Today:
- Artificial neural networks
- Backpropagation

Reading:
- Mitchell: Chapter 4
- Bishop: Chapter 5
Artificial Neural Network (ANN)

- **Biological systems** built of very complex webs of interconnected neurons.
- Highly connected to other neurons, and performs computations by combining signals from other neurons.
- Outputs of these computations may be transmitted to one or more other neurons.

- **Artificial Neural Networks** built out of a densely interconnected set of simple units (e.g., sigmoid units).
- Each unit takes real-valued inputs (possibly the outputs of other units) and produces a real-valued output (which may become input to many other units).
Connectionist Models

Consider humans:

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

→ much parallel computation

Properties of artificial neural nets (ANN’s):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
input: two features from spectral analysis of a spoken sound
output: vowel sound occurring in the context “h__d”
ALVINN
[Pomerleau 1993]
Artificial Neural Networks to learn $f: X \rightarrow Y$

- $f_w$ typically a non-linear function, $f_w: X \rightarrow Y$
- $X$ feature space: (vector of) continuous and/or discrete vars
- $Y$ output space: (vector of) continuous and/or discrete vars
- $f_w$ network of basic units

**Learning algorithm:** given $(x_d, t_d)_{d \in D}$, train weights $w$ of all units to minimize sum of squared errors of predicted network outputs.

Find parameters $w$ to minimize $\sum_{d \in D} (f_w(x_d) - t_d)^2$

Use gradient descent!
What type of units should we use?

- Classifier is a multilayer network of units.
- Each unit takes some inputs and produces one output. Output of one unit can be the input of another.
Multilayer network of Linear units?

- Advantage: we know how to do gradient descent on linear units

\[ x_0 = 1 \]
\[ v_1 = w_1 \cdot x \]
\[ v_2 = w_2 \cdot x \]
\[ z = w_3 \cdot v \]

Problem: linear of linear is just linear.

\[ z = w_{3,1}(w_1 \cdot x) + w_{3,2}(w_2 \cdot x) = (w_{3,1}w_1 + w_{3,2}w_2) \cdot x = \text{linear} \]
Multilayer network of Perceptron units?

- Advantage: Can produce highly non-linear decision boundaries!

![Diagram of a multilayer network of Perceptron units]

Threshold function: \( \int x = 1 \) if \( x \) is positive, 0 if \( x \) is negative.

Problem: discontinuous threshold is not differentiable. Can’t do gradient descent.
Multilayer network of sigmoid units

- Advantage: Can produce highly non-linear decision boundaries!
- Sigmoid is differentiable, so can use gradient descent

\[ v_1 = \sigma(w_1 \cdot x) \]
\[ v_2 = \sigma(w_2 \cdot x) \]
\[ z = \sigma(w_3 \cdot v) \]

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

Very useful in practice!
\( \sigma \) is the sigmoid function; \( \sigma(x) = \frac{1}{1+e^{-x}} \)

Nice property: \( \frac{d\sigma(x)}{dx} = \sigma(x) (1 - \sigma(x)) \)

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units \( \rightarrow \) Backpropagation
Gradient Descent to Minimize Squared Error

Goal: Given \((x_d, t_d)_{d\in D}\) find \(w\) to minimize \(E_D[w] = \frac{1}{2} \sum_{d\in D} (f_w(x_d) - t_d)^2\)

**Batch mode** Gradient Descent:
Do until satisfied
1. Compute the gradient \(\nabla E_D[w]\)
2. \(w \leftarrow w - \eta \nabla E_D[w]\)

**Incremental (stochastic)** Gradient Descent:
Do until satisfied
• For each training example \(d\) in \(D\)
  1. Compute the gradient \(\nabla E_d[w]\)
  2. \(w \leftarrow w - \eta \nabla E_d[w]\)

\(\nabla E[w] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right]\)

\(E_d[w] \equiv \frac{1}{2} (t_d - o_d)^2\)

*Note: Incremental Gradient Descent* can approximate *Batch Gradient Descent* arbitrarily closely if \(\eta\) made small enough
Gradient descent in weight space

Goal: Given \((x_d, t_d)_{d \in D}\) find \(w\) to minimize

\[
E_D[w] = \frac{1}{2} \sum_{d \in D} (f_w(x_d) - t_d)^2
\]

This error measure defines a surface over the hypothesis (i.e. weight) space

figure from Cho & Chow, *Neurocomputing* 1999
Gradient descent in weight space

Gradient descent is an iterative process aimed at finding a minimum in the error surface.

on each iteration
• current weights define a point in this space
• find direction in which error surface descends most steeply
• take a step (i.e. update weights) in that direction
Gradient descent in weight space

Calculate the gradient of $E$: \[ \nabla E(w) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

Take a step in the opposite direction

\[ w = \nabla E(w) \]

\[ w_i = \frac{\partial E}{\partial w_i} \]
Taking derivative: chain rule

Recall the chain rule from calculus

\[ y = f(u) \]

\[ u = g(x) \]

\[ \frac{y}{x} = \frac{y}{u} \frac{u}{x} \]
Gradient Descent for the Sigmoid Unit

Given \((x_d, t_d)_{d \in D}\) find \(w\) to minimize \(\sum_{d \in D} (o_d - t_d)^2\)

\[ o_d = \text{observed unit output for } x_d \]

\[ o_d = \sigma(\text{net}_d); \quad \text{net}_d = \sum_i w_i x_{i,d} \]

\[
\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 = \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) = \sum_{d \in D} (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right) \\
= -\sum_{d \in D} (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}
\]

But we know: \(\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d)\) and \(\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (w \cdot x_d)}{\partial w_i} = x_{i,d}\)

So: \(\frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}\)
Gradient Descent for the Sigmoid Unit

Given \((x_d, t_d)_{d \in D}\) find \(w\) to minimize \(\sum_{d \in D} (o_d - t_d)^2\]

\[o_d = \text{observed unit output for } x_d\]
\[o_d = \sigma(\text{net}_d); \quad \text{net}_d = \sum_i w_i x_{i,d}\]

\[
\frac{\partial E}{\partial w_i} = - \sum_{d \in D} (t_d - o_d) o_d (1 - o_d) x_{i,d}
\]

\(\delta_d\) error term \(t_d - o_d\) multiplied by \(o_d (1 - o_d)\) that comes from the derivative of the sigmoid function

\[
\frac{\partial E}{\partial w_i} = - \sum_{d \in D} \delta_d x_{i,d}
\]

Update rule: \(w \leftarrow w - \eta \nabla E[w]\)
Gradient Descent for Multilayer Networks

Given \((x_d, t_d)_{d \in D}\) find \(w\) to minimize \(\frac{1}{2} \sum_{d \in D} \sum_{k \in \text{Outputs}} (o_k,d - t_{kd})^2\)
Backpropagation Algorithm

Incremental/stochastic gradient descent

Initialize all weights to small random numbers.

**Until satisfied, Do:**

- **For** each training example \((x, t)\) **do**:
  1. Input the training example to the network and compute the 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 \( \alpha \)
  \[
  \Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n - 1)
  \]
• 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
Overfitting in ANNs

• Validation/generalization error first decreases, then increases.

• Weights tuned to fit the idiosyncrasies of the training examples that are not representative of the general distribution.

• Stop when lowest error over validation set.

• Not always obvious when lowest error over validation set has been reached.
Dealing with Overfitting

Our learning algorithm involves a parameter

\[ n \text{=} \text{number of gradient descent iterations} \]

How do we choose \( n \) to optimize future error?

- Separate available data into **training** and **validation** set
- Use **training** to perform gradient descent
- \( n \leftarrow \text{number of iterations that optimizes validation set error} \)
Dealing with Overfitting

• Regularization techniques
  • norm constraint
  • dropout
  • early stopping
  • ...

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
Expressive Capabilities of ANNs

Boolean functions:

- Every Boolean function can be represented by a network with a 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 [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrarily accuracy by a network with two hidden layers [Cybenko 1988]
Representing Simple Boolean Functions

Inputs $x_i \in \{0,1\}$

**Or function**

\[ x_{i_1} \lor x_{i_2} \lor \cdots \lor x_{i_k} \]

- $x_0 = 1$
- $w_i = 1$ if $i$ is an $i_j$
- $w_i = 0$ otherwise

**And function**

\[ x_{i_1} \land x_{i_2} \land \cdots \land x_{i_k} \]

- $x_0 = 1$
- $w_i = 1$ if $i$ is an $i_j$
- $w_i = 0$ otherwise

**And with negations**

\[ x_{i_1} \land \bar{x}_{i_2} \land \cdots \land x_{i_k} \]

- $x_0 = 1$
- $w_i = 1$ if $i$ is not negated
- $w_i = 0$ if $i$ is negated
- $w_i = -1$ if $i$ is $i_j$ negated
- $t = \#\text{ not negated}$

$w_i = 1$ if $i$ is an $i_j$

$w_i = 0$ otherwise
General Boolean functions

Every Boolean function can be represented by a network with a single hidden layer; might require exponential # of hidden units.

Can write any Boolean function as a truth table:

<table>
<thead>
<tr>
<th>000</th>
<th>+</th>
<th>View as OR of ANDs, with one AND for each positive entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>-</td>
<td>$\overline{x}_1 \overline{x}_2 \overline{x}_3 \lor \overline{x}_1 x_2 x_3 \lor x_1 x_2 \overline{x}_3 \lor x_1 x_2 x_3$</td>
</tr>
<tr>
<td>010</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Then combine AND and OR networks into a 2-layer network.
Other Activation Functions

- Generalizations of ReLU \( g\text{ReLU}(z) = \max\{z, 0\} + \alpha \min\{z, 0\} \)
- Leaky-ReLU\( (z) = \max\{z, 0\} + 0.01 \min\{z, 0\} \)
- Parametric-ReLU\( (z): \alpha \text{ learnable} \)
Artificial Neural Networks: Summary

• Highly non-linear regression/classification
• Vector valued inputs and outputs
• Potentially millions of parameters to estimate
• Actively used to model distributed computation in the brain
• Hidden layers learn intermediate representations

• Stochastic gradient descent, local minima problems
• Overfitting and how to deal with it.
Other Activation Functions

- Problem with sigmoid: saturation

Figure borrowed from *Pattern Recognition and Machine Learning*, Bishop
Other Activation Functions

- Activation function ReLU (rectified linear unit)

The Rectified Linear Activation Function

\[ g(z) = \max\{0, z\} \]

Figure from *Deep learning*, by Goodfellow, Bengio, Courville.
Other Activation Functions

- Activation function ReLU (rectified linear unit)

\[ g(z) = \max\{0, z\} \]

Gradient 0

Gradient 1