Machines Learning Part 1: Artificial Neural Netwroks

With some elements taken from: Raymond J. Mooney, Tom Mitchell and Richard Lipmann

Philippe Pasquier
Office 565 (floor 14)
pasquier@sfu.ca

SFU

Artificial Neural Networks

- Analogy to biological neural systems, the most robust learning systems we know.
- Attempt to:
  - Understand natural biological systems through computational modeling.
  - Model intelligent behavior as an “emergent” property of a large number of simple units rather than from explicitly encoded symbolic rules and algorithms.

⇨ Artificial neural networks are the paradigm of connectionist systems (connectionism vs. symbolism)

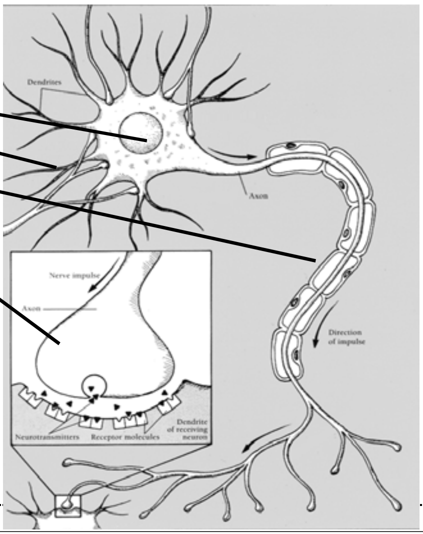
Artificial Neural Network

- While there are many types of artificial neural networks (ANN), we will focus on three kinds (in order of complexity):
  - Perceptron: Initial algorithm for learning simple neural networks (single layer) developed in the 1950’s.
  - Multilayer feedfoward networks: More complex algorithm for learning multi-layer neural networks developed in the 1980’s.
  - Self organising maps: clustering and feature similarities topological representation

⇨ We will start by looking at what they have in common: the artificial neuron

Real Neurons

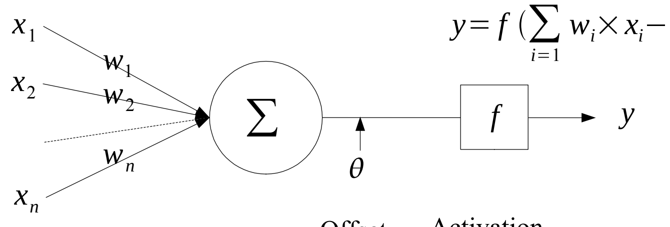
- Cell structures
  - Cell body
  - Dendrites
  - Axon
  - Synaptic terminals
- Synapses change size and strength with experience.
- Human brain has about 10^{11} neurons with an average of 10^4 connections each.



Artificial Neuron

- An artificial neuron:

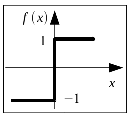
$$y = f\left(\sum_{i=1}^{n} w_i \times x_i - \theta\right)$$



Inputs Weights Sum Offset (threshold) Activation function Output

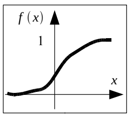
Hard Limiter:

$$f(x) = \begin{cases} 1 & if\ x > 0 \\ -1 & if\ x \le 0 \end{cases}$$



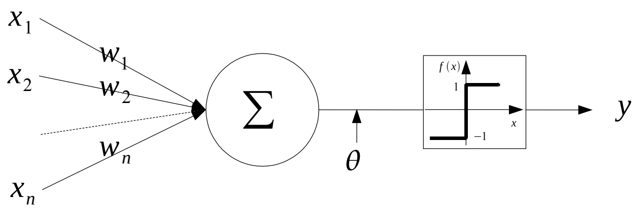
Sigmoid (logistic):

$$f(x) = \frac{1}{1+e^{x}}$$



Perceptron

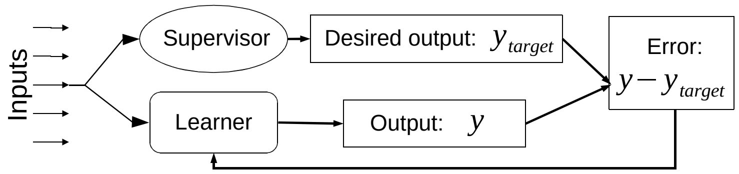
- **The Perceptron is made of one neuron with a hard limiter activation function:**



- **The Perceptron can only decide on two classes: A or B, good or bad, hot or cold, ...**
- **By using a different "squashing function" a more smooth output gain be obtained**

Perceptron

Supervised learning:



• The error being a function of the output is also a function of all the weights: learning is done by updating the weights

• The Perceptron learning rule:

$$\Delta w_i = \eta \overbrace{(y - y_{target})}^{\text{Error for that example}} x_i$$

Learning rate (e.g. 0.05) Input for that weight

- If output is correct do nothing.
- If output is too high, lower weights on active inputs
- If output is low, increase weights on active inputs (i.e. <> 0)

Perceptron Algorithm

Incremental (or stochastic) gradient descent:

1) $Set\, w_i^{t_0}$ $(1 \leq i \leq n)$ and θ to small random values

2) Present a new input instance: $x_1^t, x_2^t, \ldots, x_n^t$

3) Calculate the actual Output: $y^t = f(\sum_{i=1}^{n} w_i^t \times x_i^t - \theta)$

4) Present the desired output: y_{target}^t

5) Update the weights: $w_i^{t+1} = w_i^t + \overbrace{\eta[(y^t - y_{target}^t) \times x_i^t]}^{\Delta w_i}$

6) If Termination condition not met: go to step 2

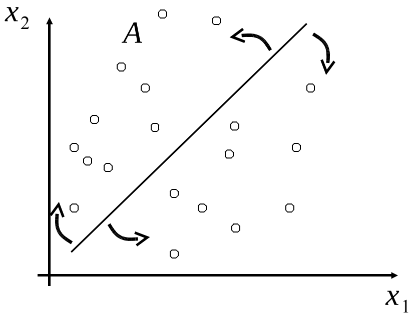
Perceptron Algorithm

- **Used for supervised learning:**
  - **One pass through the training set is called an epoch: the loop is repeated as many times as there is examples in the training set**
  - **Condition Termination:**
    - **Epochs are repeated as long as there is improvement (over a number of them)**
    - **Until all training data are correctly classified**
    - **A limit is reached:**
      - **A fixed time (e.g. 5 min)**
      - **A fixed number of epoch (e.g. 600)**
  - **The standard gradient descent update the weights only once per epoch (using the sum of the errors on all the data of the training set)**

⇨ **What can be learned by a Perceptron?**

Representational Power

- Since the Perceptron unit implements a linear (combination) function, it is searching for a linear separator that discriminates the classes.



$$w_1 x_1 + w_2 x_2 > \theta$$

$$w_2 x_2 > -w_1 x_1 + \theta$$

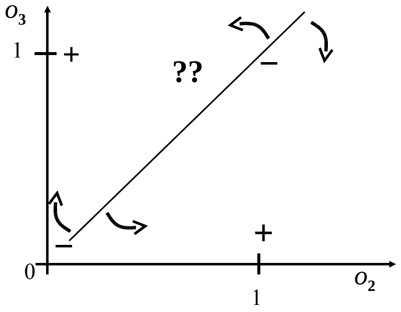
$$x_2 > -\frac{w1}{w_2} x_1 + \frac{\theta}{w_2}$$

$$x_2 = -\frac{w1}{w_2} x_1 + \frac{\theta}{w_2}$$

This is the equation of a line (or *hyperplane* in *n*-dimensional space)

Concept Perceptron Cannot Learn

- Cannot learn functions that are not linearly separable: exclusive-or, or parity function.
- But can provide a good approximation sometimes!



Perceptron: Error function

– The learning technique is called gradient descent because it going to minimise the error made by following the slope of the error curve, i.e. the gradient

– This error function has a single minima

For all examples

Error

Error function

$$E(\vec{w}) = \frac{1}{2}\sum_{d \in D}(y - y_{target})^2$$

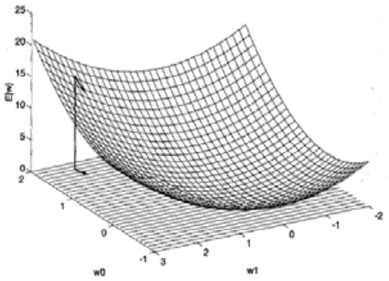
Weight vector
(hypothesis = what want want to learn)

Avoids that negative
and positive errors
annihilate each others

Gradiant Descent and the Delta Rule

- **Vector derivative are called gradient**
- **The negative of this specifies the steepest decrease in E**

The idea is to minimise the error



Hypothesis space
(all possible weight combinations)

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_{d \in D}\frac{\partial}{\partial w_i}(y - y_{target})^2$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D}(y - y_{target})(-x_{id})$$

$$\Delta w_i = \eta \sum_{d \in D}(y - y_{target})\, x_{id}$$

Parceptron: learning rules

- Different Error functions will give different learning rules (computed with the same principle of gradient descent):
- **The true gradient rule:**
  - Also called standard gradient descent
  - Update the weights according to all training examples at once

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (y - y_{target})^2$$

$$\Downarrow$$

$$\Delta w_i = \eta \sum_{d \in D} (y - y_{target})\, x_{id}$$

- **The Delta rule:**
  - Also called
    - LMS (least mean square)
    - Adaline rule
    - Widrow-Hoff rule
  - Incremental gradient descent or stochastic gradient descent
  - Update the weights after each example

$$E(\vec{w}) = \frac{1}{2} (y - y_{target})^2$$

$$\Downarrow$$

$$\Delta w_i = \eta (y - y_{target})\, x_{id}$$

Perceptron Performances

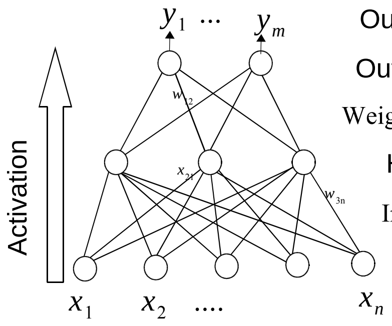
- Obviously, a system cannot learn concepts it cannot represent.
- In practice, it converges fairly quickly for linearly separable data.
- Can effectively use even incompletely converged results when only a few outliers are misclassified.
- Experimentally, the Perceptron does quite well on many benchmark data sets.
- Due to their nature, ANN are quit resistant to noise in the data: a small difference on one or several inputs does not give a big difference on the output

Perceptron Performances

- There is a plethora of theoretical results:
  - Perceptron convergence theorem: If the data is linearly separable and therefore a set of weights exist that are consistent with the data, then the Perceptron algorithm will eventually converge to a consistent set of weights.
  - Perceptron cycling theorem: If the data is not linearly separable, the Perceptron algorithm will eventually repeat a set of weights and threshold at the end of some epoch and therefore enter an infinite loop.
    - By checking for repeated weights+threshold, one can guarantee termination with either a positive or negative result.

Multi-Layer feedforwards Networks

- A typical multi-layer network consists of an input layer (not real neurons), a hidden and an output layer, each fully connected to the next, with activation being fed forward in the network
- Use Sigmoid “logistic” activation function



Outputs

Output Layer

Weight from unit i to unit j is noted w_{ji}

Hiden layer

Input from unit i into unit j is noted x_{ji}

Inputs (not real artificial neurons, but the links have weights)

Error Backpropagation Algorithm

The stochastic gradient descent version:

1) Set all weight and offset to small random values

2) Present a new input instance: $x_1^t, x_2^t, \ldots, x_n^t$

3) Calculate the actual outputs: $y_1^t, y_2^t, \ldots, y_m^t$

4) Present the desired output: $y_{target,1}^t, \ldots, y_{target,m}^t$

5) Update the weights:

$$w_{ji}^{t+1} = w_{ji}^t + \Delta w_{ji}^t \text{ , where: } \Delta w_{ji}^t = \eta\, \delta_j\, x_{ji}$$

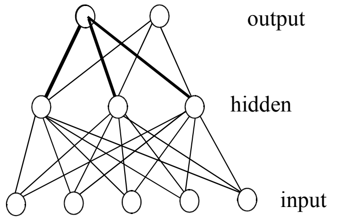
For output units k: $\delta_k = y_k(1-y_k)(y_{target,k} - y_k)$

For hidden units h: $\delta_h = y_h(1-y_h) \sum_{k \in outputs} w_{kh} \delta_k$

6) If Termination condition not met: go to step 2

Error Backpropagation: step 5.1

- **First calculate error of output units and use this to change the top layer of weights.**



Example:

Current output: y_k=0.2

Correct output: $y_{target,k}$=1.0

Error $\delta_k = y_k(1\text{–}y_k)(y_{target,k}\text{-}y_k)$

0.2(1–0.2)(1–0.2)=+0.128

Update weights for each hiden unit i:

$$w_{ki}^{t+1} = w_{ki}^t + \Delta w_{ki}^t \text{ , where: } \Delta w_{ki}^t = \eta\, \delta_k\, x_{ki}$$

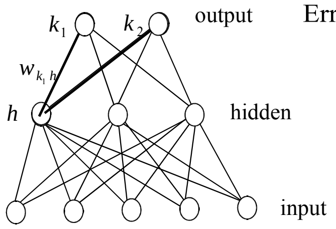
Learning rate

Activation/input received

Error detail: $\delta_k = \underbrace{y_k(1-y_k)}_{\text{Derivative of the sigmoid function}}\underbrace{(y_{target,k} - y_k)}_{\text{Usual "error"/deviation}}$

Error Backpropagation: step 5.2

- **Next calculate error for hidden units based on errors on the output units it feeds into.**



Error: $\delta_h = y_h(1-y_h) \underbrace{\sum_{k \in outputs} w_{kh}\delta_k}$

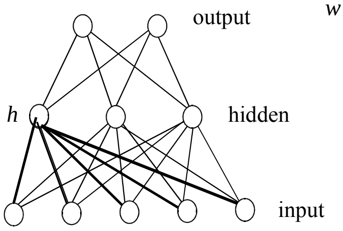
Sum of the errors of the output units (for wich we get the target values) pondered be the weights of the links

Error Backpropagation: setp 5.3

- **Finally update bottom layer of weights based on errors calculated for hidden units.**

Update weights into *h*

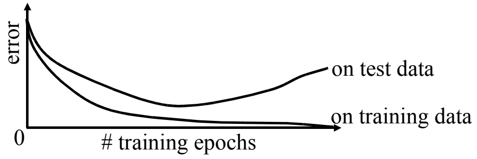
$w_{hi}^{t+1} = w_{hi}^{t} + \Delta w_{hi}^{t}$, where: $\Delta w_{hi}^{t} = \eta \delta_j x_{hi}$



Repeat 5.2 and 5.3 for every hidden units

Termination Conditions

- Various termination conditions can be used:
  - Fixed number of iteration (thousands)
  - Once the error over all the training examples falls bellow some threshold
  - Once the error on a separate training set meets some criterion
- Running too many epochs can result in over-fitting.



- **Possible solution:** keep a hold-out validation set and test accuracy on it after each sequence of 100 epoch. Stop training when additional epochs actually increase validation error.

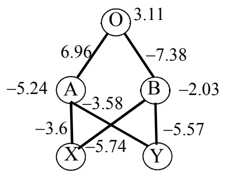
Comments on Training Algorithm

- Again, it is doing a gradient descent in the error space
- One crucial difference with the Perceptron is that the error space can have multiple local minima
- No guarantee to converge to zero training error, may converge to local optima or oscillate indefinitely.
- However, in practice, it does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required: hours or days of training for large networks.
- To avoid local-minima problems: run several trials starting with different random weights (*random restarts*).
  - Take results of trial with lowest training set error.
  - Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).

Representational Power

- Multi-layer networks can represent arbitrary functions
- The weights determine the function computed. Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer.
- Boolean functions: Any boolean function can be represented by a two-layer network with sufficient hidden units.
- Continuous functions: Any bounded continuous function can be approximated with arbitrarily small error by a two-layer network.
  - Sigmoid functions can act as a set of basis functions for composing more complex functions, like sine waves in Fourier analysis.
- Arbitrary function: Any function can be approximated to arbitrary accuracy by a three-layer network.

Example: Learned XOR Network



Hidden Unit A represents: $\neg(X \wedge Y)$
Hidden Unit B represents: $\neg(X \vee Y)$
Output O represents: $A \wedge \neg B = \neg(X \wedge Y) \wedge (X \vee Y)$
$= X \oplus Y$

Successful Applications

- Pattern recognition: speech recognition (DragonTalk), text to speech (NetTalk), handwriting recognition, face recognition (identity, orientation, ...), fraud detection, ...
- Financial Applications
  - HNC Software (eventually bought by Fair Isaac)
- Chemical Plant Control
  - Pavillion Technologies
- Automated Vehicles: ALVINN, ...
- Game Playing
  - Neurogammon

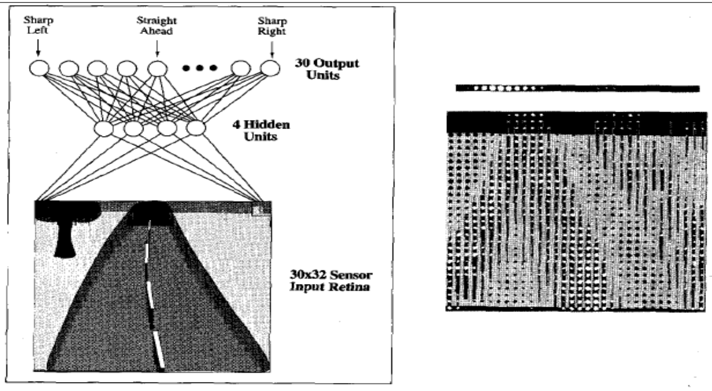
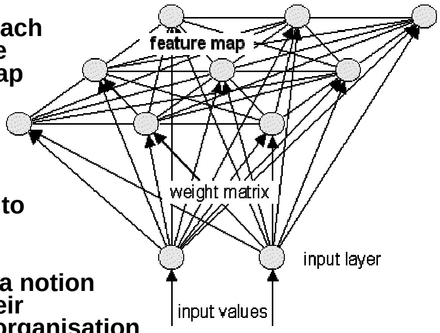


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

Self Organising Maps (Pr. Teuvo Kohonen)



- SOMs aim to associate each input vector to one of the output (neurons) on a map topologically organised so to reflect features proximity
- Unsupervised learning
- Each input is connected to every output neurons
- Output neurons are not connected together, but a notion of neighborood maps their topological (i.e. spatial) organisation to the inputs' features similarity
- While there are several variants, we present the basic version
- A SOM acts like a classifier in which the number of classes if fixed (m neurons) and are topologically disposed but there nature/features is not predetermined

Learning algorithm

1) Set all weights from n inputs node to m neurons to random values and set the initial neighborhood NE_j^0 to large values

2) Present a new input instance: $x_1^t, x_2^t, \ldots, x_n^t$

3) Calculate distance to all neurons: $d_j = \sum_{i=1}^{n} (x_i^t - w_{ji}^t)^2$

4) Select the node j^* that minimize the distance d_j

5) Update the weights of j^* and the neighborhood $NE_{j^*}^t$

$w_{ji}^{t+1} = w_{ji}^t + \eta^t (x_i^t - w_{ji}^t)$, for $j \in NE_{j^*}^t$ and $1 \leq i \leq n$

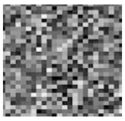
η^t is a learning rate (or gain) that decrease with time

6) Repeat steps 2-5 a fixed number of times

Example: colors

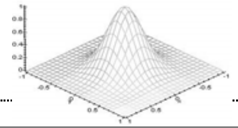
- Visualisation: each unit is associated to a pixel (or a square) and represents the last input/instance classified (i.e. a color)
- 25*25 units organised as a grid (each unit represents the weights associated with it and the elements that have been assigned to it)
- 3 inputs (Reg, Green, Blue): 3*25*25 links (i.e. Weights), here each weight represents a component (R, G or B)
- Algo:
  1. Initialise the weights, the neighborhood and the learning rate:
  2. Get an input, say green (0,6,0)
  3. Calculate the distance to each node (using Euclidian distance):
     - $d_{\text{Light green}}$ = Sqrt((0-3)^2+(6-6)^2+(0-3)^2) = 4.24
     - d_{red} = Sqrt((0-6)^2+(6-0)^2+(0-0)^2) = 8.49

     4.Light green is selected
  5. Learning: The winning weights are rewarded with becoming more like the input vector. The neighbours' weights also become more like the input vector.
  6. Decrease the size of the neighborhood and decrease $\underbrace{\text{the learning rate}}_{\text{Initially 1}}$



Random weights
=
random colors

DEMO



Remarks on SOM

- SOMs produce a low-dimensional (typically two dimensional), discretized representation of the input space of the training samples, called a map.
- The map seeks to preserve the topological properties of the input space.
- SOMs accomplish two things:
  - They reduce dimensions: in our example, the inputs were three dimensions and there were “number of input” of them and the output is only two dimensions
  - They display similarities: in our example, the similarities are obvious!

⇨ SOMs are useful for visualizing low-dimensional views of high-dimensional data

Issues in Neural Nets

- There is a lot (lot) more to be seen!
  - More efficient training methods:
    - Quickprop
    - Conjugate gradient (exploits 2nd derivative)
  - Learning the proper network architecture:
    - Grow network until able to fit data: Cascade Correlation, Upstart, ..
    - Shrink large network until unable to fit data
  - Recurrent networks that use feedback and can learn finite state machines with "backpropagation through time."
  - More biologically plausible learning algorithms based on Hebbian learning ("fire together, wire together").
- Many applications to metacreation!

?

"The only real mistake is the one from which we learn nothing."

John Powell