Multilayer Perceptrons and Intro to Support Vector Machines

Friday, 09 Friday 2007

William H. Hsu
Department of Computing and Information Sciences, KSU

Readings:
Sections 4.1-4.4, Mitchell
Section 2.2.6, Shavlik and Dietterich (Rosenblatt)
Section 2.4.5, Shavlik and Dietterich (Minsky and Papert)

Winnow Algorithm

- Algorithm *Train-Winnow* (*D*)
  - Initialize: \( \theta = n, w_i = 1 \)
  - UNTIL the termination condition is met, DO
    FOR each \( <x, t(x)> \) in *D*, DO
      1. CASE 1: no mistake - do nothing
      2. CASE 2: \( t(x) = 1 \) but \( w \cdot x < \theta - w_i \leftarrow 2w_i \text{ if } x_i = 1 \) (promotion/strengthening)
      3. CASE 3: \( t(x) = 0 \) but \( w \cdot x \geq \theta - w_i \leftarrow w_i / 2 \text{ if } x_i = 1 \) (demotion/weakening)
    - RETURN final *w*
- Winnow Algorithm Learns Linear Threshold (LT) Functions
- Converting to Disjunction Learning
  - Replace *demotion* with *elimination*
  - Change weight values to 0 instead of halving
  - Why does this work?
Winnow: An Example

\textbf{t}(x) = c(x) = x_1 \lor x_2 \lor x_{1023} \lor x_{1024}

- Initialize: \( \theta = n = 1024, w = (1, 1, ..., 1) \)
- \( <(1, 1, ..., 1), \lor> \quad w \cdot x \geq \theta, w = (1, 1, ..., 1) \quad \text{OK} \)
- \( <(0, 0, ..., 0), \lor> \quad w \cdot x < \theta, w = (1, 1, ..., 1) \quad \text{OK} \)
- \( <(1, 0, ..., 0), \lor> \quad w \cdot x < \theta, w = (2, 1, ..., 1) \quad \text{mistake} \)
- \( <(1, 0, 1, 1, 0, ..., 0), \lor> \quad w \cdot x < \theta, w = (4, 1, 2, 2, ..., 1) \quad \text{mistake} \)
- \( <(1, 0, 1, 0, 0, ..., 1), \lor> \quad w \cdot x < \theta, w = (8, 1, 4, 2, ..., 2) \quad \text{mistake} \)
- ... \( w = (512, 1, 256, 256, ..., 256) \)
- Promotions for each good variable: \(|\lg(n)| < \lg(n) + 1 = \lg(2n)\)
- \( <(1, 0, 1, 0, 0, ..., 1), \lor> \quad w \cdot x \geq \theta, w = (512, 1, 256, 256, ..., 256) \quad \text{OK} \)
- \( <(0, 0, 1, 0, 1, 1, 1, ..., 0), \lor> \quad w \cdot x \geq \theta, w = (512, 1, 0, 256, 0, 0, ..., 256) \quad \text{mistake} \)
- Last example: elimination rule (bit mask)
- Final Hypothesis: \( w = (1024, 1024, 0, 0, 0, 1, 32, ..., 1024, 1024) \)

Winnow: Mistake Bound

- Claim: \textit{Train-Winnow} makes \( O(k \log n) \) mistakes on \( k \)-disjunctions (\( \leq k \) of \( n \))
- Proof
  - \( u = \) number of mistakes on positive examples (promotions)
  - \( v = \) number of mistakes on negative examples (demotions/eliminations)
  - Lemma 1: \( u < k \lg (2n) = k (\lg n + 1) = k \lg n + k = O(k \log n) \)
  - Proof
    - A weight that corresponds to a good variable is only promoted
    - When these weights reach \( n \) there will be no more false positives
  - Lemma 2: \( v < 2(u + 1) \)
  - Proof
    - Total weight \( W = n \) initially
    - False positive: \( W(t+1) < W(t) + n \) - in worst case, every variable promoted
    - False negative: \( W(t+1) = W(t) - n/2 \) - elimination of a bad variable
    - \( 0 < W < n + un - vn/2 \Rightarrow v < 2(u + 1) \)
    - Number of mistakes: \( u + v < 3u + 2 = O(k \log n) \), Q.E.D.
Extensions to Winnow

- **Train-Winnow** Learns Monotone Disjunctions
  - Change of representation: can convert a general disjunctive formula
    - Duplicate each variable: \( x \rightarrow \{ y_+, y_- \} \)
    - \( y_+ \) denotes \( x \); \( y_- \) denotes \( \neg x \)
  - \( 2n \) variables - but can now learn general disjunctions!
  - NB: we’re not finished
    - \( \{ y_+, y_- \} \) are coupled
    - Need to keep two weights for each (original) variable and update both (how?)

- **Robust Winnow**
  - Adversarial game: may change \( c \) by adding (at cost 1) or deleting a variable \( x \)
  - Learner: makes prediction, then is told correct answer
  - **Train-Winnow-R**: same as **Train-Winnow**, but with lower weight bound of 1/2
  - **Claim**: Train-Winnow-R makes \( O(k \log n) \) mistakes (\( k \) = total cost of adversary)
  - **Proof**: generalization of previous claim

---

**NeuroSolutions and SNNS**

- **NeuroSolutions 5.x Specifications**
  - Commercial ANN simulation environment ([http://www.nd.com](http://www.nd.com)) for Windows NT
  - Supports multiple ANN architectures and training algorithms (temporal, modular)
  - Produces embedded systems
    - Extensive data handling and visualization capabilities
    - Fully modular (object-oriented) design
    - Code generation and dynamic link library (DLL) facilities
  - Benefits
    - Portability, parallelism: code tuning; fast offline learning
    - Dynamic linking: extensibility for research and development

- **Stuttgart Neural Network Simulator (SNNS) Specifications**
  - Open source ANN simulation environment for Linux
    - [http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/](http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/)
  - Supports multiple ANN architectures and training algorithms
  - Very extensive visualization facilities
  - Similar portability and parallelization benefits
Gradient Descent: Principle

• Understanding Gradient Descent for Linear Units
  – Consider simpler, unthresholded linear unit:
    \[ o(x) = \text{net}(x) = \sum_{i=0}^{n} w_i x_i \]
  – Objective: find “best fit” to \( D \)

• Approximation Algorithm
  – Quantitative objective: minimize error over training data set \( D \)
  – Error function: sum squared error (SSE)
    \[ E[w] = \text{error}_D[w] = \frac{1}{2} \sum_{i \in D} (t(x) - o(x))^2 \]

• How to Minimize?
  – Simple optimization
  – Move in direction of steepest gradient in weight-error space
    • Computed by finding tangent
    • i.e. partial derivatives (of \( E \)) with respect to weights (\( w \))

Gradient Descent: Derivation of Delta/LMS (Widrow-Hoff) Rule

• Definition: Gradient
  \[ \nabla E[w] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

• Modified Gradient Descent Training Rule
  \[ \Delta w = -r \nabla E[w] \]
  \[ \Delta w_i = -r \frac{\partial E}{\partial w_i} \]
  \[ \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \frac{1}{2} \sum_{i \in D} (t(x) - o(x))^2 \right] = \frac{1}{2} \sum_{i \in D} \frac{\partial}{\partial w_i} (t(x) - o(x))^2 \]
  \[ = \frac{1}{2} \sum_{i \in D} 2(t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - o(x)) = \sum_{i \in D} [(t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - w \cdot x)] \]
  \[ \frac{\partial E}{\partial w_i} = \sum_{i \in D} (t(x) - o(x)) (1 - x_i) \]
Gradient Descent: Algorithm using Delta/LMS Rule

- Algorithm Gradient-Descent \((D, r)\)
  - Each training example is a pair of the form \(\langle x, t(x) \rangle\), where \(x\) is the vector of input values and \(t(x)\) is the output value. \(r\) is the learning rate (e.g., 0.05)
  - Initialize all weights \(w_i\) to (small) random values
  - UNTIL the termination condition is met, DO
     - Initialize each \(\Delta w_i\) to zero
     - FOR each \(\langle x, t(x) \rangle\) in \(D\), DO
       - Input the instance \(x\) to the unit and compute the output \(o\)
         - FOR each linear unit weight \(w_i\), DO
           - \(\Delta w_i \leftarrow \Delta w_i + r(t - o)x_i\)
           - \(w_i \leftarrow w_i + \Delta w_i\)
     - RETURN final \(w\)

- Mechanics of Delta Rule
  - Gradient is based on a derivative
  - Significance: later, will use nonlinear activation functions (aka transfer functions, squashing functions)

LS Concepts: Can Achieve Perfect Classification
- Example A: perceptron training rule converges

Non-LS Concepts: Can Only Approximate
- Example B: not LS; delta rule converges, but can’t do better than 3 correct
- Example C: not LS; better results from delta rule

Weight Vector \(w = \text{Sum of Misclassified } x \in D\)
- Perceptron: minimize \(w\)
- Delta Rule: minimize error = distance from separator (i.e., maximize \(\frac{\partial E}{\partial w}\))
Incremental (Stochastic) Gradient Descent

- **Batch Mode Gradient Descent**
  - UNTIL the termination condition is met, DO
    1. Compute the gradient $\nabla E_D[\hat{w}]$
    2. $\hat{w} \leftarrow \hat{w} - r \nabla E_D[\hat{w}]$
  - RETURN final $\hat{w}$

- **Incremental (Online) Mode Gradient Descent**
  - UNTIL the termination condition is met, DO
    FOR each $<x, t(x)>$ in $D$, DO
      1. Compute the gradient $\nabla E_D[\hat{w}]$
      2. $\hat{w} \leftarrow \hat{w} - r \nabla E_D[\hat{w}]$
    RETURN final $\hat{w}$

- **Emulating Batch Mode**
  - Incremental gradient descent can approximate batch gradient descent arbitrarily closely if $r$ made small enough

Learning Disjunctions

- **Hidden Disjunction to Be Learned**
  - $c(x) = x_1' \lor x_2' \lor \ldots \lor x_m'$ (e.g., $x_2 \lor x_4 \lor x_5 \ldots \lor x_{100}$)
  - Number of disjunctions: $3^n$ (each $x_i$: included, negation included, or excluded)
  - Change of representation: can turn into a monotone disjunctive formula?
    - How?
      - How many disjunctions then?
    - Recall from COLT: mistake bounds
      - $\log|C| = O(n)$
      - Elimination algorithm makes $O(n)$ mistakes

- **Many Irrelevant Attributes**
  - Suppose only $k << n$ attributes occur in disjunction $c$ - i.e., $\log|C| = O(k \log n)$
  - Example: learning natural language (e.g., learning over text)
  - Idea: use a Winnow - perceptron-type LTU model (Littlestone, 1988)
    - Strengthen weights for false positives
    - Learn from negative examples too: weaken weights for false negatives
Winnow Algorithm

• Algorithm Train-Winnow (D)
  – Initialize: $\theta = n$, $w_i = 1$
  – UNTIL the termination condition is met, DO
    FOR each $<x, t(x)>$ in $D$, DO
      1. CASE 1: no mistake - do nothing
      2. CASE 2: $t(x) = 1$ but $w \cdot x < \theta - w_i \leftarrow 2w_i$ if $x_i = 1$ (promotion/strengthening)
      3. CASE 3: $t(x) = 0$ but $w \cdot x \geq \theta - w_i \leftarrow w_i / 2$ if $x_i = 1$ (demotion/weakening)
    – RETURN final $w$

• Winnow Algorithm Learns Linear Threshold (LT) Functions

• Converting to Disjunction Learning
  – Replace demotion with elimination
  – Change weight values to 0 instead of halving
  – Why does this work?

Winnow: An Example

• $t(x) = c(x) = x_1 \lor x_2 \lor x_{1023} \lor x_{1024}$
  – Initialize: $\theta = n = 1024$, $w = (1, 1, 1, ..., 1)$
  – $<(1, 1, 1, ..., 1), +>$ $w \cdot x \geq \theta$, $w = (512, 1, 256, 256, ..., 256)$ OK
  – $<(0, 0, 0, ..., 0), ->$ $w \cdot x < \theta$, $w = (2, 1, 1, ..., 1)$ OK
  – $<(1, 0, 1, 1, 0, 0, ..., 0), +>$ $w \cdot x < \theta$, $w = (2, 1, 1, 1, 1, 1, ..., 1)$ mistake
  – $<(1, 0, 1, 0, 0, 0, ..., 1), ->$ $w \cdot x < \theta$, $w = (512, 1, 256, 256, ..., 256)$ mistake
  – $<(1, 0, 0, 0, 0, 0, ..., 1), +>$ $w \cdot x < \theta$, $w = (8, 1, 4, 2, ..., 2)$ mistake
  – $<...>$ $w = (512, 1, 256, 256, ..., 256)$

• Promotions for each good variable: $|\lg(n)| < \lg(n) + 1 - \lg(2n)$
  – $<(1, 0, 1, 0, 0, ..., 1), +>$ $w \cdot x \geq \theta$, $w = (512, 1, 256, 256, ..., 256)$ OK
  – $<(0, 0, 1, 0, 1, 1, 0, 0, ..., 0), ->$ $w \cdot x \geq \theta$, $w = (512, 1, 0, 256, 0, 0, ..., 256)$ mistake
  – $<...>$ $w = (1024, 1024, 0, 0, 1, 32, ..., 1024, 1024)$

• Final Hypothesis: $w = (1024, 1024, 0, 0, 1, 32, ..., 1024, 1024)$
Winnow: Mistake Bound

- **Claim:** \(\text{Train-Winnow makes } O(k \log n) \text{ mistakes on } k\text{-disjunctions (} \leq k \text{ of } n\text{)} \)
- **Proof**
  - \(u\) = number of mistakes on positive examples (promotions)
  - \(v\) = number of mistakes on negative examples (demotions/eliminations)
  - **Lemma 1:** \(u < k \log (2n) = k (\log n + 1) = k \log n + k = O(k \log n)\)
    - **Proof**
    - A weight that corresponds to a good variable is only promoted
    - When these weights reach \(n\) there will be no more false positives
  - **Lemma 2:** \(v < 2(u + 1)\)
    - **Proof**
    - Total weight \(W = n\) initially
    - False positive: \(W(t+1) < W(t) + n\) - in worst case, every variable promoted
    - False negative: \(W(t+1) < W(t) - n/2\) - elimination of a bad variable
    - \(0 < W < n + un - vn/2 \Rightarrow v < 2(u + 1)\)
  - Number of mistakes: \(u + v < 3u + 2 = O(k \log n)\), Q.E.D.

Extensions to Winnow

- **Train-Winnow** Learns Monotone Disjunctions
  - Change of representation: can convert a general disjunctive formula
    - Duplicate each variable: \(x \rightarrow \{y_+, y_\}\)
    - \(y_+\) denotes \(x\); \(y_-\) denotes \(\neg x\)
    - \(2n\) variables - but can now learn general disjunctions!
  - NB: we’re not finished
    - \(\{y_+, y_\}\) are coupled
    - Need to keep two weights for each (original) variable and update both (how?)
- **Robust Winnow**
  - Adversarial game: may change \(c\) by adding (at cost 1) or deleting a variable \(x\)
  - Learner: makes prediction, then is told correct answer
  - **Train-Winnow-R:** same as **Train-Winnow**, but with lower weight bound of 1/2
  - **Claim:** **Train-Winnow-R** makes \(O(k \log n)\) mistakes (\(k = \) total cost of adversary)
  - **Proof:** generalization of previous claim
Multi-Layer Networks of Nonlinear Units

- **Nonlinear Units**
  - Recall: activation function $\text{sgn}(w \cdot x)$
  - Nonlinear activation function: generalization of $\text{sgn}$

- **Multi-Layer Networks**
  - A specific type: Multi-Layer Perceptrons (MLPs)
  - Definition: a multi-layer feedforward network is composed of an input layer, one or more hidden layers, and an output layer
  - “Layers”: counted in weight layers (e.g., 1 hidden layer = 2-layer network)
  - Only hidden and output layers contain perceptrons (threshold or nonlinear units)

- **MLPs in Theory**
  - Network (of 2 or more layers) can represent any function (arbitrarily small error)
  - Training even 3-unit multi-layer ANNs is $\text{NP}$-hard (Blum and Rivest, 1992)

- **MLPs in Practice**
  - Finding or designing effective networks for arbitrary functions is difficult
  - Training is very computation-intensive even when structure is “known”

Nonlinear Activation Functions

- **Sigmoid Activation Function**
  - Linear threshold gate activation function: $\text{sgn}(w \cdot x)$
  - Nonlinear activation (aka transfer, squashing) function: generalization of $\text{sgn}$
  - $\sigma$ is the sigmoid function $\sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$
  - Can derive gradient rules to train
    - One sigmoid unit
    - Multi-layer, feedforward networks of sigmoid units (using backpropagation)
  - **Hyperbolic Tangent Activation Function**
    - $\sigma(\text{net}) = \frac{e^{\text{net}} - e^{-\text{net}}}{e^{\text{net}} + e^{-\text{net}}}$
Error Gradient for a Sigmoid Unit

- Recall: Gradient of Error Function
  \[ \nabla E = \left[ \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

- Gradient of Sigmoid Activation Function
  \[ \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{(x_i \in D)} (t(x) - o(x))^2 = \frac{1}{2} \sum \left( \frac{\partial}{\partial w_i} (t(x) - o(x))^2 \right) \]
  \[ = \frac{1}{2} \sum_{(x_i \in D)} \left( 2(t(x) - o(x)) \frac{\partial}{\partial w_i} (t(x) - o(x)) \right) = \sum_{(x_i \in D)} \left( (t(x) - o(x)) \frac{\partial o(x)}{\partial w_i} \right) \]

- But We Know:
  \[ \frac{\partial o(x)}{\partial w_i} = \sigma'(w \cdot x) \cdot o(x)(1 - o(x)) \]
  \[ \frac{\partial o(x)}{\partial o(x)} = o(x)(1 - o(x)) \]

- So:
  \[ \frac{\partial E}{\partial w_i} = \sum_{(x_i \in D)} \left( (t(x) - o(x)) \cdot o(x)(1 - o(x)) \right) \cdot x_i \]

Backpropagation Algorithm

- Intuitive Idea: Distribute Blame for Error to Previous Layers
- Algorithm Train-by-Backprop \((D, r)\)
  - Each training example is a pair of the form \(<x, t(x)>\), where \(x\) is the vector of input values and \(t(x)\) is the output value. \(r\) is the learning rate (e.g., 0.05)
  - Initialize all weights \(w_i\) to (small) random values
  - UNTIL the termination condition is met, DO
  - FOR each \(<x, t(x)>\) in \(D\), DO
    - Input the instance \(x\) to the unit and compute the output \(o(x) = \sigma(\text{net}(x))\)
    - FOR each output unit \(k\), DO
      - \(\delta_k = o_k(x)(1 - o_k(x))t_k(x) - o_k(x)\)
    - FOR each hidden unit \(j\), DO
      - \(\delta_j = h_j(x)(1 - h_j(x)) \sum_{k\text{ outputs}} v_{jk}\delta_j\)
    - Update each \(w = u_{ij} (a = h_i)\) or \(w = v_{jk} (a = o_k)\)
      - \(w_{\text{start-layer, end-layer}} \leftarrow w_{\text{start-layer, end-layer}} + \Delta w_{\text{start-layer, end-layer}}\)
    - RETURN final \(u, v\)
Backpropagation and Local Optima

- **Gradient Descent in Backprop**
  - Performed over entire network weight vector
  - Easily generalized to arbitrary directed graphs
  - **Property**: Backprop on feedforward ANNs will find a local (not necessarily global) error minimum

- **Backprop in Practice**
  - Local optimization often works well (can run multiple times)
  - Often include weight momentum $\alpha$
    \[ \Delta w_{start-layer, end-layer}(n) = r \Delta w_{end-layer, start-layer} \]  
  - Minimizes error over training examples - generalization to subsequent instances?
  - Training often very slow: thousands of iterations over $D$ (epochs)
  - Inference (applying network after training) typically very fast
    - Classification
    - Control

Feedforward ANNs: Representational Power and Bias

- **Representational (i.e., Expressive) Power**
  - Backprop presented for feedforward ANNs with single hidden layer (2-layer)
  - 2-layer feedforward ANN
    - Any Boolean function (simulate a 2-layer AND-OR network)
    - Any bounded continuous function (*approximate with arbitrarily small error*)  
      [Cybenko, 1989; Hornik et al., 1989]
  - Sigmoid functions: set of basis functions; used to compose arbitrary functions
  - 3-layer feedforward ANN: any function (*approximate with arbitrarily small error*)  
    [Cybenko, 1988]
  - Functions that ANNs are good at acquiring: Network Efficiently Representable Functions (NERFs) - how to characterize?  
    [Russell and Norvig, 1995]

- **Inductive Bias of ANNs**
  - $n$-dimensional Euclidean space (weight space)
  - Continuous (error function smooth with respect to weight parameters)
  - Preference bias: “smooth interpolation” among positive examples
  - Not well understood yet (known to be computationally hard)
Learning Hidden Layer Representations

• Hidden Units and Feature Extraction
  – Training procedure: hidden unit representations that minimize error $E$
  – Sometimes backprop will define new hidden features that are not explicit in the input representation $x$, but which capture properties of the input instances that are most relevant to learning the target function $t(x)$
  – Hidden units express newly constructed features
  – Change of representation to linearly separable $D'$

• A Target Function (Sparse aka 1-of-C, Coding)

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0.01 0.1 0.1</td>
<td>0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

– Can this be learned? (Why or why not?)

Training:
Evolution of Error and Hidden Unit Encoding

$\text{error}_{D}(o_t)$

Sum of squared errors for each output unit

$h_j(01000000), 1 \leq j \leq 3$
Training: Weight Evolution

- Input-to-Hidden Unit Weights and Feature Extraction
  - Changes in first weight layer values correspond to changes in hidden layer encoding and consequent output squared errors
  - $w_0$ (bias weight, analogue of threshold in LTU) converges to a value near 0
  - Several changes in first 1000 epochs (different encodings)

Convergence of Backpropagation

- No Guarantee of Convergence to Global Optimum Solution
  - Compare: perceptron convergence (to best $h \in H$, provided $h \in H$; i.e., LS)
  - Gradient descent to some local error minimum (perhaps not global minimum...)
  - Possible improvements on backprop (BP)
    - Momentum term (BP variant with slightly different weight update rule)
    - Stochastic gradient descent (BP algorithm variant)
    - Train multiple nets with different initial weights; find a good mixture
  - Improvements on feedforward networks
    - Bayesian learning for ANNs (e.g., simulated annealing) - later
    - Other global optimization methods that integrate over multiple networks
- Nature of Convergence
  - Initialize weights near zero
  - Therefore, initial network near-linear
  - Increasingly non-linear functions possible as training progresses
Overfitting in ANNs

• Recall: Definition of Overfitting
  – \( h' \) worse than \( h \) on \( D_{\text{train}} \) but better on \( D_{\text{test}} \)

• Overtraining: A Type of Overfitting
  – Due to excessive iterations
  – Avoidance: stopping criterion (cross-validation: holdout, \( k \)-fold)
  – Avoidance: weight decay

Error versus epochs (Example 1)

Other Causes of Overfitting Possible

– Number of hidden units sometimes set in advance
– Too few hidden units (“underfitting”)
  • ANNs with no growth
  • Analogy: underdetermined linear system of equations (more unknowns than equations)
– Too many hidden units
  • ANNs with no pruning
  • Analogy: fitting a quadratic polynomial with an approximator of degree >> 2

Solution Approaches

– Prevention: attribute subset selection (using pre-filter or wrapper)
– Avoidance
  • Hold out cross-validation (CV) set or split \( k \) ways (when to stop?)
  • Weight decay: decrease each weight by some factor on each epoch
– Detection/recovery: random restarts, addition and deletion of weights, units
Example: Neural Nets for Face Recognition

- 90% Accurate Learning Head Pose, Recognizing 1-of-20 Faces

Example: NetTalk

- Sejnowski and Rosenberg, 1987
- Early Large-Scale Application of Backprop
  - Learning to convert text to speech
    - Acquired model: a mapping from letters to phonemes and stress marks
    - Output passed to a speech synthesizer
  - Good performance after training on a vocabulary of ~1000 words
- Very Sophisticated Input-Output Encoding
  - Input: 7-letter window; determines the phoneme for the center letter and context on each side; distributed (i.e., sparse) representation: 200 bits
  - Output: units for articulatory modifiers (e.g., “voiced”), stress, closest phoneme; distributed representation
  - 40 hidden units; 10000 weights total
- Experimental Results
  - Vocabulary: trained on 1024 of 1463 (informal) and 1000 of 20000 (dictionary)
  - 78% on informal, ~60% on dictionary

http://www.boltz.cs.cmu.edu/benchmarks/nettalk.html
Alternative Error Functions

- Penalize Large Weights (with Penalty Factor $w_p$)
  $$E(w) = \frac{1}{2} \sum_{(x,t) \in D} \sum_{k \text{ outputs}} (t_k(x) - o_k(x))^2 + w_p \sum_{\text{start - layer, end - layer}} w^2$$

- Train on Both Target Slopes and Values
  $$E(w) = \frac{1}{2} \sum_{(x,t) \in D} \sum_{k \text{ outputs}} (t_k(x) - o_k(x))^2 + w_p \sum_{i \text{ inputs}} \left( \frac{\partial t_i(x)}{\partial x_j} - \frac{\partial o_k(x)}{\partial x_j} \right)^2$$

- Tie Together Weights
  - e.g., in phoneme recognition network
  - See: Connectionist Speech Recognition [Bourlard and Morgan, 1994]

Recurrent Networks

- Representing Time Series with ANNs
  - Feedforward ANN: $y(t + 1) = \text{net}(x(t))$
  - Need to capture temporal relationships

- Solution Approaches
  - Directed cycles
  - Feedback
    - Output-to-input [Jordan]
    - Hidden-to-input [Elman]
    - Input-to-input
  - Captures time-lagged relationships
    - Among $x(t' \leq t)$ and $y(t + 1)$
    - Among $y(t' \leq t)$ and $y(t + 1)$
  - Learning with recurrent ANNs
    - Elman, 1990; Jordan, 1987
    - Principe and deVries, 1992
    - Mozer, 1994; Hsu and Ray, 1998
New Neuronal Models

- Neurons with State
  - Neuroids [Valiant, 1994]
  - Each basic unit may have a state
  - Each may use a different update rule (or compute differently based on state)
  - Adaptive model of network
    - Random graph structure
    - Basic elements receive meaning as part of learning process

- Pulse Coding
  - Spiking neurons [Maass and Schmitt, 1997]
  - Output represents more than activation level
  - Phase shift between firing sequences counts and adds expressivity

- New Update Rules
  - Non-additive update [Stein and Meredith, 1993; Seguin, 1998]
  - Spiking neuron model

- Other Temporal Codings: (Firing) Rate Coding

Some Current Issues and Open Problems in ANN Research

- Hybrid Approaches
  - Incorporating knowledge and analytical learning into ANNs
    - Knowledge-based neural networks [Flann and Dietterich, 1989]
  - Combining uncertain reasoning and ANN learning and inference
    - Probabilistic ANNs

- Global Optimization with ANNs
  - Markov chain Monte Carlo (MCMC) [Neal, 1996] - e.g., simulated annealing
  - Relationship to genetic algorithms - later

- Understanding ANN Output
  - Knowledge extraction from ANNs
    - Rule extraction
    - Other decision surfaces
  - Decision support and KDD applications [Fayyad et al, 1996]

- Many, Many More Issues (Robust Reasoning, Representations, etc.)
Terminology

- **Multi-Layer ANNs**
  - Focused on one species: (feedforward) multi-layer perceptrons (MLPs)
  - Input layer: an implicit layer containing $x_i$
  - Hidden layer: a layer containing input-to-hidden unit weights and producing $h_j$
  - Output layer: a layer containing hidden-to-output unit weights and producing $o_k$
  - $n$-layer ANN: an ANN containing $n - 1$ hidden layers
  - Epoch: one training iteration
  - Basis function: set of functions that span $H$

- **Overfitting**
  - Overfitting: $h$ does better than $h'$ on training data and worse on test data
  - Overtraining: overfitting due to training for too many epochs
  - Prevention, avoidance, and recovery techniques
    - Prevention: attribute subset selection
    - Avoidance: stopping (termination) criteria (CV-based), weight decay

- **Recurrent ANNs**: Temporal ANNs with Directed Cycles

Summary Points

- **Multi-Layer ANNs**
  - Focused on feedforward MLPs
  - Backpropagation of error: distributes penalty (loss) function throughout network
  - Gradient learning: takes derivative of error surface with respect to weights
    - Error is based on difference between desired output ($t$) and actual output ($o$)
    - Actual output ($o$) is based on activation function
    - Must take partial derivative of $\sigma \Rightarrow$ choose one that is easy to differentiate
    - Two $\sigma$ definitions: sigmoid (aka logistic) and hyperbolic tangent ($tanh$)

- **Overfitting in ANNs**
  - Prevention: attribute subset selection
  - Avoidance: cross-validation, weight decay

- **ANN Applications**: Face Recognition, Text-to-Speech

- **Open Problems**
  - Recurrent ANNs: Can Express Temporal Depth (Non-Markovity)

- **Next**: Statistical Foundations and Evaluation, Bayesian Learning Intro