CSC242: Intro to AI

Lecture 19:
Neural Networks Part II
Part I Review
Reserve Readings

Artificial Neural Networks
Chapter 4 of *Machine Learning* by Tom Mitchell

Click here

Then here
Artificial Neural Networks

[Read Ch. 4]
[Recommended exercises 4.1, 4.2, 4.5, 4.9, 4.11]

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face Recognition
- Advanced topics
Perceptron

\[ o(x_1, \ldots, x_n) = \begin{cases} 
1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\
-1 & \text{otherwise.} 
\end{cases} \]

Sometimes we’ll use simpler vector notation:

\[ o(\vec{x}) = \begin{cases} 
1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\
-1 & \text{otherwise.} 
\end{cases} \]
Linear Classifier

\[ w_0 + w_1 x_1 + w_2 x_2 = 0 \]

\[ w \cdot x = 0 \]

All instances of one class are above the line: \( w \cdot x > 0 \)

All instances of one class are below the line: \( w \cdot x < 0 \)

\[ h_w(x) = \text{Threshold}(w \cdot x) \]
\[ b(x_1, x_2) = x_2 - 1.7x_1 + 4.9 \]
Training

- **Training** is using data to set the weights for a perceptron (or network of perceptrons)

- **Idea:**
  - Start with random weights
  - For each piece of data:
    - Set inputs to the data features
    - Compare output to the label (target value)
    - If not same then adjust the weights
Perceptron training rule

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

where

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

Where:

- \( t = c(\vec{x}) \) is target value
- \( o \) is perceptron output
- \( \eta \) is small constant (e.g., 0.1) called learning rate
Gradient Descent

\textbf{Gradient-Descent}(training_examples, \eta)

Each training example is a pair of the form \(\langle \vec{x}, t \rangle\), where \(\vec{x}\) is the vector of input values, and \(t\) is the target output value. \(\eta\) is the learning rate (e.g., .05).

- Initialize each \(w_i\) to some small random value
- Until the termination condition is met, Do
  - Initialize each \(\Delta w_i\) to zero.
  - For each \(\langle \vec{x}, t \rangle\) in training_examples, Do
    * Input the instance \(\vec{x}\) to the unit and compute the output \(o\)
    * For each linear unit weight \(w_i\), Do
      \[ \Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i \]
  - For each linear unit weight \(w_i\), Do
    \[ w_i \leftarrow w_i + \Delta w_i \]
Gradient
\[ \nabla E[\tilde{w}] = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]
Training rule:
\[ \Delta \tilde{w} = -\eta \nabla E[\tilde{w}] \]
i.e.,
\[ \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]
Part II
Multilayer Neural Nets
Backpropagation
Deep Learning
Face Recognition Project
Multilayer Networks of Sigmoid Units
How does this differ from a Perceptron?

Smooth threshold
Range 0 to 1, not -1 to +1

$\sigma(x)$ is the sigmoid function

$$1 \quad 1 + e^{-x}$$

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

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\rightarrow$ Backpropagation
Error Gradient for a Sigmoid Unit
Error Gradient for a Sigmoid Unit

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

\[
= \frac{1}{2} \sum_{d} \frac{\partial}{\partial w_i} (t_d - o_d)^2
\]

\[
= \frac{1}{2} \sum_{d} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)
\]

\[
= \sum_{d} (t_d - o_d) \left( -\frac{\partial o_d}{\partial w_i} \right)
\]

\[
= -\sum_{d} (t_d - o_d) \frac{\partial o_d}{\partial \text{net}_d} \frac{\partial \text{net}_d}{\partial w_i}
\]

But we know:

\[
\frac{\partial o_d}{\partial \text{net}_d} = \frac{\partial \sigma(\text{net}_d)}{\partial \text{net}_d} = o_d(1 - o_d)
\]

\[
\frac{\partial \text{net}_d}{\partial w_i} = \frac{\partial (\bar{w} \cdot \bar{x}_d)}{\partial w_i} = x_{i,d}
\]

So:

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

- Perceptron:
  \[
  \frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})
  \]
  \[
  \Delta w_i = \eta(t - o)x_i
  \]

- Sigmoidal Unit:
  \[
  \frac{\partial E}{\partial w_i} = -\sum_{d \in D} (t_d - o_d)o_d(1 - o_d)x_{i,d}
  \]
  \[
  \Delta w_i = \eta(t - o)o(1 - o)x_i
  \]
Multilayer Networks

- How to train the hidden (middle) units?
  - We don’t have a target output for them!
- Idea:
  - First, calculate deltas for output units
  - Use the weighted average of these deltas to compute a delta for each hidden unit
Backpropagation Algorithm

Initialize all weights to small random numbers. Until satisfied, Do

- For each training example, 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
Learning Hidden Layer Representations

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Can this be learned??
Learning Hidden Layer Representations

A network:

Learned hidden layer representation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>→ .89 .04 .08</td>
<td>→ 10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>→ .01 .11 .88</td>
<td>→ 01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>→ .01 .97 .27</td>
<td>→ 00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>→ .99 .97 .71</td>
<td>→ 00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>→ .03 .05 .02</td>
<td>→ 00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>→ .22 .99 .99</td>
<td>→ 00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>→ .80 .01 .98</td>
<td>→ 00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>→ .60 .94 .01</td>
<td>→ 00000001</td>
</tr>
</tbody>
</table>
Training

Sum of squared errors for each output unit
Training

Weights from inputs to one hidden unit
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 network with 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 arbitrary accuracy by a network with two hidden layers [Cybenko 1988].
Using Validation Set to Prevent Overfitting

![Graph showing error versus weight updates](image)

- **Training set error**
- **Validation set error**
Neural Nets for Face Recognition

90% accurate learning head pose, and recognizing 1-of-20 faces
Learned Hidden Unit Weights

left  strt  right  up

30x32 inputs

Learned Weights

Typical input images
The code for this assignment is broken into several modules:

- **facetrain.c**: the top-level program which uses all of the modules below to implement an image classification system. You will need to modify this code to change network sizes and learning parameters. The performance evaluation routines `performance_on_imagelist()` and `evaluate_performance()` are also in this module; you will need to modify these for your face and expression recognizers.

- **imagenet.c**: interface routines for loading images into the input units of a network, and setting up target vectors for training. You will need to modify the routine `load_target`, when implementing the face recognizer and the pose recognizer, to set up appropriate target vectors for the output encodings you choose.

- **backprop.c, backprop.h**: the neural network package. Supports three-layer fully-connected feedforward networks, using the backpropagation algorithm for weight tuning. Provides high level routines for creating, training, and using networks. **You will not need to modify any code in this module to complete the assignment.**
Coming Up

- April 17 – In-Class Workshop for Project 3
  - Live highly attractive TAs will personally help you complete the project!
  - An afternoon you will not soon forget!