Artificial Neural Networks (ANNs)

Biologically inspired computational model:

(1) **Simple** computational units (neurons).
(2) **Highly interconnected** - connectionist view
(3) Vast **parallel** computation, consider:
   - Human brain has $\sim 10^{11}$ neurons
   - Slow computational units, switching time $\sim 10^{-3}$ sec
     (compared to the computer $>10^{-10}$ sec)
   - Yet, you can recognize a face in $\sim 10^{-1}$ sec
   - This implies only about 100 sequential, computational neuron steps - this seems too low for something as complicated as recognizing a face
   - **Parallel processing**

ANNs are naturally parallel - each neuron is a self-contained computational unit that depends only on its inputs.
Learning

We have seen machine learning with different representations:

(1) Decision trees -- symbolic representation of various decision rules -- “disjunction of conjunctions”
(2) Perceptron -- learning of weights that represent an linear decision surface classifying a set of objects into two groups

Different representations give rise to different hypothesis or model spaces. Machine learning algorithms search these model spaces for the best fitting model.
The Perceptron

- A simple, single layered neural “network” - only has a single neuron.
- However, even this simple neural network is already powerful enough to perform (linear) classification tasks.
The Architecture

Perceptron Computation:
\[ y = \text{sgn}\left( b + \sum_{i=1}^{m} w_i x_i \right) \]

Note: \[ y \in \{+1,-1\} \]

Transfer Function:
\[ \text{sgn}(k) = \begin{cases} +1 & \text{if } k \geq 0 \\ -1 & \text{otherwise} \end{cases} \]
A perceptron computes the value,

\[ y = \text{sgn} \left( b + \sum_{i=1}^{m} w_i x_i \right) \]

Ignoring the activation function \( \text{sgn} \) and setting \( m = 1 \), we obtain,

\[ y' = b + w_1 x_1 \]

But this is the equation of a line with slope \( w \) and offset \( b \).

**Observation:** For the general case the perceptron computes a hyperplane in order to accomplish its classification task,

\[ y' = b + \sum_{i=1}^{m} w_i x_i = b + \vec{w} \cdot \vec{x} \]
Perceptron Learning Revisited

Initialize $\overrightarrow{w}$ and $b$ to random values.
repeat
  for each $(\overrightarrow{x}_i, y_i) \in D$ do
    if $\hat{f}(\overrightarrow{x}_i) \neq y_i$ then
      Update $\overrightarrow{w}$ and $b$ incrementally.
    end if
  end for
until $D$ is perfectly classified.
return $\overrightarrow{w}$ and $b$

Constructs a line (hyperplane) as a classifier

$$h + \overrightarrow{w} \cdot \overrightarrow{x}$$
In order for the hyperplane to become a classifier we need to find $b$ and $w$ => learning!
What About Non-Linearity?

Can we learn this decision surface? …Yes! Multi-Layer Perceptrons
Multi-Layer Perceptrons (ANNs)

Notice the smooth Transfer function!

Input Layer | Hidden Layer | Output Layer
---|---|---
\(x_0\) | \(x_1\) | \(x_2\) | ... | \(x_{n-1}\) | \(x_n\)

\[\begin{align*}
    y &= \sigma(\text{net}) \\
    \text{net} &= \sum_{i=0}^{n} w_i x_i \\
    o &= \frac{1}{1 + \epsilon^{-\text{net}}}
\end{align*}\]

≡ Linear/Input Unit
≡
How do we train?

Perceptron was easy:

Every time we found an error of the predicted value $f(x_i)$ compared to the label in the training set $y_i$, we update $w$ and $b$.

Not so easy in multi-layer neural networks – the error can occur deep in the network!
Artificial Neural Networks

We have to be a bit smarter in the case of ANNs: compute the signal (feed forward) and then use the error at the output to update all the weights by propagating the error back through the network.
Back Propagation Training
Backpropagation

\[ E = (y' - y)^2 \]  \hspace{1cm} \text{(output error)}

\[ \delta_0 = y(1 - y)E \]  \hspace{1cm} \text{(output node error)}

\[ w_{ho} \leftarrow w_{ho} + \alpha \delta_o \]  \hspace{1cm} \text{(weight update)}

\[ \delta_h = y(1 - y)w_{ho} \delta_o \]  \hspace{1cm} \text{(hidden node error)}

\[ w_{ih} \leftarrow w_{ih} + \alpha \delta_h \]  \hspace{1cm} \text{(weight update)}

This only works because

\[ \delta_o = y(1 - y)E = \frac{\partial E}{\partial w \cdot x} = \frac{\partial (y' - y)^2}{\partial w \cdot x} = 2(y' - y)\left(\frac{\partial y'}{\partial w \cdot x} - \frac{\partial y}{\partial w \cdot x}\right) \]

and the output y is differentiable because the transfer function is differentiable. Also note, everything is based on the rate of change of the error...we are searching in the direction where the rate of change will minimize the output error.

For this to work transfer Function has to be smooth!!
Initialize the weights in the network (often randomly)
Do
  For each example e in the training set
    // forward pass
    y = compute neural net output
    y' = label for e from training data
    Calculate error $E = (y' - y)^2$ at the output units
    // backward pass
    Compute error $\delta_o$ for weights from a hidden node $h$ to the output node $o$ using $E$
    Compute error $\delta_h$ for weights from an input node $i$ to hidden node $h$ using $\delta_o$
    Update the weights in the network
  Until all examples classified correctly or stopping criterion satisfied
Return the network
Neural Network Learning

- Define the network error in terms of weights $w$ as
  $$E_x(w) = (y' - y)^2$$
  for some training instance $x$.
- Use the gradient (slope) of the error surface to guide the search towards appropriate weights:
  $$\Delta w_k = -\eta \frac{\partial E_x}{\partial w_k}$$

☞ Backpropagation can be understood as a stochastic gradient search on the error surface of the network.
Representational Power

- Every bounded continuous function can be approximated with arbitrarily small error by a network with one hidden layer.
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.
Hidden Layer Representations

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?
This neural network architecture is sometimes also called autoencoder because of its ability to invent new representations of the input data and is a popular building block in deep-learning.