.
- Standard solution is to use the non-linear, differentiable sigmoidal “logistic” function:

\[
O_j = \frac{1}{1 + e^{-(\text{net}_j - T_j)}}
\]
Gradient Descent

• Define objective to minimize error:

\[ E(W) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2 \]

where \( D \) is the set of training examples, \( K \) is the set of output units, \( t_{kd} \) and \( o_{kd} \) are, respectively, the teacher and current output for unit \( k \) for example \( d \).

• Learning rule to change weights to minimize error is:

\[ \Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \]
Derivative of Sigmoid Function

- The derivative of a sigmoid unit with respect to an input is:

\[
o_j = g(in_j) \quad \frac{\partial o_j}{\partial in_j} = o_j(1 - o_j)
\]
Update rule for $w_{jk}$:

\[ W_{jk} \]

\[ W_{ij} \]
Error Backpropagation

Update rule for $w_{ij}$:

output, $k$
$W_{jk}$
hidden, $j$

$w_{ij}$
input, $i$
Backpropagation Learning Rule

Each weight changed by:

\[ \Delta w_{ij} = \eta \delta_j o_i \]

\[ \delta_j = o_j (1 - o_j) (t_j - o_j) \quad \text{if } j \text{ is an output unit} \]

\[ \delta_j = o_j (1 - o_j) \sum_k \delta_k w_{jk} \quad \text{if } j \text{ is a hidden unit} \]

where \( \eta \) is a constant called the learning rate
\( \delta_j \) is the error measure for unit \( j \)
Error Backpropagation

• First calculate error of output units and use this to change the top layer of weights.

Current output: $o_j = 0.2$
Correct output: $t_j = 1.0$
Error $\delta_j = o_j(1-o_j)(t_j-o_j) = 0.2(1-0.2)(1-0.2) = 0.128$

Update weights into $j$

$$\Delta w_{ij} = \eta \delta_j o_i$$
Error Backpropagation

- Next calculate error for hidden units based on errors on the output units it feeds into.

$$\delta_j = o_j (1 - o_j) \sum_k \delta_k w_{jk}$$
Error Backpropagation

- Finally update bottom layer of weights based on errors calculated for hidden units.

\[
\delta_j = o_j (1 - o_j) \sum_k \delta_k w_{jk}
\]

Update weights into \(j\)

\[
\Delta w_{ij} = \eta \delta_j o_i
\]
**Backpropagation Training Algorithm**

Create the 3-layer network with $H$ hidden units with full connectivity between layers. Set weights to small random real values. Until all training examples produce the correct value (within $\varepsilon$), or mean squared error ceases to decrease, or other termination criteria:

1. Begin epoch
2. For each training example, $d$, do:
   1. Calculate network output for $d$’s input values
   2. Compute error between current output and correct output for $d$
   3. Update weights by backpropagating error and using learning rule
3. End epoch
Comments on Training Algorithm

- May converge to local optima or oscillate indefinitely. However, in practice, does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (*random restarts*).
  - Take results of trial with lowest training set error.
  - Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).
- Neural Networks are prone to overfitting.
• Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.
• On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..
• But often times the non-linearities in the feature extraction can make interpretation of the hidden layers very difficult.
• This leads to Neural Networks being treated as black boxes.
**Successful Applications**

- Text to Speech (NetTalk)
- Fraud detection
- Financial Applications
  - HNC (eventually bought by Fair Isaac)
- Chemical Plant Control
  - Pavillion Technologies
- Automated Vehicles
- Game Playing
  - Neurogammon
- Handwriting recognition