Neural Networks

Introduction to Artificial Intelligence
CSE 150
May 29, 2007
Administration

- Last programming assignment has been posted!
- Final Exam: Tuesday, June 12, 11:30-2:30
Last Lecture

• Naïve Bayes classifier (needed for programming assignment)

• Reading
  – Chapter 13 (for probability)
This Lecture

• Neural Networks
  – Representation
  – Perceptron
  – Learning in linear networks

Reading

• Chapter 20, section 5 only
This Lecture

Neural Networks
What & Why

- Artificial Neural Networks: a bottom-up attempt to model the functionality of the brain
- Two main areas of activity:
  - Biological  
    Try to model biological neural systems
  - Computational
    - Artificial neural networks are biologically inspired but *not necessarily biologically plausible*
    - So may use other terms: Connectionism, Parallel Distributed Processing, Adaptive Systems Theory
- Interests in neural network differ according to profession.
NETTALK vs. DECTALK

DECTALK is a system developed by DEC which reads English characters and produces, with a 95% accuracy, the correct pronunciation for an input word or text pattern.

• DECTALK is an expert system which took 20 years to finalise
• It uses a list of pronunciation rules and a large dictionary of exceptions.

NETTALK, a neural network version of DECTALK, was constructed over one summer vacation.

• After 16 hours of training the system could read a 100-word sample text with 98% accuracy!
• When trained with 15,000 words it achieved 86% accuracy on a test set.

NETTALK is an example of one of the advantages of the neural network approach over the symbolic approach to Artificial Intelligence.

• It is difficult to simulate the learning process in a symbolic system; rules and exceptions must be known.
• On the other hand, neural systems exhibit learning very clearly; the network learns by example.
Consider humans:

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

Properties of ANNs:

- Many neuron-like switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
Perceptrons

Linear threshold units

\[ o = \begin{cases} 1 & \text{if } net > \theta \\ 0 & \text{otherwise} \end{cases} \]

\[ net = \sum_{i=0}^{n} w_i x_i \]

\[ o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_1 x_1 + \cdots + w_n x_n > \theta \\ -1 & \text{otherwise} \end{cases} \]

\( X_i \): inputs

\( W_i \): weights

\( \theta \): threshold
Perceptrons

The threshold can be easily forced to 0 by introducing an additional weight input $W_0 = \theta$ from a unit that is always -1.

\[ o(x_1, \ldots, x_n) = \begin{cases} 
1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\
-1 & \text{otherwise.}
\end{cases} \]
How powerful is a perceptron?
Threshold = 0

**Inverter**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$w_0 = 0.5$

$w_1 = -1$

**Boolean AND**

<table>
<thead>
<tr>
<th>input</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$w_0 = -1.5$

$w_1 = 1$

$w_2 = 1$

**Boolean OR**

<table>
<thead>
<tr>
<th>input</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$w_0 = -0.5$

$w_1 = 1$

$w_2 = 1$

**Boolean XOR**

<table>
<thead>
<tr>
<th>input</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Gary Cottrell’s modifications of slides originally produced by David Kriegman
Concept Space & Linear Separability

Linear Separability

Linear Separability

Linear Separability

Concept Space & Linear Separability

CSE 150, Spring 2007
Gary Cottrell’s modifications of slides originally produced by David Kriegman
Learning rule:

- If I’m ON and I’m supposed to be OFF, LOWER weights from active inputs and RAISE the threshold
- If I’m OFF and I’m supposed to be ON, RAISE weights from active inputs and LOWER the threshold
Training Perceptrons

Perceptron Training Rule

\[ w_i' \leftarrow w_i + \Delta w_i \]

\[ \Delta w_i = \eta (t - o) x_i \]

Where:
- \( t \) is target value
- \( o \) is perceptron output
- \( \eta \) is small constant (e.g., .1) called learning rate

Converges, if...

- training data linearly separable
- step size \( \eta \) sufficiently small
- no “hidden” units
Gradient Descent

Learn $w_i$'s that minimize squared error

$$E[\vec{w}] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$D$ = training data
Gradient Descent

Gradient: \[ \nabla E[\vec{w}] = \begin{bmatrix} \frac{\partial E}{\partial w_0} & \frac{\partial E}{\partial w_1} & \cdots & \frac{\partial E}{\partial w_n} \end{bmatrix} \]

Training rule: \[ \Delta \vec{w} = -\eta \nabla E[\vec{w}] \]

\[ \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]
Gradient Descent

- To find the best direction in the feature space we compute the gradient of $E$ with respect to each of the components of $\mathbf{w}$

$$\nabla E(\mathbf{w}) \equiv [\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}]$$

- This vector specifies the direction the produces the steepest increase in $E$;
- We want to modify $\mathbf{w}$ in the direction of $-\nabla E(\mathbf{w})$

$$\mathbf{w} = \mathbf{w} + \Delta \mathbf{w}$$

- Where:

$$\Delta \mathbf{w} = - R \nabla E(\mathbf{w})$$
• STOP HERE!!!
• Now do: derivation of the learning rule for linear networks
• Derivation of the learning rule for non-linear networks.
Batch Learning

- Initialize each \( w_i \) to small random value

- Repeat until termination:

  \[
  \Delta w_i = 0
  \]

  For each training example \( d \) do

  \[
  o_d \leftarrow \sigma(\Sigma_i w_i x_{i,d})
  \]

  \[
  \Delta w_i \leftarrow \Delta w_i + \eta (t_d - o_d) o_d (1-o_d) x_{i,d}
  \]

  \[
  w_i \leftarrow w_i + \Delta w_i
  \]
Begin: Some slides borrowed and modified from Sanjoy Dasgupta
Learning classifiers

Input space: \( X = \{28 \times 28 \text{ images}\} \)

Classes (or labels): \( Y = \{0, 1, 2, \ldots, 9\} \)

Given some labeled examples, learn to classify images, i.e., learn a prediction rule

\[ f: X \rightarrow Y \]

Most work assumes a binary classification task, such as \( Y = \{-1, +1\} \), eg.

-1: \{digits 0, 2, 3, 4, 5, 6, 7, 8, 9\}

+1: \{digit 1\}
“Batch” learning

1. Start with a set of labeled examples.

2. Learn an underlying rule $f: X \to Y$.

3. Use this to classify future examples.
The images $\mathbf{x}$ lie in $\mathbb{R}^{784}$

ASSUME: they have had the mean subtracted: If they are linearly separable - then we don’t have to worry about the threshold.

Linear classifier parametrized by a weight vector $\mathbf{w}$:

$$f_{\mathbf{w}}(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$
Learning linear classifiers

Given training data

\((x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\)

Find a vector \(w\) such that

\[ y^{(i)} (w \cdot x^{(i)}) \geq 0 \text{ for all } i = 1, \ldots, m \]

This is *linear programming*. 
The trouble with LP

DIFFICULT

Margin $r$

EASY
When there’s a margin:

Perceptron algorithm (Rosenblatt, 1958)

\[ w = 0 \]
\[ \text{while some } (x,y) \text{ is misclassified:} \]
\[ w = w + yx \]

**Claim:** If all points have length at most one, and there is a margin \( r > 0 \), then this algorithm makes at most \( 1/r^2 \) updates.
Perceptron in action

\[ w = x^{(1)} - x^{(7)} \]

Sparse representation:
\[ [(1,1), (7,-1)] \]

\[ w = 0 \]
while some \((x,y)\) is misclassified:
\[ w = w + yx \]
End: Some slides borrowed and modified from Sanjoy Dasgupta

- I didn’t really use the following slides, but you may find them useful!!
- Most of what I did was on the board…
Incremental (Online) Learning

- Initialize each $w_i$ to small random value
- Repeat until termination:
  - For each training example $d$ do
    - $\Delta w_i = 0$
    - $o_d \leftarrow \sum_i w_i x_{i,d}$
    - $\Delta w_i \leftarrow \Delta w_i + \eta (t_d - o_d) o_d (1-o_d) x_{i,d}$
    - $w_i \leftarrow w_i + \Delta w_i$
Summary: Single Layer Networks

- Variety of update rules
  - Multiplicative \( \Delta w_i = R(t_d - o_d)x_{id} \)
  - Additive
- Batch and incremental algorithms
- Various convergence and efficiency conditions
- There are other ways to learn linear functions
  - Linear Programming (general purpose)
  - Probabilistic Classifiers (some assumption)

- Although simple and restrictive -- linear predictors perform very well on many realistic problems
- However, the representational restriction is limiting in many applications
Increasing Expressiveness: Multi-Layer Neural Networks

Boolean XOR

<table>
<thead>
<tr>
<th>input \ x1</th>
<th>input \ x2</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2-layer Neural Net
Multi-Layer Neural Network

- Multi-layer networks can represent arbitrary functions, but building effective learning methods for such network was thought to be difficult.

- Networks are composed of an input layer, hidden layer(s) and output layer. Activation is feed-forward from input to output.

![Diagram of a multi-layer neural network with layers labeled as input, hidden, and output, and activation arrows pointing upwards.]
Multi-Layer Neural Network

- Patterns of activation are presented at the inputs and the resulting activation of the outputs is computed.

- The values of the weights determine the function computed. A network with one hidden layer is sufficient to represent every Boolean function. With real weights every real valued function can be approximated with a single hidden layer.
Basic Unit in Multi-Layer Neural Network

- **Linear Unit**: $o_j = \mathbf{w} \cdot \mathbf{x}$ multiple layers of linear functions produce linear functions. We want to represent nonlinear functions.

- **Threshold units**: $o_j = \text{sgn}(\mathbf{w} \cdot \mathbf{x} - T)$ are not differentiable, hence unsuitable for gradient descent.

- Use a non-linear, differentiable output function such as the sigmoid (or logistic) function:

  $$O_j = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} - T_j)}}$$
**Model Neuron (Logistic)**

- Use a non-linear, differentiable output function such as the sigmoid or logistic function.

\[
\text{net}_j = \sum w_{ij} \cdot x_i
\]

- Net input to a unit is defined as:

\[
O_j = \frac{1}{1 + e^{-(\text{net}_j - T_j)}}
\]
Representational Power

• The Backpropagation version presented is for networks with a single hidden layer,

But:

• Any Boolean function can be represented by a two layer network (simulate a two layer AND-OR network)

• Any bounded continuous function can be approximated with arbitrary small error by a two layer network. Sigmoid functions provide a set of basis function from which arbitrary function can be composed.

• Any function can be approximated to arbitrary accuracy by a three layer network.
Backpropagation Learning Rule

Since there are multiple output units, we define the error $E$ as the sum over all the network output units.

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2$$

where $D$ is the set of training examples,

$K$ is the set of output units

This can be used to derive the (global) learning rule which performs gradient descent in the weight space in an attempt to minimize the error function.

$$\Delta \vec{w}_{i\ell} = -\alpha \frac{\partial E}{\partial \vec{w}_{i\ell}}$$
Backpropagation Training Algorithm

- Create a fully connected three layer network. Initialize weights.

Until all examples produce the correct output within \( \varepsilon \) (or other criteria):

- For each example in the training set do:
  - Compute the network output for this example
  - Compute the error between the output and target value
  - For each output unit \( k \), compute error term
    \[
    \delta_k = (t_k - o_k) o_k (1 - o_k)
    \]
  - For each hidden unit, compute error term:
    \[
    \delta_j = o_j (1 - o_j) \cdot \sum_{k \in \text{downstream}(j)} -\delta_k w_{jk}
    \]
  - Update network weights
    \[
    \Delta w_{ij} = \eta \delta_j x_{ij}
    \]
- End epoch
Comments on Training

• No guarantee of convergence; may oscillate or reach a local minimum

• In practice, many large networks can be trained on large amounts of data for realistic problems.

• Many epochs (tens of thousands) may be needed for adequate training. Large datasets may require many hours or days of CPU time.

• Termination criteria: Number of epochs; threshold on training set error; Not decrease in error, increased error on validation set.

• To avoid local minima: several trials with different random initial weights.
Preference Bias: Early Stopping

- Overfitting in ANN
- Stop training when error goes up on validation set
Advanced Topic in Neural Networks

- Recurrent network: Network with feedback loops

Output $Y(t)$

Hidden Nodes $H(t)$

Inputs $H(t-1)$
**New Neuron Models**

- **Neurons with states:** (Neuroids, Valiant 94)
  - Each basic unit may have a state, and may use a different update rule, or just compute differently depending on the state.
  - Adaptive model of a network: a random graph structure in which basic elements receive a “meaning” as part of the learning process.

- **Spiking Neurons:** (Maass)
  Output represents more than just firing rate.
  Phase shift between firing sequences counts and adds expressivity

- **New Update rules:**
  - multiplicative update
  - Spiking neuron model
Conclusions

• Perceptrons: learning is convergent, but limited expressiveness.
• Multi-layer neural networks are more expressive and can be trained using gradient descent (backprop).
• Much slower to train than other learning methods, but exploring a rich space works well in many domains.
• Representation is not very accessible and difficult to take advantage of domain (prior) knowledge
• Analogy to brain and successful applications have generated significant interest.