What’s on the menu today?

- Supervised Learning
  - Why supervised learning?
    - Classification
  - Function Approximation
  - Perceptrons & Learning Rule
  - Linear Separability: Minsky-Papert deliver the bad news
  - Multilayer networks to the rescue
  - Function Approximation
  - Backpropagating (errors)
  - Radial Basis Function Networks
  - Recurrent Networks
  - Demos
Why Supervised Learning?

Two Primary Tasks

1. **Classification**
   - Inputs \( u_1, u_2, \ldots \) and discrete classes \( C_1, C_2, \ldots, C_k \)
   - Training examples: \((u_1, C_2), (u_2, C_7), \ldots\)
   - Learn the mapping from an arbitrary input to its class
   - Example: Inputs = images, output classes = face, not a face

2. **Function Approximation (regression)**
   - Inputs \( u_1, u_2, \ldots \) and continuous outputs \( v_1, v_2, \ldots \)
   - Training examples: (input, desired output) pairs
   - Learn to map an arbitrary input to its corresponding output
   - Example: Highway driving
     - Input = road image, output = steering angle

Perceptrons

- Fancy name for a type of layered feedforward networks
- Uses artificial neurons (“units”) with binary inputs and outputs
Perceptron uses “Threshold Units”

- Artificial neuron:
  - m binary inputs and 1 output (-1 or 1)
  - Synaptic weights $w_{ij}$
  - Threshold $\mu_i$

$$v_i = \Theta\left(\sum_j w_{ij}u_j - \mu_i\right)$$

$\Theta(x) = 1$ if $x \geq 0$ and -1 if $x < 0$

Perceptrons and Classification

- Consider a single-layer perceptron
  - Weighted sum forms a **linear hyperplane**
  - Everything **on one side** of this hyperplane is in **class 1** (output = +1) and everything **on other side** is class 2 (output = -1)
  - Any function that is linearly separable can be computed by a perceptron

- Example: AND is linearly separable
  - $a \text{ AND } b = 1$ if and only if $a = 1$ and $b = 1$
Perceptron Learning Rule

- Given inputs \( u \) and desired output \( v^d \), adjust \( w \) and \( \mu \) as follows:

  1. Compute error signal \( e = (v^d - v) \) where \( v \) is the current output:

     \[
     v = \Theta(\sum_j w_j u_j - \mu) = \Theta(w^T u - \mu)
     \]

  2. Change weights and threshold according to \( e \):

     \( \Rightarrow \) For positive inputs, increase weights if error is positive and decrease if error is negative

     \( \Rightarrow \) For positive inputs, decrease threshold if error is positive, increase if error is negative

     \[
     w \to w + \varepsilon(v^d - v)u
     \]

     \[
     \mu \to \mu - \varepsilon(v^d - v)
     \]

     \( A \to B \) means replace \( A \) with \( B \)

Linear Inseparability

- Single-layer perceptron with threshold units fails if classification task is not linearly separable

  \( \Rightarrow \) Example: XOR

  \( \Rightarrow \) a XOR \( b = 1 \) iff \( (a = -1, b = 1) \) or \( (a = 1, b = -1) \)

  \( \Rightarrow \) No single line can separate the “yes” (+1) outputs from the “no” (-1) outputs!

- Minsky and Papert’s book showing such negative results was very influential – put a damper on neural networks research for over a decade!
Solution in 1980s: Multilayer perceptrons

- Removes limitations of single-layer networks
  - Can solve XOR

- An example of a two-layer perceptron that computes XOR

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

Multilayer Perceptron: What does it do?
Example: Perceptrons as Constraint Satisfaction Networks

\[ 1 + \frac{1}{2} x - y < 0 \]
\[ 1 + \frac{1}{2} x - y > 0 \]
Example: Perceptrons as Constraint Satisfaction Networks

Perceptrons as Constraint Satisfaction Networks
Function Approximation

- We want networks that can learn a function
  - Network maps real-valued inputs to real-valued output
  - Want to generalize to predict outputs for new inputs
  - Idea: Given input data, minimize errors between network’s output and desired output by changing weights

Continuous output values \(\rightarrow\) Can’t use binary threshold units anymore

To minimize errors, a differentiable output function is desirable

\[
\begin{align*}
\text{Sigmoidal Networks} \\
\text{The most common activation function:} \\
\text{Sigmoid function:} \\
g(a) &= \frac{1}{1 + e^{-\beta a}} \\
(\text{non-linear “squashing” function})
\end{align*}
\]
Gradient-Descent Learning ("Hill-Climbing")

- Given training examples $(u^m, d^m)$ ($m = 1, \ldots, N$), define an error function (cost function or "energy" function)

$$E(w) = \frac{1}{2} \sum_m (d^m - v^m)^2 \quad v^m = g(w^T u^m)$$

- Would like to change $w$ so that $E(w)$ is minimized

$\Rightarrow$ Gradient Descent: Change $w$ in proportion to $-dE/dw$

$$w \rightarrow w - \varepsilon \frac{dE}{dw}$$

$$\frac{dE}{dw} = -\sum_m (d^m - v^m) \frac{dv^m}{dw} = -\sum_m (d^m - v^m) g'(w^T u^m) u^m$$

"Stochastic" Gradient Descent

- What if the inputs only arrive one-by-one?

- Stochastic gradient descent approximates sum over all inputs with an "on-line" running sum:

$$w \rightarrow w - \varepsilon \frac{dE_i}{dw}$$

$$\frac{dE_i}{dw} = -(d^m - v^m) g'(w^T u^m) u^m$$

Also known as the "delta rule" or "LMS rule"

delta = error
But wait….

- Delta rule tells us how to modify the connections from input to output (one layer network)
  - One layer networks are not that interesting (remember XOR?)

- What if we have multiple layers?

\[ u = (u_1 \ u_2 \ \ldots \ u_K)^T \]

\[ v = (v_1 \ v_2 \ \ldots \ v_J)^T; \ \text{Desired = } d \]

Delta rule can be used to adapt these weights

How do we adapt these?

Let’s Backpropagate (Errors)

- Backpropagation = gradient-descent learning for multilayer feedforward networks

- Idea: Propagate credit/blame for errors back to internal nodes
  - Use delta rule to change weights for output layer
  - Use chain rule (from calculus) to change weights for internal “hidden” nodes

\[ \text{error} = (d - v) \]

Backpropagate this to correct all weights
Notation for Backprop

\[ v_i^m = g\left(\sum_j W_{ij} g\left(\sum_k w_{jk} u_k^m\right)\right) \]

Find \( W \) and \( w \) that minimize total squared output error:

\[ E(W, w) = \frac{1}{2} \sum_m ||d_m^m - v_m^m||^2 \]

\[ = \frac{1}{2} \sum_{m,i} (d_i^m - v_i^m)^2 \]

Backpropagation (for Math lovers’ eyes only!)

✦ Learning rule for hidden-output connection weights:

\[ W_{ij} \rightarrow W_{ij} = -\varepsilon \frac{\partial E}{\partial W_{ij}} \]

\[ \frac{dE}{dW_{ij}} = -\sum_m (d_m^m - v_m^m) g'(\sum_j W_{ij} x_j^m) x_j^m \]

✦ Learning rule for input-hidden connection weights:

\[ w_{jk} \rightarrow w_{jk} = -\varepsilon \frac{\partial E}{\partial w_{jk}} \quad \text{But:} \quad \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial x_j^m} \cdot \frac{\partial x_j^m}{\partial w_{jk}} \quad \{\text{chain rule}\} \]

\[ \frac{dE}{dw_{jk}} = -\sum_{m,i} (d_i^m - v_i^m) g'(\sum_j W_{ij} x_j^m) W_{ij} g'(\sum_k w_{jk} u_k^m) u_k^m \]
Alternate Method: Radial Basis Function Networks

Radial Basis Function Networks

"activation" function:

\[ a_j = \sqrt{\sum_{i=1}^{n} (x_i - \mu_{i,j})^2} \]
Radial Basis Function Networks

output function: (Gaussian bell-shaped function)

\[ h(a) = e^{-\frac{a^2}{2\sigma^2}} \]

output of network:

\[ \text{out}_j = \sum_i w_{ij} h_i \]

• Main Idea: Use a mixture of Gaussians to approximate the output
• Gaussians are called “basis functions”
RBF networks

- Radial basis functions
  - Hidden units store means and variances
  - Hidden units compute a Gaussian function of inputs $x_1, \ldots, x_n$ that constitute the input vector $x$
- Learn weights $w_i$, means $\mu_i$, and variances $\sigma_i$ by minimizing squared error function (gradient descent learning)

$$h_i = \exp\left[ -\frac{(x - \mu_i)^T(x - \mu_i)}{2\sigma^2} \right], \quad y = \sum_i h_i w_i$$

RBF Networks and Multilayer Perceptrons
Recurrent Supervised Networks

✦ Why use recurrent networks?
  ➔ To keep track of recent history and context
  ➔ Can learn temporal patterns (time series or oscillations)

✦ Examples
  ➔ Hopfield network (see previous lecture and textbook)
  ➔ Recurrent backpropagation networks: for small sequences, *unfold network in time dimension* and use backpropagation learning
  ➔ Partially recurrent networks E.g. Elman net

Partially Recurrent Networks

✦ Example
  ➔ Elman net
    ➔ Partially recurrent
    ➔ Context units keep *internal memory of past inputs*
    ➔ Fixed context weights
    ➔ Backpropagation for learning
    ➔ E.g. Can disambiguate A→B→C and C→B→A

Elman network
**Demos** (by Keith Grochow, CSE 599, 2001)

- Neural network learns to balance a pole on a cart
  - System:
    - 4 state variables: $x_{\text{cart}}$, $v_{\text{cart}}$, $\theta_{\text{pole}}$, $v_{\text{pole}}$
    - 1 input: Force on cart
  - Backprop Network:
    - Input: State variables
    - Output: New force on cart
- NN learns to back a truck into a loading dock
  - System (Nyugen and Widrow, 1989):
    - State variables: $x_{\text{cab}}$, $y_{\text{cab}}$, $\theta_{\text{cab}}$
    - 1 input: new $\theta_{\text{steering}}$
  - Backprop Network:
    - Input: State variables
    - Output: Steering angle $\theta_{\text{steering}}$

Next Class: Reinforcement Learning

- Things to do:
  - Read Chapter 9
  - Finish Last Homework (due this Friday, 5pm)
  - Work on mini-project

I’ll be bäck (for reinf. learning)