Artificial Neural Networks
(based on T. MItchell Chapter 4)

Connectionist Models

Artificial Neural Networks (ANN/NN): general framework for learning a function (real-valued, discrete-valued, vector-valued) from examples.

Initially, the field developed from the analogy with biological neural networks

1.1 Biological motivation

In humans, biological neural networks:

- Neuron switching time ˜ .001 second
- Number of neurons ˜ 10^{10}
- Connections per neuron ˜ 10^{4-5}
- Scene recognition time ˜ .1 second
- It does not seem that 100 inference steps are sufficient

→ much parallel computation

Although analogy with biological NN provided the original impetus for ANN, the field developed independently of these, just as abstract models of (parallel) computation.

Efforts along the biological sgnificance of NN models remain and are now a very strong component of computational neuroscience.

1.2 Properties of artificial neural nets (ANN's)

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

1.3 When to use ANN?

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data

*based on Mitchell's notes

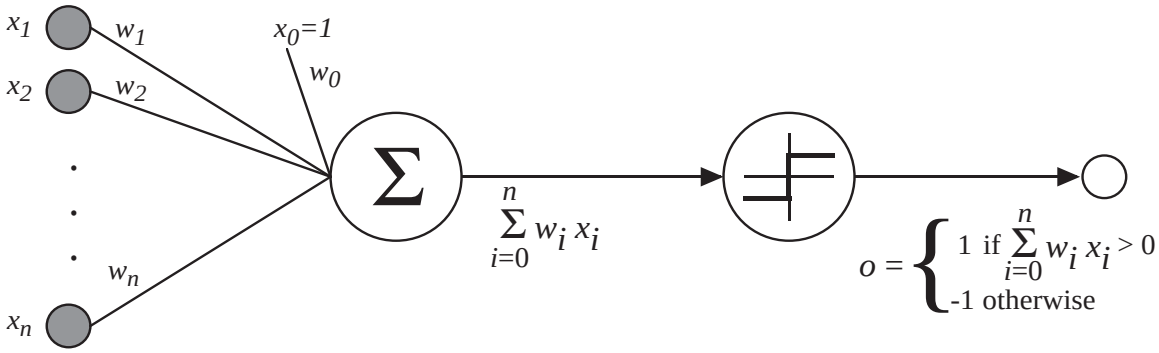


Figure 1: A single perceptron

- Form of target function is unknown
- Human readability of result is NOT important (!!!!)

Remark 1 *The last point is* **very important***:*

- *The weights in the ANN* ***encode*** *the knowledge captured by the network.*
  - *This type of knowledge is NOT for "human consumption".*
  - *It is difficult for people to:*
    - *understand exactly what how/what the network has learned,*
    - *which are the important attributes,*
    - *express the concept in some linguistic description.*

Remark 2 *In recent years sgnificant effort has been going on in extracting rules from the NN, but such steps are* **in addition** *to the NN and not part of it! NN tend to be* **black boxes***: in goes the example, out comes its corresponding output.*

NN have been used successfully in

- scene recognition and understanding;
- speech recognition;
- various classification tasks in high dimensional domains (e.g. financial, stock market prediction).

2 The Perceptron

The first model of ANN was the perceptron (Rosenblatt 1959, The perceptron: A probabilistic model for information storage and organization in the brain, *Psycholgical Review*, 65, 386-408.) The history of the perceptron is very interesting! Its first model is actually displayed at the Smithoninan!

Fact 1 *The preceptron takes a vector-valued input, calculates a linear combination of these and outputs a 1 if the result is greater than some threshold or -1 otherwise (see figure 1).*

Given the inputs $x_1, \ldots, x_n$,

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

In vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2}$$

Remark 3 *In terms of our previously used vocabulary: the hypothesis space for the perceptron is*

$$H = \{\vec{w} | \vec{w} \in \Re^{(n+1)}\}$$

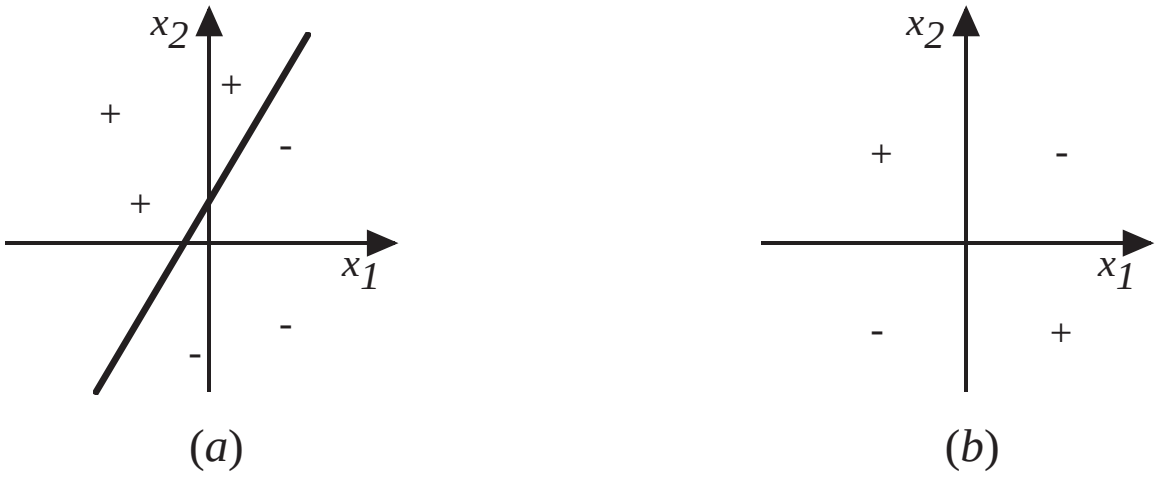


Figure 2: The decision surface for the perceptron

2.1 The Decision Surface generated by a Perceptron

Fact 2 *The perceptron can represent many useful functions: logical* **AND, OR, NAND(¬ AND), NOR (¬ OR)**.

⟶ *This means that, in fact, the perceptron (alone, or with several levels) can represent* **every** *boolean function, because any such function can be represented in terms of the boolean primitives* **AND, OR, NOT, NAND, NOR**.

For example,

- *a two-input, one-output perceptron, with the weights* $w_0 = -0.8, w_1 = w_2 = 0.5$ *represent* $g(x_1, x_2) = AND(x_1, x_2)$
- *the same perceptron can represent* $OR(x_1, x_2)$ *if the weight* $w_0 = -0.3$

Fact 3 *However, some functions are not representable: for example the logical* **XOR**. *In fact, we have the following result:*

- **If the training set for the perceptron is linearly separable, then a learning procedure based on the perceptron training rule will converge in a finite number of steps, in the sense that the perceptron will correcly classify all the examples.**
- *When the training set is not linearly separable, a network of perceptrons is necessary.*

Training the Perceptron

The Perceptron Training Rule: The idea behind the perceptron training rule is to devise a procedure such that when a positive example (true output is +1) is missclassified, the weights must be altered in order to increase the value of $\vec{w} \cdot \vec{x}$:

1. If the component x_i of $\vec{x} > 0$ then w_i, the corresponding component of w must be increased as well;
2. If $x_i < 0$, w_i must be decreased.

This behavior is captured by the following 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: $o = sgn(\vec{w} \cdot \vec{x})$
- η is small constant (e.g., 0.1) called *learning rate*

One can prove that the perceptron rule converges if

- if training data is linearly separable
- and η sufficiently small

MATLAB code which implements this rule:

```
function [w, iterations, e]=per(x, y, eta, error, i0)
%% [w, iterations, e] = delta(x,y, eta, error, i0)
%% implements the basic perceptron   rule;
%% x : data ; y : true output; eta: learning rate;
%% error : desired approximation error;
%% i0: threshold on the number of epochs
%% (iterations through the whole data set

[r, c]=size(x);
 w=rand(1,c+1);
 iterations=0;
e=error;
while e >= error &&  iterations <=i0
 iterations=iterations+1;
 wrong=0;
 for i=1:r,
    if sum(w .* [x(i,:),1])<0,
        out=-1;
    else
        out=1;
    end
     dw=eta*(y(i)-out)*[x(i,:),1];
     w=w+dw;
 end
%% compute current error
for i=1:r,
    if y(i)*(sum(w .* [x(i,:),1])) < 0
        wrong=wrong+1;
    end
end
e=wrong/r;
end
```

The files `toydatax.mat` and `toydatay.mat` contain some small training sets. The file `heart.txt` contains a more realistic data set.

3 Gradient Descent Training

To overcome the limitation of the perceptron rule for data which are not linearly separable other rules can be designed:

The **gradient descent rule** is derived from the minimization a measure of the training error. More precisely, to understand, consider simpler *linear unit*, where

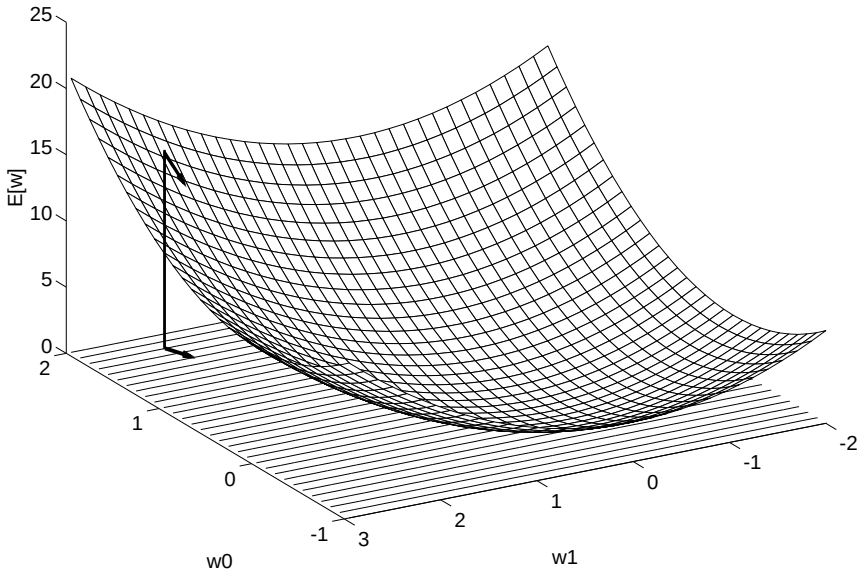
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i's 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.

Note that since o is a function of $\vec{w}$ the error, E is a function of $\vec{w}$.



To derive the gradient descent rule we need to consider the partial derivatives of E with respect to the components of $\vec{w}$, or the gradient

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

Question: Why is the gradient important?

Answer: Because *when considered as a vector in the weight space, the gradient specifies the direction that produces the steepest (fastest) increase in E.*

This means that its negative gives the direction of the steepest decrease!

Therefore, for the steepest decrease, the training rule must be

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

or, writing this by components,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{3}$$

Then, in principle, the weight update rule is

$$\vec{w} \leftarrow \vec{w} + \nabla E(\vec{w})$$

To construct a practical algorithm we must carry out the computation of $\frac{\partial E}{\partial w_i}$ as follows:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 take\ \partial\ under\ sum \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d})
\end{aligned} \tag{4}$$

Therefore,

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d}) \tag{5}$$

where $x_{i,d}$ is the ith component of the training example $\vec{x_d}$.
Substituting (5) into (3) we obtain **the weight update rule for the gradient descent**:

$$\nabla w_i = \eta \sum_{d \in D} (t_d - o_d) x_{i,d} \tag{6}$$

Gradient Descent Algorithm

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).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_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

$$w_i \leftarrow w_i + \Delta w_i$$

3.1 Summary

- *The perceptron training rule guaranteed to succeed if*
  - The training examples are linearly separable;
  - The learning rate η is sufficiently small.
- **By contrast, the linear unit training rule that uses gradient descent is:**
  - guaranteed to converge to hypothesis with minimum squared error
  - given sufficiently small learning rate η
  - even when training data contains noise, and
  - even when training data not separable by H

4 Incremental (Stochastic) Gradient Descent

Idea.

Approximate the gradient descent by updating the weights **incrementally**, after calculating the error for **each** individual example.

Therefore, instead of a **batch mode**, Gradient Descent, in which the steps are:

Do until satisfied

---

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

---

use an **incremental mode** Gradient Descent with steps:

Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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

is replaced by

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

Very Important Fact 1 *A very important result states that the* Incremental Gradient Descent *can approximate the* Batch Gradient Descent *arbitrarily closely if η (the learning rate) is made small enough.*

The Incremental Gradient Descent algorithm is then similar to the gradient descent except that we take $\nabla E_d(\vec{w})$ instead of $\nabla E_D(\vec{w})$ to obtain:

Incremental Gradient Descent Algorithm

INCREMENTAL-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).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_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 $o = \vec{w} \cdot \vec{x}$
    - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \eta (t - o) x_i \tag{7}$$

4.1 The Gradient Descent versus The Incremental Stochastic Gradient Descent

- **ERROR**:
  - In Gradient Descent the error is summed over all examples before updating the weights;
  - In stochastic Incremental Gradient Descent weights are updated by **each** example;
- **COMPUTATIONAL ASPECTS:** The computations required depend on the learning rate η and the computation required by each update step (basically, η determines the number of steps);
  - In the Gradient Descent summing over all examples at each step requires more computation per step, but because it actually uses the true gradient, η needs not be very small;
- **LOCAL MINIMUM**: Stochastic Gradient Descent can avoid the local minima for $\nabla E(\vec{w})$ by using $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$.

The equation

$$\Delta w_i = \eta(t - o)x_i$$

is known as the *delta rule*, or LMS (least mean squares), Adaline, Widrow-Hoff.

It looks similar to the perceptron rule, and indeed it is, with the difference that for the perceptron,

$$o = \mathbf{sgn}(\vec{w} \cdot \vec{x})$$

while for the gradient descent is

$$o = \vec{w} \cdot \vec{x}$$

The *delta* rule can also be used to train a thresholded perceptron:
Let

$$o = \vec{w} \cdot \vec{x}$$

and

$$o' = sgn(\vec{w} \cdot \vec{x})$$

To train a perceptron to fit target values ± 1 for o' the same target values can be used for o and the *delta* rule.

Since $o' = \pm 1$ then of course $o = sgn(o') = \pm 1$ and $o = 1\ or\ -1$ exactly for the values for which $o' = 1,\ or\ -1$.

However, the weights which minimize the error in o will not necessarily minimize the error in o'.

4.2 The Perceptron Rule vs Delta Rule

- **UPDATES**
  - The perceptron rule updates weights based on the error in the thresholded perceptron output;
  - The *delta rule* updates weights based on the **true magnitude of the perceptron output** (unthresholded).
- **Convergence Properties**
  1. For a set of linearly separable data, and small learning rate, the **perceptron rule converges after a finite number of steps** to a hypothesis that perfectly classifies the training data.
  2. The delta rule converges **asymptotically**, that is it can require an unbounded number of steps, to the hypothesis with minimum error, and it does this **regardless of whether the training data is linearly separable or not**.

Other algorithms for solving this type of problems

- **Linear Programming (LP)**: simultaneous inequalities ($\vec{w} \cdot \vec{x} > 0$ for positive examples, $\vec{w} \cdot \vec{x} \leq 0$ for negative examples.
- **Support vector machines (SVM)**: simultaneous inequalities with max. generalization (for linearly separable data).