Recall: Classification Problem

How do we build a classifier to distinguish between faces and other objects?
The human brain is extremely good at classifying images

Can we develop classification methods by emulating the brain?

Neurons (Brain Cells)

Output spike roughly dependent on whether weighted sum of inputs reaches a threshold
Neurons as “Threshold Units”

Artificial neuron "spikes" (output = +1) if weighted sum exceeds threshold

Neurons are Classifiers!

Each “neuron” defines a hyperplane \( \sum_j w_{ji}u_j - \mu_i = 0 \)

Spike = +1 output (class \( C_1 \))

\( \sum_j w_{ji}u_j > \mu_i \)

No spike = -1 output (class \( C_2 \))

\( \sum_j w_{ji}u_j \leq \mu_i \)
**Neurons can compute functions**

**Example: AND function**

A separating hyperplane

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$v = 1$ iff $u_1 + u_2 - 1.5 > 0$

Similarly for OR and NOT

---

**What about the XOR function?**

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Can a neuron separate the +1 outputs from the -1 outputs?
Linear Inseparability

Artificial neuron with threshold fails if classification task is not linearly separable

- Example: XOR
  - No single line can separate the “yes” (+1) outputs from the “no” (-1) outputs!

Minsky and Papert’s book showing such negative results put a damper on neural networks research for over a decade!

How do we deal with linear inseparability?
Multilayer Networks

Removes limitations of single-layer networks
  • Can solve XOR

Example: Two-layer network that computes XOR

Output is +1 if and only if \( x + y - 2*(x + y > 1.5?) - 0.5 > 0 \)

Perceptron

Fancy name for layered “feed-forward” network (no loops)

Network of artificial neurons (“units”) with binary inputs and binary outputs (+1 or -1)
What if we want to learn continuous-valued functions?

This is called “regression” (or curve fitting) in statistics
• E.g., Linear regression = fitting a line to a set of points

Regression using Neural Networks

We want networks that can learn a function
• Network maps real-valued inputs to real-valued output

Continuous output values ⇒ Can’t use binary threshold units anymore
**Sigmoid Neurons**

\[ v = g(w \cdot u) \]

- **Input nodes**: \( u = (u_1, u_2, u_3)^T \)
- **Output**: \( v = g(w \cdot u) \)

**Sigmoid output function**:

\[ g(a) = \frac{1}{1 + e^{-\beta a}} \]

Non-linear “squashing” function: Squashes input to be between 0 and 1. Parameter \( \beta \) controls the slope.

---

**Learning the weights**

**Given**: Training data (input \( u \), desired output \( d \))

**Problem**: How do we learn the weights \( w \)?

**Idea**: Minimize squared error between network's output and desired output:

\[ E(w) = (d - v)^2 \]

where \( v = g(w \cdot u) \)

Starting from random values for \( w \), want to change \( w \) so that \( E(w) \) is minimized – How?
Learning by **Gradient-Descent**
( opposite of "Hill-Climbing")

Change \( w \) so that \( E(w) \) is minimized

- Use Gradient Descent: Change \( w \) in proportion to \(-dE/dw\) (why?)

\[
\begin{align*}
    w &\rightarrow w - \varepsilon \frac{dE}{dw} \\
    \frac{dE}{dw} &= -2(d - v) \frac{dv}{dw} = -2(d - v)g'(w \cdot u)u
\end{align*}
\]

- Derivative of sigmoid

\( \text{delta} = \text{error} \)

Also known as the “delta rule” or “LMS (least mean square) rule”

---

**But wait!**

This rule is for a one layer network

- One layer networks are not that interesting!!
  (remember XOR?)

What if we have multiple layers?
Learning Multilayer Networks

\[ v_i = g \left( \sum_j W_{ji} g \left( \sum_k w_{kj} u_k \right) \right) \]

Start with random weights \( W, w \)

Given input vector \( u \), network produces output vector \( v \)

Use gradient descent to find \( W \) and \( w \) that minimize total error over all output units (labeled \( i \)):

\[ E(W, w) = \frac{1}{2} \sum_i (d_i - v_i)^2 \]

This leads to the famous “Backpropagation” learning rule

Backpropagation: Output Weights

Learning rule for hidden-output weights \( W \):

\[ W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}} \quad \{ \text{gradient descent} \} \]

\[ \frac{dE}{dW_{ji}} = -(d_i - v_i) g'(\sum_j W_{ji} x_j) x_j \quad \{ \text{delta rule} \} \]
Backpropagation: Hidden Weights

\[ E(W,w) = \frac{1}{2} \sum_i (d_i - v_i)^2 \]

\[ v_i^m = g(\sum_j W_{ji} x_j) \]

\[ x_j = g(\sum_k w_{kj} u_k) \]

Learning rule for input-hidden weights \( w \):

\[ w_{kj} \rightarrow w_{kj} - \varepsilon \frac{dE}{dw_{kj}} \]

But:

\[ \frac{dE}{dw_{kj}} = \frac{dE}{dx_j} \cdot \frac{dx_j}{dw_{kj}} \]  \{chain rule\}

\[ \frac{dE}{dw_{kj}} = \left[ - \sum_i (d_i - v_i) g'(\sum_j W_{ji} x_j) W_{ji} \right] \cdot \left[ g'(\sum_k w_{kj} u_k) u_k \right] \]

Next Time

- Wrap up of machine learning
  - Learning to drive using neural networks
  - Ensemble learning
- To Do:
  - Project 4 due this Wednesday!
  - Read Chapter 18