Lecture 6: Learning - Artificial Neural Networks

- ♢ Overview
- ♢ Perceptron learning
- ♢ Sigmoid Units
- ♢ Multi-layered feed-forward neural networks
- ♢ Backpropagation

Connectionist Models

Based on the assumption that a computational architecture similar to the brain would duplicate (at least some of) its wonderful abilities.

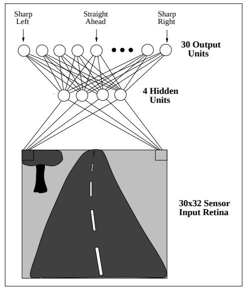
Properties of artificial neural nets (ANNs):
  Many neuron-like threshold switching units
  Many weighted interconnections among units
  Highly parallel, distributed process
  Emphasis on tuning weights automatically

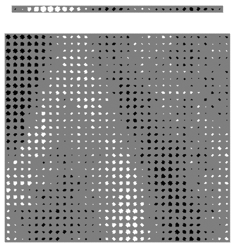
MANY different kinds of architectures, motivated both by biology and mathematics/efficiency of computation

Consider the human brain

- Contains ~ 10^{10} neurons, each of which may have up to ~ 10^{4-5} input/output connections
- Each neuron is fairly slow, with a switching time of ~ 1 milisecond
- Yet the brain is very fast and reliable at computationally intensive tasks (e.g. vision, speech recognition, knowledge retrieval)
- Although computers are at least 1 million times faster in raw switching speed!
- The brain is also more fault-tolerant, and exhibits graceful degradation with damage
- Maybe this is due to its architecture, which ensures massive parallel computation!

Example: ALVINN (Pomerleau, 1993)





What is a neural network?

A graph of simple individual units ("neurons")

The edges of the graph are links on which the neurons can send data to each other

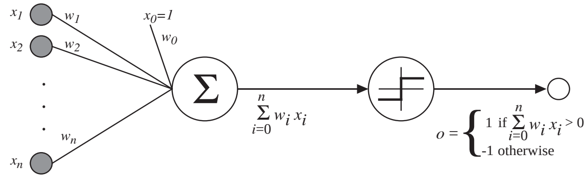
The edges have *weights*, which multiply the data that is sent

Learning = choosing weight values for all edges in the graph

Sometimes learning means adding/deleting nodes

In the vast majority of applications, the graph is acyclic and directed, and the the learning algorithm is *backpropagation*

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 will add a fixed component $x_0 = 1$ to all the instances and use simpler vector notation:

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

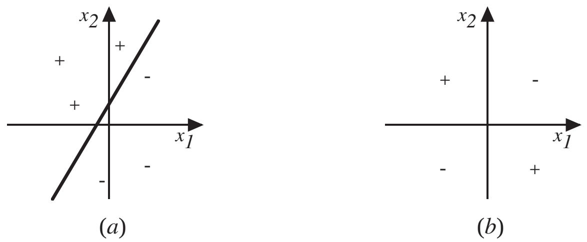
When to Consider Neural Networks

- ♢ Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- ♢ Output is discrete or real valued, or a *vector of values*
- ♢ Possibly noisy data
- ♢ Training time is unimportant
- ♢ Form of target function is unknown
- ♢ Human readability of result is unimportant

Examples:

Speech phoneme recognition [Waibel]
Speech synthesis [Nettalk]
Image classification [Kanade, Baluja, Rowley]
Automatic driving [Pomerleau]
Financial prediction

Decision Surface of a Perceptron



Represents some useful functions:

What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

But some functions not representable (E.g. not linearly separable)

Therefore, we will want networks of these...

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

η is small constant (e.g., 0.1) called *learning rate*

Can prove it will converge if training data is linearly separable and η sufficiently small

Fails to converge (oscillates) if the data is not linearly separable

Linear Units

We would like to have an algorithm that converges when the training examples are not separable too

Ideally, it would converge to a "best fit" or "minimum error" on the training data

Idea: consider just a *linear unit*, with no threshold:

$o = w_0 + w_1 x_1 + \cdots + w_n x_n$

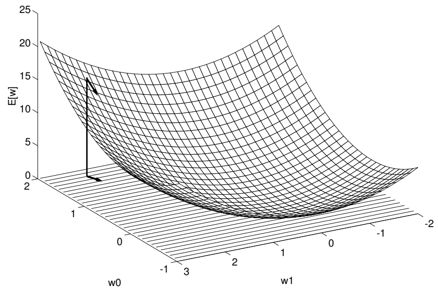
Goal: learn w_is that minimize the squared error

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

where D is set of training examples

Hill-climbing search for a good set of weights!

Gradient Descent



Gradient: $\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$

Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}] \text{ i.e. } \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

Gradient Descent

`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. η is the learning rate (e.g., .05).

1. Initialize each w_i to some small random value
2. Until the termination condition is met, Do:
   (a) Initialize each Δw_i to zero.
   (b) For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do:
      i. Input the instance $\vec{x}$ to the unit and compute the output o
      ii. For each linear unit weight w_i, Do
         $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
   (c) For each linear unit weight w_i, Do:
      $w_i \leftarrow w_i + \Delta w_i$

Summary

Perceptron training rule guaranteed to succeed if:

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent:

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent: repeat until satisifed:

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent: repeat until satsified:

For each training example d in D

1. Compute the gradient $\nabla E_d[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad\qquad E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

Building networks of individual units

Perceptrons have very simple decision surfaces

If we connect them into networks, the error surface for the network is not differentiable (because of the hard threshold)

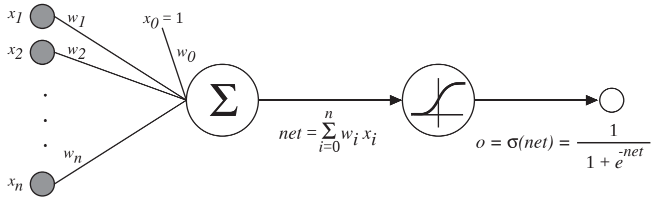
So we cannot apply gradient descent to find a good set of weights...

Networks of linear units are not satsifactory either (why?)

We would like a "soft" threshold!

Nicer math, and closer to biological neurons...

Sigmoid Unit



$\sigma(x)$ is the sigmoid function

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

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

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_d (t_d - o_d) \left(- \frac{\partial o_d}{\partial w_i} \right) \\
&= - \sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_d} \frac{\partial net_d}{\partial w_i}
\end{aligned}$$

Where (see figure)

$$net_d = \sum_{i=0}^{n} w_i x_i$$

Error Gradient for a Sigmoid Unit (2)

But we know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1-o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

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

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

   $$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{hk} \delta_k$$

4. Update each network weight w_{ij}

   $$w_{ij} \leftarrow w_{ij} + \eta \delta_j x_{ij}$$

   x_{ij} is the input from unit i into unit j (so for the output neurons, the x's are the signals received from the hidden layer neurons)

Why this algorithm?

For the output units, this is just like the jupddate for a single neuron

The only difference is that now the error function for the whole network is defined over all the outputs:

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

where t_{kd} and o_{kd} are the target and output values associated with the kth output unit and dth training example

For the hidden units, we have to compute how much they influence the overall error

But they only influence the error of the units immediately downstream from them!

The rest is a matter of applying the chain rule...

Expressiveness of ANNs

- Every boolean function can be represented by a network with single hidden layer, but might require exponential (in number of inputs) hidden units
- Every bounded continuous function can be approximated with arbitrarily small error, by a network with one, sufficiently large 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].

Inductive bias is roughly *smooth interpolation between points*

Convergence of Backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Can be much worse than global minimum
- There can be MANY local minima (Auer et al, 1997)

Partial solution: train multiple nets with different inital weights

Restarting is a standard trick in hill-climbing algorithms

More tricks:

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses
- *Make sure the units start with different weights, to break symmetry!*

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs (not only two layers)
- In theory it will find a local, not necessarily global error minimum, but in practice, it often works well (can run multiple times)
- Minimizes error over *training* examples
  Will it generalize well to subsequent examples?
  See the overfitting issue...
- Training can take thousands of iterations $\rightarrow$ VERY SLOW!
  But using network after training is very fast

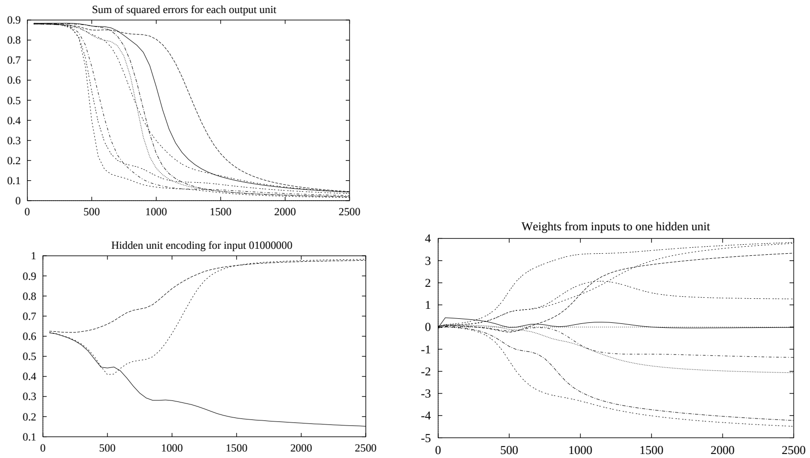
Example

A target function:

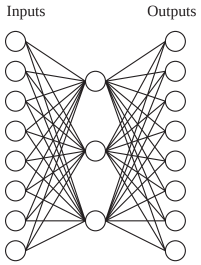
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned??

Evolution during Training



Learning Hidden Layer Representations



Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Overfitting in ANNs