Machine Learning — Week 3

Date: 27th, 29th January, 2004
Scribe: Rajashree Paul, Wendy-Yang Wang, Ming Zhou

Outline

Entropy

Entropy of V: H(V) = -?$P(V=v_i)\log_2 P(V=v_i)$

= # of bits needed to obtain full information

Entropy ˜ measure of uncertainty

Example:

Say, there are 2 classes: class1 and class2 with their probabilities P_1 and P_2.

$H = P_1(-\log_2 P_1) + P_2(-\log_2 P_2)$

If $P_1=0$ and $P_2=1$

H= - (0.log0 + 1.log1) = 0

If $P_1 = ½$ and $P_2 = ½$

H= -(½ $\log_2$ ½ +½ $\log_2$ ½) = 1

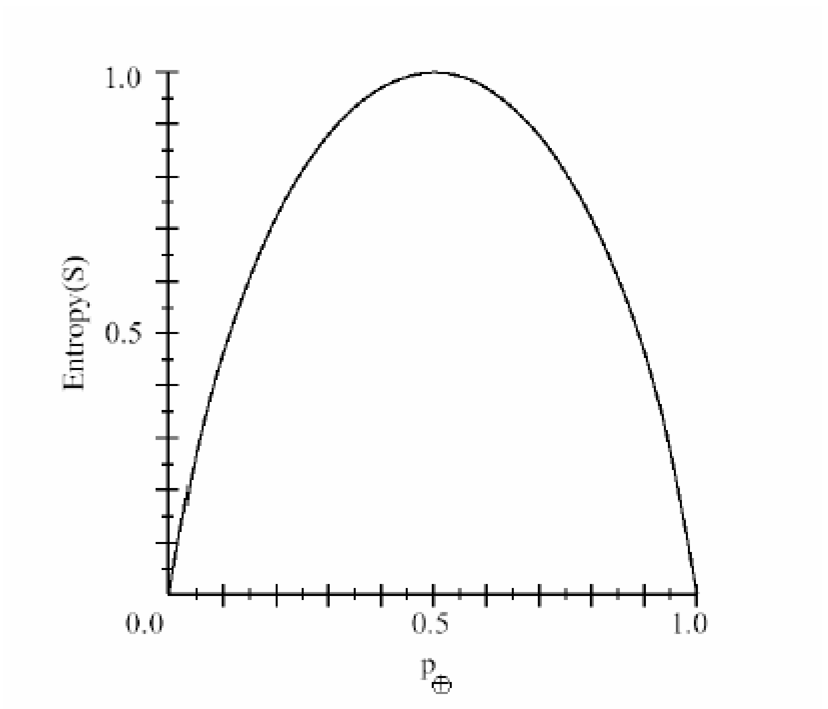


Figure 1 : P(V) vs H(V)

Examples:

Fair Coin: H(½ , ½) = 1 bit

Biased Coin: H(1/100, 99/100) = -1/100 $\log_2$(1/100) – 99/100 $\log_2$(99/100) = 0.08 bits

As P(V) → 1

Information of actual outcome → 0

H(0, 1) = H(1, 0) = 0 bits i.e. No Uncertainty left in source.

The **logarithmic function** is as follows:

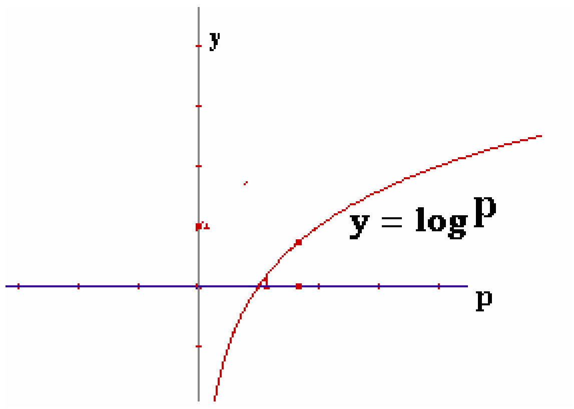


Figure 2: Logarithmic function

In the interval of 0 to 1, P decreases linearly and -log(P) decreases at a slower rate.

So, frequent messages get shorter codes. For example, in speed dial of mobile phone we put the person's number whom we call frequently.
Based on this, Samuel Morse invented **Morse Code** which is a binary code of {**-**, **.**} and used to represent English alphabets and numbers in Information Theory. The main idea behind is to give simplest or shortest codes to the most frequently used letters.
He counted the number of letters in sets of printers' type and found that E is most commonly used letter(frequency=12,000) followed by T (9,000) followed by A,I,N,O,S and so on with Z having lowest frequency (200).

The figures he came up with were:

12,000	**E**	**2,500**	**F**
9,000	**T**	**2,000**	**W, Y**
8,000	**A, I, N, O, S**	**1,700**	**G, P**
6,400	**H**	**1,600**	**B**
6,200	**R**	**1,200**	**V**
4,400	**D**	**800**	**K**
4,000	**L**	**500**	**Q**
3,400	**U**	**400**	**J, X**
3,400	**C, M**	**200**	**Z**

So he used shortest codes for E and T.

MORSE CODE ALPHABETS:

The International Morse code characters are:

A	.-	**N**	-.	**0**	-----
B	-...	**O**	---	**1**	.----
C	-.-.	**P**	.--.	**2**	..---
D	-..	**Q**	--.-	**3**	...--
E	.	**R**	.-.	**4**	....-
F	..-.	**S**	...	**5**	.....
G	--.	**T**	-	**6**	-....
H	....	**U**	..-	**7**	--...
I	..	**V**	...-	**8**	---..
J	.---	**W**	.--	**9**	----.
K	-.-	**X**	-..-	**Fullstop**	.-.-.-
L	.-..	**Y**	-.--	**Comma**	--..--
M	--	**Z**	--..	**Query**	..--..

Later Shannon proposed a closed form of optimal number of bits.

Shannon's Theorem tells that in an optimal encoding, ceiling value of ($-\log_2 Pi$) bits are assigned to Class 'i'. This gives a lower bound result of expected message length.

However, since number of bits must have an integer value it can be greater than Entropy.

If probability is given, Entropy and message length can be obtained. Similarly, if message length is given, probability can be assigned.

Message Length, $L = -\log_2 P$
$2^{-L} = P$
Thus, given a distribution of message length we can get a probability distribution.

For example, in the Morse Code Alphabet
Message Length, L of the letter 'E' is 1 and that of letter 'A' is 2
Therefore, probability, P of letter 'E' = $2^{-1} = ½$
And probability of letter 'A' = $2^{-2} = ¼$

Entropy of collection of Examples:

Training data provides estimates of probabilities (when exact probabilities are not given)

Given a training set with { p positive, n negative} examples:

$H(p/(p+n), n/(p+n)) = -p/(p+n) \log_2 (p/(p+n)) - n/(p+n) \log_2(n/(p+n))$

Eg: wrt 12 restaurant examples, S:
$p = n = ½$ => $H(½ , ½) = 1$ bit
So, 1 bit of information is needed to classify a randomly picked example from S.

So, Entropy of a collection S, $E(S) = -? P_i \log_2 P_i$

What happens to Entropy when S is split by attributes?

Say, S is split by attribute A

$E_{now} =$

$$\sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where Values(A) is the set of all possible values for attribute A, and S_v is the subset of S for which attribute A has value v (i.e., $S_v = \{ s ? S \mid A(s) = v\}$).

Information Gained by testing attribute A:

♦ Gain(A) = H (p/ (p+n), n/(p+n)) – Uncert(A)

Where, Uncert (A) = Uncertainty remaining after getting information on attribute A

Assume A divides training set E into E_1, E_2,…………E_v, where A has v distinct values.
E i has {$p_i^{(A)}$ positive, $n_i^{(A)}$ negative}examples.

♦ Entropy of each E_i is:

$$H\left(\frac{p_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}, \frac{n_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}\right)$$

♦ Uncert (A) = Expected information content
=> weigh contribution of each E_i

Uncert (A) =

$$\sum_{i=1}^{v} \frac{p_i^{(A)}+n_i^{(A)}}{p+n} \; H\left(\frac{p_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}, \frac{n_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}\right)$$

Therefore, Information Gain of a attribute A relative to a collection of examples S is defined as :

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

$$\equiv H\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \sum_{i=1}^{v} \frac{p_i^{(A)}+n_i^{(A)}}{p+n} \; H\left(\frac{p_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}, \frac{n_i^{(A)}}{p_i^{(A)}+n_i^{(A)}}\right)$$

This value represents:

♦ Reduction in entropy caused by partitioning the collection of examples, S on this attribute, A.

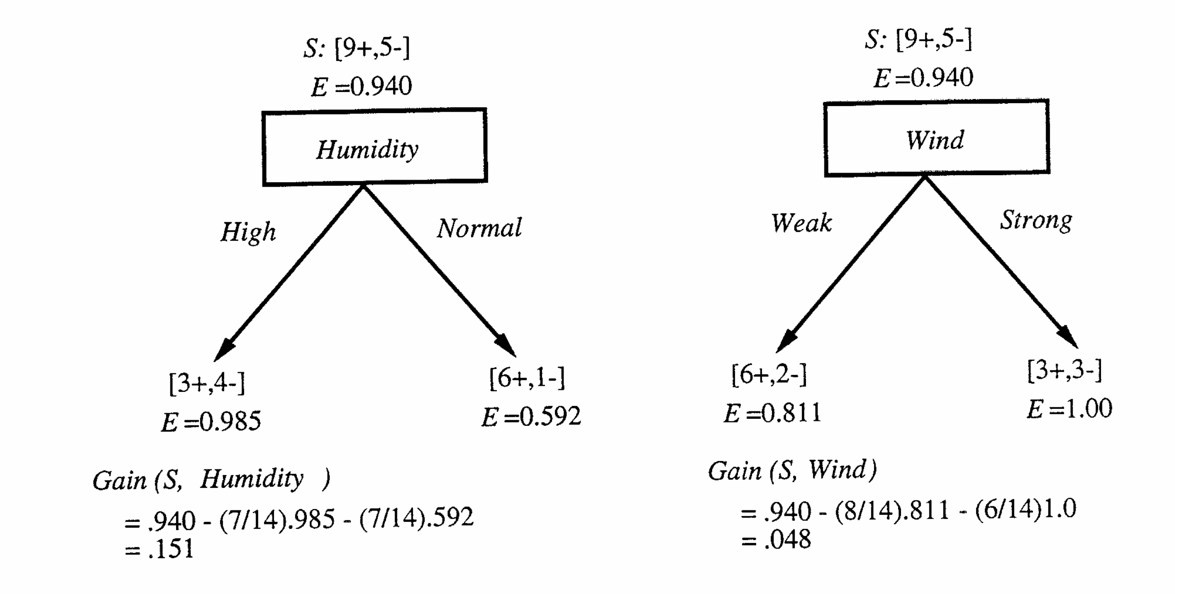
♦ The number of bits saved when encoding the target value of an arbitrary member of S, by knowing the value of attribute A.

Which attribute is the best classifier?

Information gain provides a precise measure used by ID3 to select the best attribute at each step in growing the tree.

So the aim is to find the attribute which maximizes the Information Gain.

Greedy approach → Split on attribute that most reduces entropy (uncertainty) of class, over training examples that reach there.



For the above example,
Given an initial collection S of 9 positive and 5 negative examples,[9+, 5-], sorting these by their humidity produces collection of [3+, 4-] (Humidity = High) and [6+,1-] (Humidity = Normal)
'Humidity' provides greater information gain than 'Wind'. So 'Humidity' is better classifier attribute compared to 'Wind'.

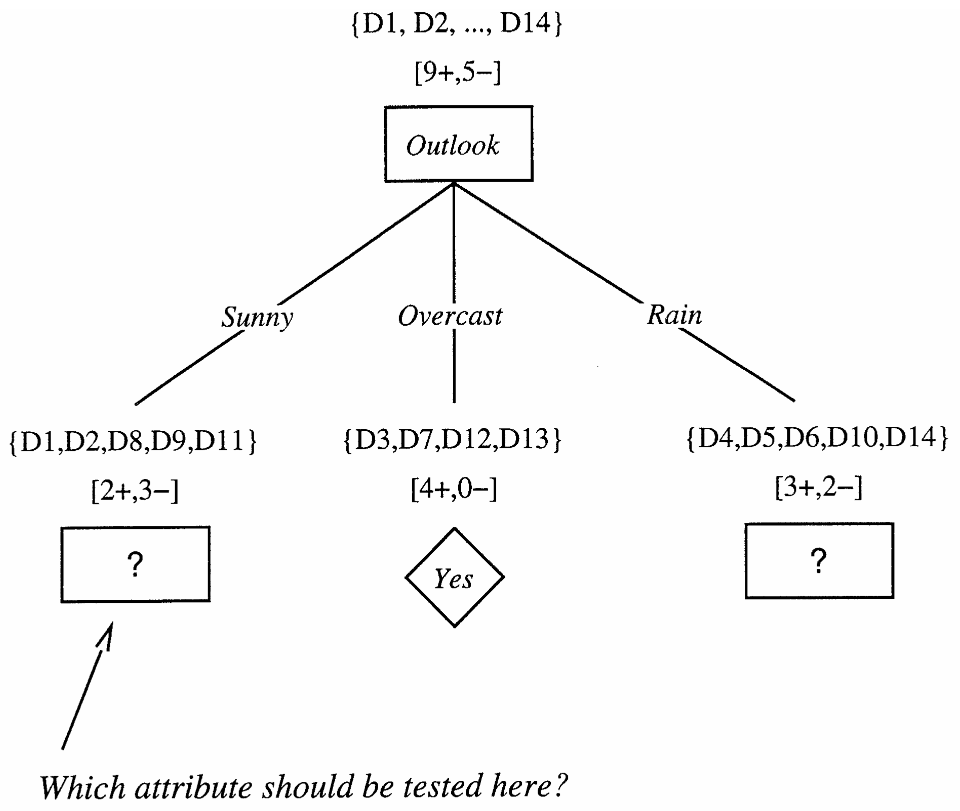
The process of selecting a new attribute for partitioning as the tree is built:

Let us consider the following example (Table 3.2 from Tom M Mitchell's book):

Task: Learn f(outlook, temperature, humidity, wind) ? { Yes, No}

Gain(S, outlook) = 0.246
Gain(S, humidity) = 0.151
Gain(S, wind) = 0.048
Gain(S, temperature) =0.029

Outlook attribute provides the best prediction of the target attribute so it is chosen as the decision attribute for the root node.



$S_{sunny} = \{D1,D2,D8,D9,D11\}$

$Gain\ (S_{sunny}, Humidity) = .970 - (3/5)\ 0.0 - (2/5)\ 0.0 = .970$

$Gain\ (S_{sunny}, Temperature) = .970 - (2/5)\ 0.0 - (2/5)\ 1.0 - (1/5)\ 0.0 = .570$

$Gain\ (S_{sunny}, Wind) = .970 - (2/5)\ 1.0 - (3/5)\ .918 = .019$

So, 'Humidity' is selected as the best predictive attribute at the left most node of the tree shown above.

Alternative Attribute Selection Heuristic: Gain Ratio

→ Information gain has the disadvantage that it prefers attributes with large number of values that split the data into many small, pure subsets, e.g. patient ID number, name, date, etc.

→Need alternative metric that discounts gain in these cases.

⇨ Quinlan's (1986) gain ratio is one approach.

→ First measure the amount of information provided by an attribute that is not specific to the category, i.e. the entropy of the data with respect to the values of an attribute.

$$SplitInfo(S, A) = \sum_{i=1}^{c} -\frac{|S_i|}{|S|}\log_2\left(\frac{|S_i|}{|S|}\right)$$

where *Si* is the subset of the examples *S* that have the *i*-th value for attribute *A*. The more uniformly examples are distributed among the values of an attribute, the higher its *SplitInfo* is.

→ *GainRatio* then uses *SplitInfo* to discourage selecting such features.

$$GainRatio(S, A) = \frac{Gain(S, A)}{SplitInfo(S, A)}$$

Hypothesis Space in Decision Tree Induction:

- Conducts a search of the space of decision Trees, which can represent all possible finite discrete functions relative to the given attributes.

- Maintains a single discrete hypothesis consistent with the data, so there is no way to determine how many other decision trees are consistent with the data, or to create useful instance queries that resolve competing hypotheses.

- Performs Hill-climbing search so may find locally optimal solution.

- Guaranteed to find a tree that fits any noise-free training set but it may not be the smallest.

- Performs batch learning – bases each decision on all remaining training examples. Algorithm can be modified to terminate early to avoid fitting noisy data.

Inductive Bias

Given:
→ Concept learning algorithm L
→ Instance space X
→ Training examples D_c ={x, c(x)}, labeled by (unknown) target concept c

Let:

→ $L(D_c)$ be classifier returned by L after training on data D_c
→ $L(x_i, D_c)$ be assigned to instance x_i by $L(D_c)$

Definition:

The **Inductive Bias** of L is any minimal set of assertions B such that for all target function c, corresponding training example D_c

$$\boxed{(B \wedge D_c) \quad \models \quad \forall x_i \in X \;\; c(x_i) = L(x_i, D_c)}$$

Where A¦ B means "A logically entails B"

Inductive Bias in C4.5:

C4.5 has a preference bias and not restriction bias (as in case of Candidate-Elimination algorithm).

- Prefers shorter trees over the longer trees in the space of possible trees
- Prefers trees that place high information gain attributes close to the root

Why prefer shorter trees?

→ **Occam's razor:** Prefer the simplest (shortest) hypothesis that fits the data.

- Argument in favor:

  → Fewer short hypotheses than long hypotheses
    - ⇨ a short hypothesis that fits data unlikely to be coincidence
    - ⇨ a long hypothesis that fits data might be coincidence

- Argument opposed:

  → There exists many ways to define small sets of hypotheses
    - ⇨ E.g.: all trees with prime number of nodes whose attributes all begin with "Z"

  → What is so special about small sets based on size of hypothesis?

<u>Avoid Overfitting: Reduced-Error Pruning</u>

A hypothesis overfits the training examples if some other hypothesis that fits the training example less well but actually performs better over the entire distribution of instances (including instances beyond the training set).

Reduced-Error Pruning:

→ Split data into ***Training*** and ***Validation*** Set.

Algorithm:

Do until further pruning is harmful:

- ⇨ Evaluate impact on *Validation* set of pruning each possible node (plus those below it)
- ⇨ Greedily remove the node that most improves accuracy on *Validation* set.

→ Produces small version of accurate subtree

→ Drawback: Limited data set

Comments of Learner

- ♦ Hypothesis space is complete.
  - ⇨ contains target function
- ♦ No back tracking
  - ⇨ fast
  - ⇨ local minima
- ♦ Statistically based search choices
  - ⇨ robust to noisy data
- ♦ Inductive bias
  - ⇨ prefer shortest tree

How to know that the hypothesis classifier is GOOD?

- ➢ Trust the learner
- ➢ Hypothesis looks good to the user: User Acceptable
- ➢ Test the learner using Validation set: Cross fold training and testing

<u>Issues in Design of Decision Tree Learner</u>

✓ What attributes to split on?

- ✓ When to stop?
- ✓ Should tree be pruned?
- ✓ How to evaluate classifier (decision tree learner)?

Using Decision Tree Classifier:

Requirements:

Instances presented by Attribute-Value pairs

→ E.g.: “bar = yes”, “size = large”, “type = French”, “temp = 82.6”
(Boolean, Discrete, Nominal, Continuous)

Can handle:

→ Disjunctive descriptions
→Errors in training data
→Missing attribute values in data

Our focus:

→ Target function output is discrete (Decision Tree also works for continuous outputs)

Examples:

→ Equipment or medical diagnosis
→ Credit risk analysis
→ Modeling calendar scheduling preferences

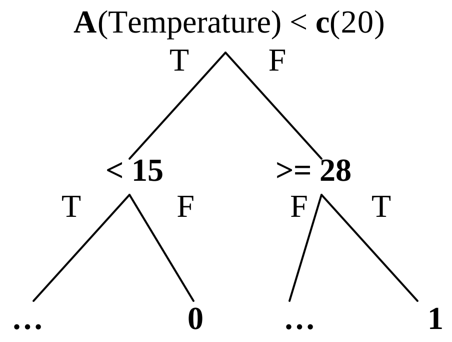
<u>More for Decision Tree Learning</u>

Decision Tree for continuous space

For instances with continuous(numeric) attributes, normally there are two methods.

1) discretize the real-valued attributes into ranges such as *big, medium and small.* Then deal them as the discrete values.
2) we can use **thresholds (cut)** on the features to split the nodes(As shown in fig A):
   Splitting the nodes based on thresholds of the form $A < c$ that partition the examples into those with $A < c$ and those with $A >= c$. The information gain of such splits are easily calculated and compared to splits on discrete features in order to select the best split.

The cutting value can be determined by various methods: 1). c can be one of the values from the training attribute A 2) it can be median, quartiles, … of the ordered values of A.



Temperature = {1, 2, …8, 10… 14, **15**, 17,**20**,…28, 35}

Fig A Decision tree for continuous attributes

Alternatives to Decision Trees

Due to the property of tree structure, there exists redundancy in the learnt decision trees. Two alternatives to deal with such problem are provided here.

➢ Decision Lists

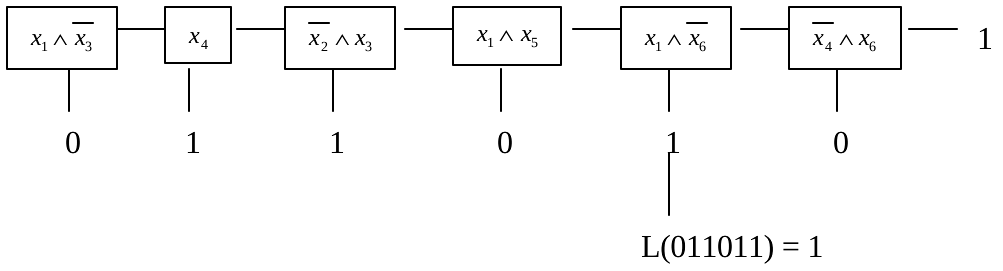
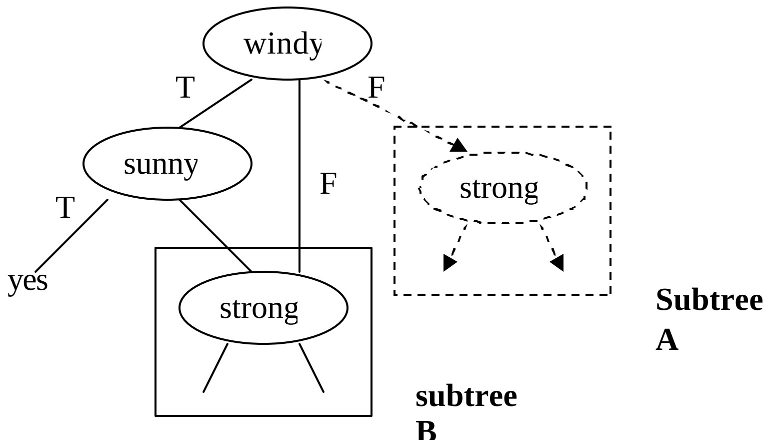

$x_1 \wedge \overline{x_3}$
x_4
$\overline{x_2} \wedge x_3$
$x_1 \wedge x_5$
$x_1 \wedge \overline{x_6}$
$\overline{x_4} \wedge x_6$
1
0
1
1
0
1
0
L(011011) = 1

Fig B Decision list

We could consider decision list to be an extended "If…Then…Elseif…." rule. As shown in Fig B, A 2-decision list and the path followed by an input. Evaluation starts at the leftmost item and continues to the right until the first condition is satisfied, at which point the binary value below becomes the final result of the evaluation.

➢ Decision Diagram



As shown in the left diagram, suppose subtree A and subtree B are the same in the original decision tree, then subtree A can be eliminated, and make the original directed edge point to subtree B. This make a decision diagram with equivalent functionality.

Artificial Neural Networks(ANNs)

<u>Overview of Artificial Neural Network</u>

Artificial Neural Network(ANNs) is a system that attempts to model the **highly massive parallel and distributed processing** of the human brain. ANNs are composed of multiple layers of simple processing elements called neurons, with weighted links interconnecting the neurons together. Learning is accomplished by adjusting these weights to cause the overall network to output appropriate results.

➢ Analogy to the Brain

Consider: 1) The speed at which the brain recognizes images;
Only 0.1 second for human to recognize a scene.

2) How many neurons populating a brain;
Human brain has 10^{11} neurons, each of which connected to 10^4 others.

3) The speed at which a single neuron transmits signals.
The neuron switching time is 0.001 second.

Appealing fact: human can make complex decisions in a surprisingly short time.

✧ Biological Neuron

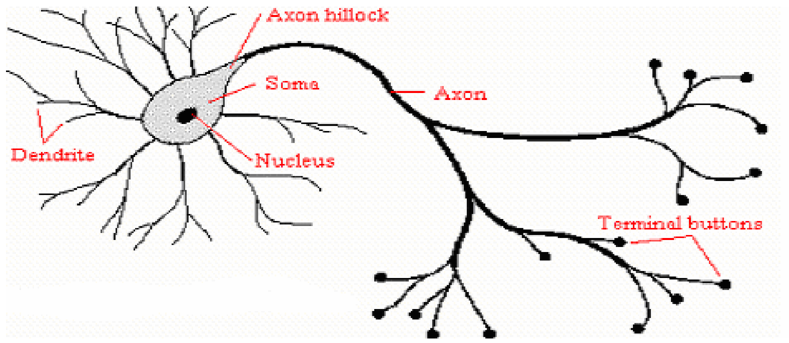


Fig A. Schematic of biological neuron

✧ Simple Model of a Neuron

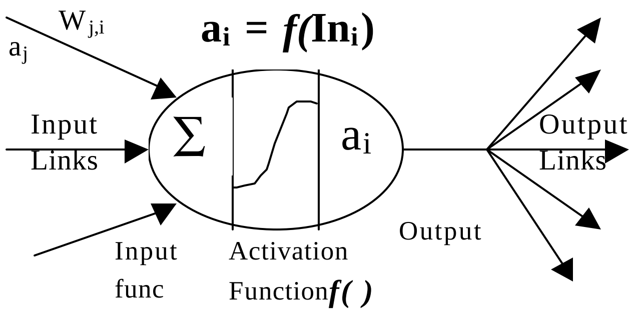

$W_{j,i}$
a_j
$\mathbf{a_i} = \boldsymbol{f}(\mathbf{In_i})$
Input Links
Σ
a_i
Output Links
Output
Input func
Activation Function $\boldsymbol{f}(\ )$

Fig B Schematic of the artificial neuron

✧ Set of input links from other units
✧ Set of output links to other units
✧ An activation level:

Input function Σ: Sum of inputs, get **Ini**
Activation function $\mathbf{a_i} = \boldsymbol{f}(\mathbf{In_i})$[1] : Linear or non- linear transformation.

➢ Two Views of Neural Networks

✧ Study and model biological learning processes
eg: Design Computer Modeled after brain. [Von Neumann, 1985] But to build the Connection Machine simulating such a high parallel computing is beyond current hardware capability.

✧ **Mathematical representation of nonlinear systems (Our Focus)**
Lots of **Simple** Computational Unit → Massively parallel implementation to

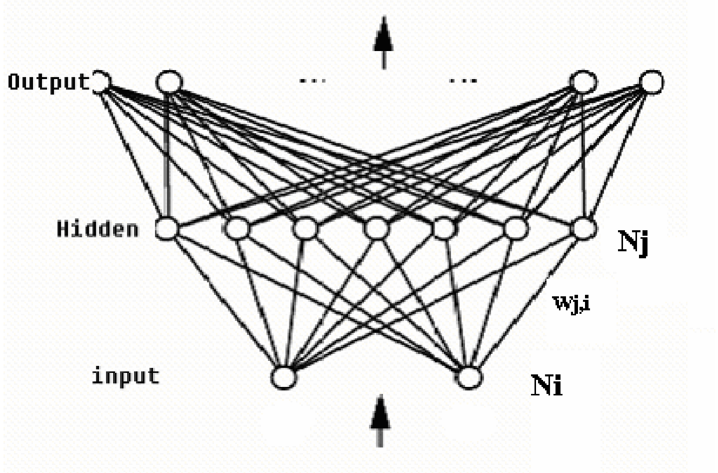
1 ***f( ),*** which is also called squashing function, can be of many forms.

realize **complicated** task

Construction of Neural Networks

- A task easy for humans but hard for engineers…
  High parallel and distributed processing, time and cost consuming.

- Components of ANN
  - Many neuron-like threshold switching units (Nj)
  - Many weighted interconnections among units($W_{j,i}$)? R
  - Layers of network, normally three: Input, Hidden and Output.



Note: Normally, in ANN learning, Architecture of Nodes and connection are fixed.
Learning is automatically tuning and finding the best combination of weights

Problems Suitable for ANN

- Input instances are represented by high dimensional, discrete or real-valued pairs.
- The output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
- The training examples may contain errors.
- The form of target function is unknown.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required..
- The ability of humans to understand the learned target function is not important.

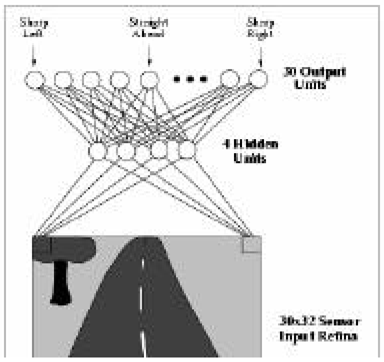
Applications of ANN

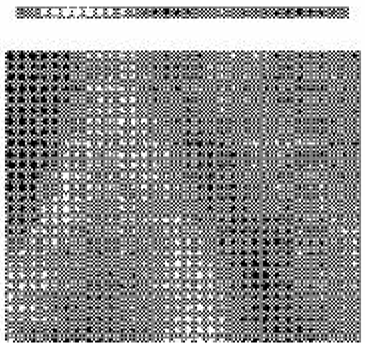
Most effective method for interpret noisy, complex sensor input data, such as inputs from cameras and microphones.

- Control
  Drive cars, Control plants, Pronunciation: NETtalk (mapping text to phonemes)
- Recognize/Classify
  Handwritten characters, Spoken words, Images(eg: faces), Credit risks
- Predict
  Market forecasting, Trend analysis

Prototypical Example — ALVINN

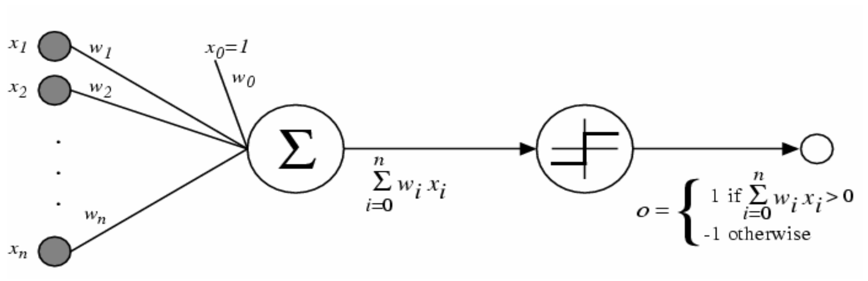
The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle driving at speeds up to 70 miles per hour.





Perceptron

A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, and outputs a 1 if the result is greater than some threshold and –1 otherwise. The following diagram illustrate a typical perceptron:



x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

$$o(x_1,\ldots,x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + \cdots + w_nx_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Vector Representation:

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

- Can eliminate threshold

  Create extra input x_0 fixed at –1.

$$o_i = step_t(\sum_{j=1}^{n} w_{j,i}x_j) = step_0(\sum_{j=0}^{n} w_{j,i}x_j)$$

where

$w_{0,i} = t$ (threshold)

$x_0 = -1$

- Can have different "squashing" function

Squashing function get its name by mapping a wide range of input to a narrow range of output. Typical Functions are: Threshold (as shown above), Linear, Logistic(Sigmoid) and Hyperbolic Tangent. Sigmoid is often used due to its smooth shape and nice property.

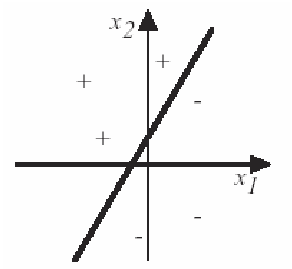
- Version Space

Learning a perceptron involves choosing values for the weights $w_0, w_1, w_2, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

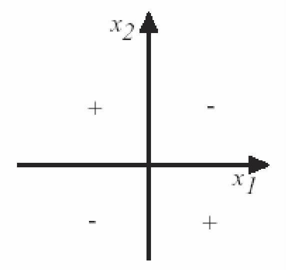
H = { < $w_0, w_1, w_2, \ldots, w_n$> | w_i ? R}

- Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances. The perceptron output a 1 for the instances lying on the one side of the hyperplane and output a –1 for the instances lying on the other side. As shown in the following figure, the equation for the hyperplane is $w_0 + x_1*w_1 + x_2*w_2 = 0$.



A single perceptron can be used to separate set of examples that is linearly separable. It can be used to represent many Boolean functions, like AND and OR. In fact, every Boolean function can be represented by some network of perceptrons only two levels deep. But there exist some set of examples, like XOR that is not linearly separable, which can not be represented by perceptrons:



As shown in the above example, no line can be found to separate the positive examples from the negative examples. Such target function can not be represented by perceptrons. Some example Boolean function will be discuss in Appendix A.

Perceptron Training Rule

➢ Why train a perceptron ?

The target function is not know. Given a set of training instances, train the perceptron to approximate the target function. To be precise, the task is to determine a weight vector that causes the perceptron to produce the correct +1 or –1 output for each of the given training examples.

Two algorithms are available to solve this learning problem: the perceptron training rule and the delta rule. These two algorithms are guaranteed to converge to some acceptable hypotheses.

➢ Perceptron training Rule

The goal of the algorithms is to find an acceptable weight vector. It begins with random weights, then iteratively apply the perceptron to the training example, modifying the perceptron weights whenever it misclassifies an example. The process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. The weight is modified as follows :

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

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., .1) called *learning rate*

❖ **Idea:**

- After each labeled instance < x, t >, Err = t – o (x) where t = c(x) is the target value and o(x) is the perceptron output
- The idea is to move the weight in appropriate direction to make Err → 0. If Err > 0 , then we need to increase o(x), else we need to decrease o(x). So to modify the weight, we should have a term (t – o)
- Input i contributes $w_i * x_i$ to the total input. If $x_i > 0$, increasing w_i will increase o(x); If $x_i < 0$, increasing w_i will decrease o(x). Thus the change of the weight should has the has a term $x_i * (t – o)$
- ? is called learning which moderate the degree to which weights are changed at each step
- Thus we have the following fomula for the algorithm:

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

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

which is the key for the perceptron training rule algorithm.
An example is provided in appendix B

➢ Correctness of Perceptron Training Rule

✧ The rule is intuitively right

Change the weight in a direction to reduce the error.

✧ Excellent Convergency[Rosenblatt 1960]

If function *f* can be represented by perceptron, then learning algorithm is guaranteed to quickly converge to *f*

✧ Converge Conditions

1) The training data is linearly separable
2) The learning rate is sufficiently small

✧ Proof

The weight space has no local minima, so given enough training example, the training algorithm will find the correct function.

✧ About learning rate

If η is too large, may overshot.
If η is too small, the algorithm may take too long

So often $\eta = \eta(k)$ which decay with the number of iterations (k).

Gradien Descent and Delta Rule

➢ Why Delta Rule Algorithm

The perceptron rule algorithm fails to converge if the training example set is not linearly separable. On the other hand, the delta rule algorithm converges toward a best fit approximation to the target concept if given non-linearly separable training examples. The key idea behind the delta rule is to use gradient descent to search the hypothesis space of possible weight vectors to find the weight that best fit the training examples.

➢ Squared Error Function

For better understanding of the delta training rule, here we consider only the unthresholded perceptron. That is to say, instead of output 1 and –1, the perceptron output the linear sum of the inputs:

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

And we have the error function of a hypothesis (weight vector) relative to the training examples is defined as :

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

in which D is the set of training examples.

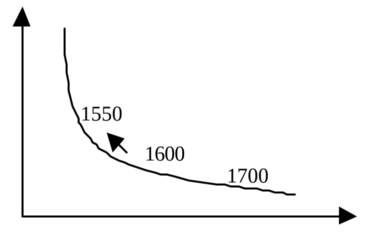
➢ Gradient

The gradient of Error function E with respect to weight vector $\vec{w}$ is defined as:

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

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. Thus the negative of the gradient vector gives the direction of the steepest decrease, which is the direction we desire.

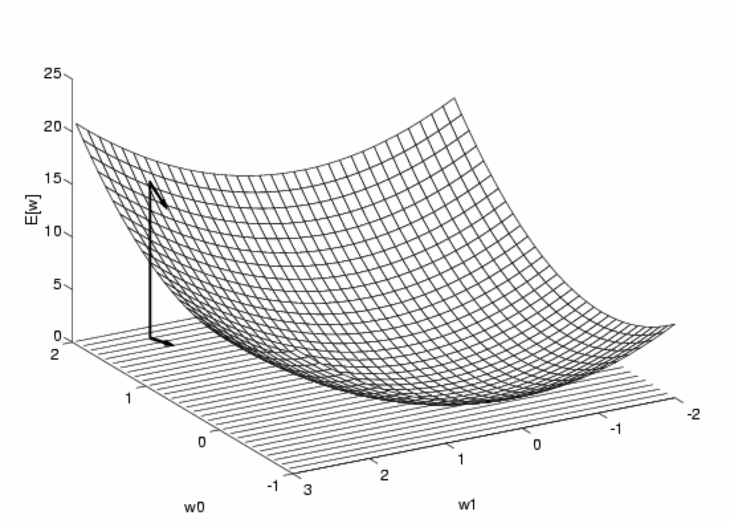
✧ Intuition of Gradient Descent



The intuition behind the gradient descent algorithm is rather simple. Suppose we are looking for a house numbered 1415, and we are in 1600. When look at the two neighbors, one is 1550, and the other is 1700. Then intuitively we will choose to go in the 1550 direction because it lead to a decrease in the number, which is the direction we desire. The same is with gradient descent algorithm. Our goal is to decrease the error function, and we know that the negative vector of gradient leads to the decrease of the error, so we go the direction defined by the negative vector of gradient.

✧ Graph of Gradient Descent

Here the w_0 and w_1 plane represents the entire hypothesis space, while the vertical axis indicates the error E relative to some fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0 and w_1 plane producing steepest descent along the error surface.



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:

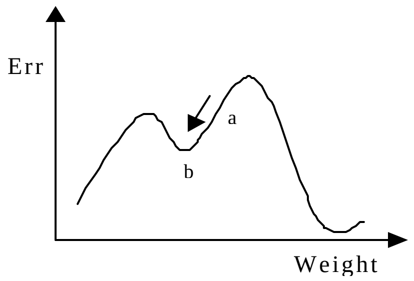
$$\Delta \vec{w} = \eta \nabla E[\vec{w}]$$

ie,

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

Gradient descent algorithm starts with an arbitrary initial weight vector, then repeatedly modifies it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface.

✧ Local Optimal



The algorithm may not be able to find an optimal hypothesis for the set of training examples. As illustrated in the above figure, suppose the initial weight vector is a. Using the gradient descent algorithm, the weight vector will change in the direction indicated by the arrow. When it reaches point b, it finds a local optimal, and output as the hypothesis for the training example. But obviously there exists point (weight vector), which produces lower error.

➢ Delta Rule

We have already known that the gradient is :

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

We are interested in the negated gradient, which would be the change to the weight vector when multiplied by the learning rate:

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

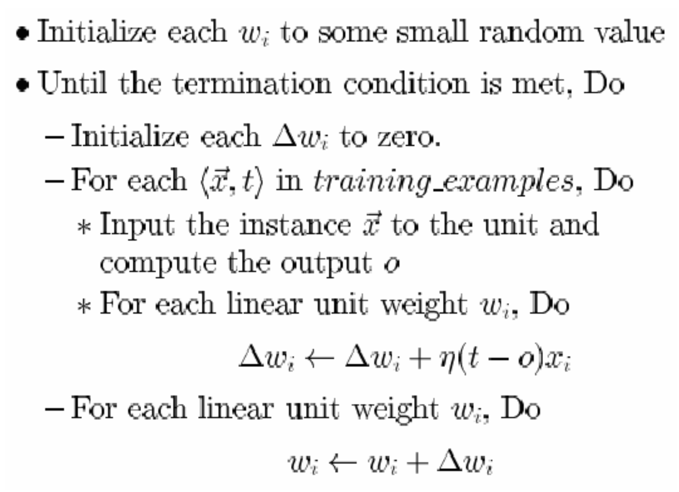
For each weight w_i, the change would be:

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

And we can calculate each partial derivative as follows:

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

Thus we can easily compute the modification to the weight vector by using the inputs, target values and output values for each training examples. The algorithm is given as follows:



```
• 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
              $w_i \leftarrow w_i + \Delta w_i$
```

<u>Perceptron Training Rule vs. Delta Rule</u>

➢ Perceptron Training

The perceptron training rule guaranteed to succeed given:

- ✧ The training examples are linearly separable
- ✧ Sufficiently small learning rate?

➢ Delta Rule

The delta rule training using gradient descent will

- ✧ guaranteed to converge to a hypothesis with minimum training error
- ✧ given sufficiently small learning rate?
- ✧ even when the training data contains noise
- ✧ even when the training data is not linearly separable.

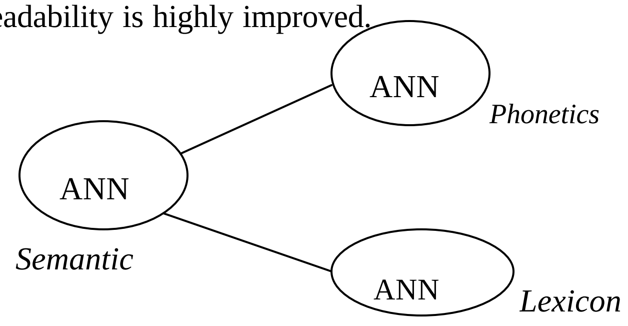
<u>Some comments</u>

➢ Training and representation power of Decision Tree and ANN.

The Version space of ANN is the set of all possible real-valued weighted vectors. It can represent both discrete and continuous function. While the space of decision trees can represent any discrete-valued function defined over discrete-valued instances. Decision tree training can have a relatively shorter training time.

➢ Readability Improvement:

Introduce the hybrid structure, as shown below. It's an application in Linguistic. Each node in the network can also represented by another ANN. In such a hybrid structure, the readability is highly improved.



Hybrid ANN

Appendix

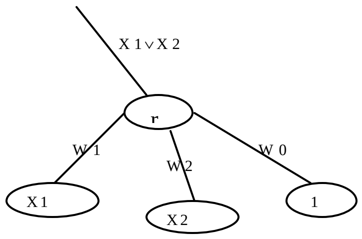
A. Function Examples

Many Boolean function can be represented if we assume Boolean values of 1 (true) and –1 (false).

❖ Function OR

X1	X2	X1∨X2
0	0	-1
0	1	1
1	0	1
1	1	1

Given the true value table, we construct a perceptron representing the Boolean function OR by setting w_1=0.5, w_2=0.5 and w_0= -0.3.

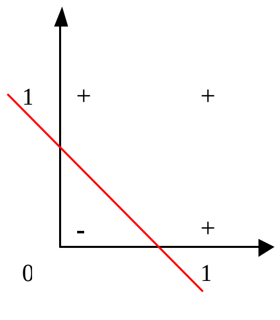


Parameters:

W1	=	0.5
W2	=	0.5
W0	=	-0.3

It is easy to verify that for $x_1 = 1$, $x_2 = 1$, the output of the perceptron is 1 because $w_0 + x_1 * w_1 + x_2 * w_2 = (-0.3) + 1* 0.5 + 1 * 0.5 = 0.7 > 0$. And it also output 1 for $x_1 = 1$, $x_2 = 0$ and $x_1 = 0$, $x_2 = 1$.

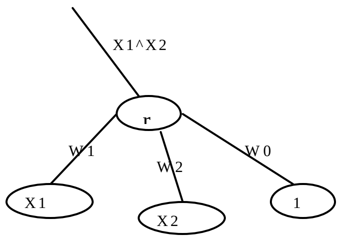
But for $x_1 =0$, $x_2 = 0$, the output of the perceptron is –1 because $w_0 + x_1 * w_1 + x_2 * w_2 = (-0.3) + 0* 0.5 + 0 * 0.5 = -0.3 < 0$. The linearly separable hyperplane is illustrated as the following figure.



❖ Function AND

X1	X2	X1^X2
0	0	-1
0	1	-1
1	0	-1
1	1	1

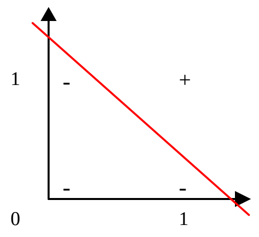
Given the true value table, we construct a perceptron representing the Boolean function AND by setting w_1=0.5, w_2=0.5 and w_0= -0.9.



Parameters:

W1	=	0.5
W2	=	0.5
W0	=	-0.9

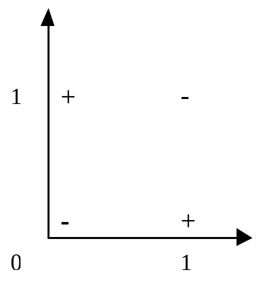
It is easy to verify that for x_1 = 1, x_2 = 1, the output of the perceptron is 1 because $w_0 + x_1*w_1+x_2*w_2$ = (-0.9) + 1* 0.5 + 1 * 0.5 = 0.1 > 0. But for all other values for example x_1 =1, x_2 = 0, the output of the perceptron is –1 because $w_0 + x_1*w_1+x_2*w_2$ = (-0.9) + 1* 0.5 + 0 * 0.5 = -0.4 < 0. The linearly separable hyperplane is illustrated as the following figure.



❖ **Function XOR**

X1	X2	X1? X2
0	0	-1
0	1	1
1	0	1
1	1	-1

As we can see, function XOR can not be done because it is not linear separable. However we adjust the values of w_1, w_2 and w_0, we can not make the positive examples on one side of the hyperplane while the negative example on the other side of the hyperplane.



It is easy to understand why function XOR can not be represented by a perceptron if we look at it from a perspective of entropy. From the true value table, we can see the entropy of the whole set of function values is 1. We can split the set of function value from the direction of x_1 or x_2. But the entropy of the resulting subsets remains to be 1. Intuitively, it make no sense to change the values of w_1, w_2 in order to find a direction to have the set of examples separated because the entropy will change not even a little bit. That is to say, function XOR can no t be represented by a perceptron.

There exists two ways to deal with nonlinear reparability: one way is to use multiple layers of perceptrons combined together to approximate more complex decision surface; the other way is to use Support Vector Machine which uses some kernel function to change the example space in order to keeping it linear separable because linear-separability has many good properties.

B. Example for perceptron training rule

Given two examples:

<1, 1, 1> and < 1, 0 , -1 >

Suppose the initial weights for the perceptron are:

w_1=0.5, w_2=0.5 and w_0= -0.4

The algorithm takes in the first training example < 1 , 1, 1 >, thus we have t(x) = 1. And $w_0 + x_1 * w_1 + x_2 * w_2 = -0.4 + 1 * 0.5 + 1 * 0.5 = 0.6 > 0$. Thus o(x) = 1. The algorithm correctly classifies the training instance, so no weight is changed.

Then the algorithm takes in the second training example < 1, 0 , -1 >. We have t(x) = -1. And $w_0 + x_1 * w_1 + x_2 * w_2 = -0.4 + 1 * 0.5 + 0 * 0.5 = 0.1 > 0$. Thus o(x) = 1. We have Err = t(x) – o(x) = -1 – 1 = -2.

? w_1= ? $(t-o)x_1 = 0.1 * (-2) * 1 = -0.2$, so $w_1 = 0.5 + (-0.2) = 0.3$

? w_2= ? $(t-o)x_2 = 0.1 * (-2) * 0 = 0$, so $w_2 = 0.5$

? w_0= ? $(t-o)x_0 = 0.1 * (-2) * 1 = -0.2$, so $w_0 = -0.4 + (-0.2) = -0.6$

For another iteration, when takes in <1,1,1>, we have $w_0 + x_1 * w_1 + x_2 * w_2 = -0.6 + 1 * 0.5 + 1 * 0.5 = 0.4 > 0$, thus o(x) = 1, correct

When takes in <1,0,-1>, we have $w_0 + x_1 * w_1 + x_2 * w_2 = -0.6 + 1 * 0.5 + 0 * 0.5 = -0.1$, thus o(x) = -1, correct.

So the algorithm classifies all the training examples correctly, so the it stop and output the learnt weight vector < w_0 ,w_1, w_2> = < -0.6, 0.3, 0.5 >