Machine Learning
CS-527A

Artificial Neural Networks

Burchan (bourch-khan) Bayazit
http://www.cse.wustl.edu/~bayazit/courses/cs527a/

Mailing list: cs-527a@cse.wustl.edu

Artificial Neural Networks (ANN)

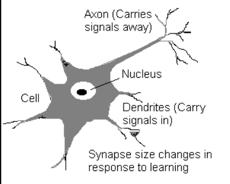
Neural network inspired by biological nervous systems, such as our brain

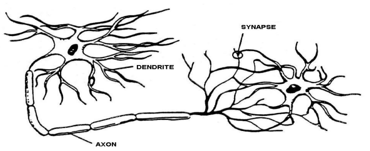
Useful for learning real-valued, discrete-valued or vector-valued functions.

Applied to problems such as interpreting visual scenes, speech recognition, learning robot control strategies.

Works well with noisy, complex sensor data such as inputs from cameras and microphones.

ANN

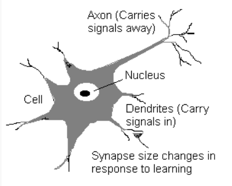




Inspiration from Neurobiology

- A neuron: many-inputs / one-output unit
- Cell Body is 5 – 10 microns in diameter

ANN



- incoming signals from other neurons determine if the neuron shall excite ("fire")
- Axon turn the processed inputs to outputs.
- Synapses are the electrochemical contact between neurons.

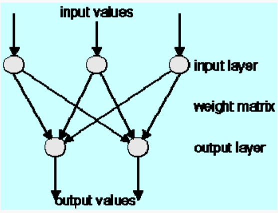
ANN

- In human brain, approximately 10^{11} neurons are densely interconnected.
- They are arranged in networks
- Each neuron connected to 10^4 others on average
- Fastest neuron switching time 10^{-3} seconds
- ANN motivation by biological neuron systems; however many features are inconsistent with biological systems.

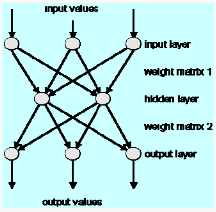
ANN – Short History

- McCulloch & Pitts (1943) are generally recognized as the designers of the first neural network
- Their ideas such as threshold and many simple units combining to give increased computational power are still in use today
- In the 50's and 60's, many researchers worked on the perceptron
- In 1969, Minsky and Papert showed that perceptrons were limited so neural network research died down for about 15 years
- In the mid 80's interest revived (Parket and LeCun)

ANN

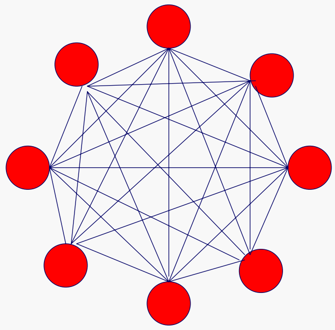


One Layer Perceptron



Two Layer Perceptron

Hopfield Network



Types Neural Network Architectures

Many kinds of structures, main distinction made between two classes:

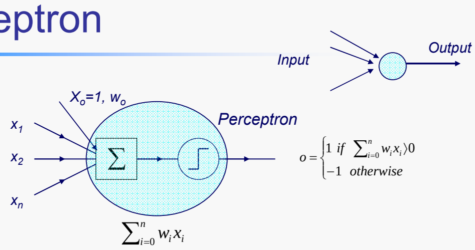
a) feed- forward (a directed acyclic graph (DAG): links are unidirectional, no cycles

- There is no internal state other than the weights.

b) recurrent: links form arbitrary topologies e.g., Hopfield Networks and Boltzmann machines

Recurrent networks: can be unstable, or oscillate, or exhibit chaotic behavior e.g., given some input values, can take a long time to compute stable output and learning is made more difficult….
However, can implement more complex agent designs and can model systems with state

Perceptron

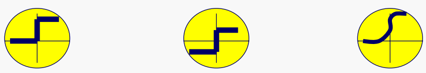


The McCullogh-Pitts model

The perceptron calculates a weighted sum of inputs and compares it to a threshold. If the sum is higher than the threshold, the output is set to 1, otherwise to -1.

Learning is finding weights w_i

g = Activation functions for units



Step function
(Linear Threshold Unit)
step(x) = 1, if x >= threshold
0, if x < threshold

Sign function
sign(x) = +1, if x >= 0
-1, if x < 0

Sigmoid function
sigmoid(x) = 1/(1+e^-x^)

Adding an extra input with activation a_0 = - 1 and weight $W_{0,j}$ = t is equivalent to having a threshold at t. This way we can always assume a 0 threshold.

Perceptron

Mathematical Representation

$$o(x_1, x_2, \ldots, x_n) = \begin{cases} 1\,if\ \ w_o + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1\,otherwise \end{cases}$$

$$o(\vec{x}) = \operatorname{sgn}(\vec{w}\cdot\vec{x}) \quad \textit{where}$$

$$\operatorname{sgn}(y) = \begin{cases} 1\,if\ \ y > 0 \\ -1\,otherwise \end{cases} \qquad H = \left\{\vec{w} \middle| \vec{w} \in R^{(n+1)}\right\}$$

Perceptron

$o(\vec{x})$ defines N-dimensional space and (N-1) dimensional plane.

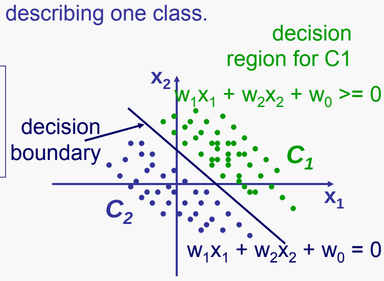
The perceptron returns 1 for data points lying on one side of the hyperplane and -1 for data points lying on the other side.

If the positive and negative examples are separated by a hyperplane, they are called linearly separable sets of examples. But it is not always the case.

Perceptron

The equation below describes a (hyper-)plane in the input space consisting of real valued m-dimensional vectors. The plane splits the input space into two regions, each of them describing one class.

$$\sum_{i=1}^{m} w_i x_i + w_0 = 0$$



Perceptron Learning

We have either (-1) or (+) as the output and inputs are either 0 or 1
There are 4 cases
- *The output is suppose to be +1 and perceptron returns +1*
- *The output is suppose to be -1 and perceptron returns -1*
- *The output is suppose to be +1 and perceptron returns -1*
- *The output is suppose to be -1 and perceptron returns +1*

If Case 1 or 2, do nothing since the perceptron returns right result
If Case 3 $w_0+w_1x_1+w_2x_2+\ldots+w_nx_n>0$ we need to increase the weights so that the left side of the equation will become greater than 0
If Case 4, the weights must be decreased

So we can use following update rule that satisfies this

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

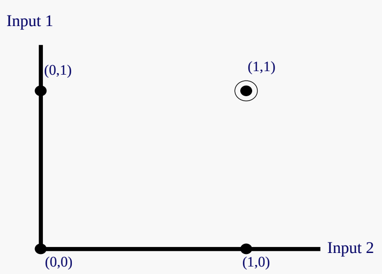
$$\Delta w_i = \eta(t-o)x_i$$

t is the target output, o is the output generated by the perceptron and η is a positive constant known as the learning rate.

Perceptron Learning

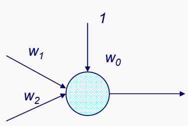
For each training data <x,t> ∈D
Find o=o(x)
update each weight $w_i=\Delta w_i+w_i$ where $\Delta w_i=(t-o)x_i$

Learning AND function



Training Data:
(0,1,0)
(0,0,0)
(1,0,0)
(1,1,1)

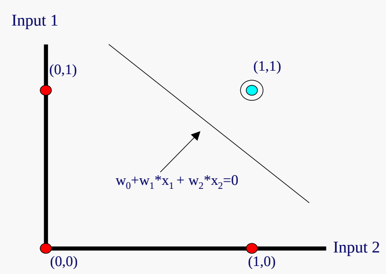
Learning AND function



w_0=-1
w_1=0.6
w_2=0.6

Learning AND function

- Output space for AND gate



Training Data:
(0,1,0)
(0,0,0)
(1,0,0)
(1,1,1)

Limitations of the Perceptron

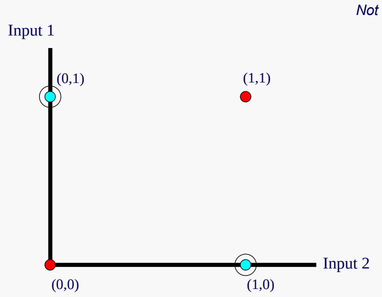
- Only binary input-output values
- Only two layers
- Separates the space linearly

Only two layers

- Minsky and Papert (1969) showed that a two-layer Perceptron cannot represent certain logical functions
- Some of these are very fundamental, in particular the exclusive or (XOR)
- Do you want coffee XOR tea?

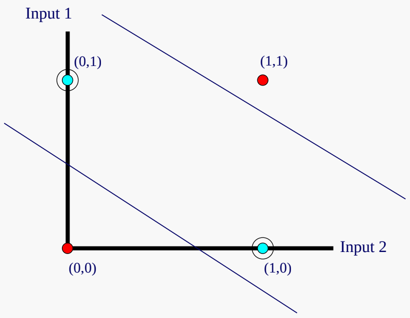
Learning XOR

Not Linearly Separable



Learning XOR

Not Linearly Separable



Solution to Linear Inseparability

- *Use another training rule (delta rule)*
- *Backpropagation*

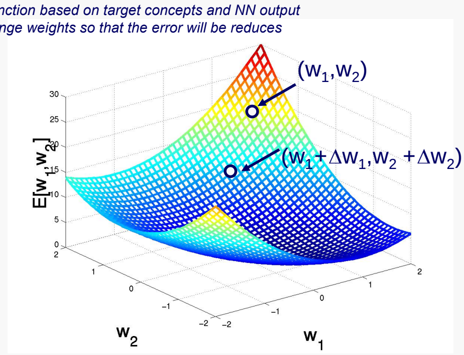
ANN

Gradient Descent and the Delta Rule

- Delta Rule designed to converge examples that are not linearly separable.
- Uses gradient descent to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

Gradient Descent

Define an error function based on target concepts and NN output
The goal is to change weights so that the error will be reduces



Gradient Descent

- Training error of a hypothesis:

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

D is the set of training examples,
T_d is the target output for training example d,
and o_d is the output of the linear unit for training example d.

How to find Δw?

Derivation of the Gradient Descent Rule

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

Direction of the steepest descent along the error surface

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \quad \text{where} \quad \Delta\vec{w} = -\eta\nabla E(\vec{w})$$

The negative sign is present as we want to go in the direction that decreases E.

For the i^{th} component:

$$w_i \leftarrow w_i + \Delta w_i \quad \text{where} \quad \Delta w = -\eta\frac{\partial E}{\partial w_i}$$

How to find Δw?

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

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_{d \in D}2(t_d - o_d)\frac{\partial(t_d - o_d)}{\partial w_i} = \sum_{d \in D}(t_d - o_d)\frac{\partial(t_d - \vec{w}\cdot\vec{x}_d)}{\partial w_i}$$

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

Where x_{id} is the single input component x_i for for training example d

Hence $$\Delta w_i = \eta\sum_{d \in D}(t_d - o_d)x_{id}$$

Gradient-Descent Algorithm

- 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 and η is the learning rate (e.g. 0.5)
- 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

      $w_i \leftarrow w_i + \Delta w_i$

Training Strategies

- Online training:
  - Update weights after each sample
- Offline (batch training):
  - Compute error over all samples
    - Then update weights
- Online training “noisy”
  - Sensitive to individual instances
  - However, may escape local minima

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

    Propagate the input forward through the network:

    1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

    Propagate the errors backward through the network:

    2. For each network output unit k, calculate its error term δ_k

    $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

    3. For each hidden unit h, calculate its error term δ_h

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

    4. Update each network weight w_{ji}

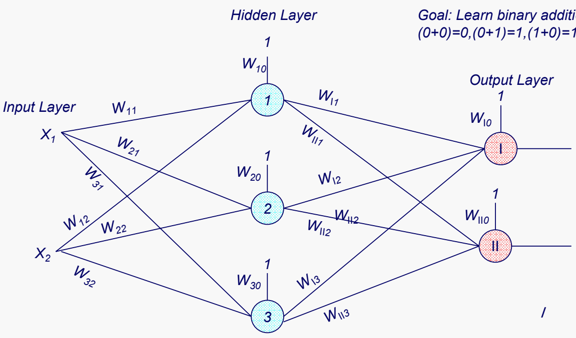
    $$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

    where

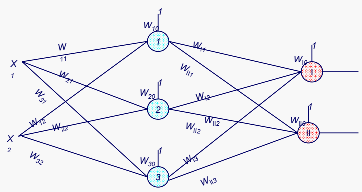
    $$\Delta w_{ji} = \eta\, \delta_j\, x_{ji}$$

Example: Learning addition

Goal: Learn binary addition: i.e.: (0+0)=0,(0+1)=1,(1+0)=1,(1+1)=10



Example: Learning addition



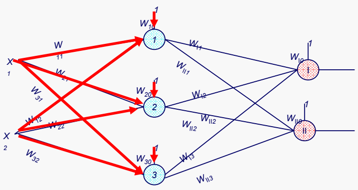
Training Data

Inputs	*Target Concept*
0,0	0,0
0,1	0,1
1,0	0,1
1,1	1,0

Activation Function

$$O(y) = \frac{1}{1+e^{-y}}$$

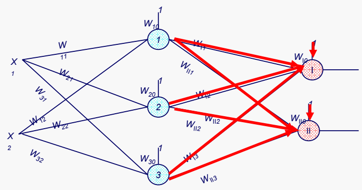
Example: Learning addition



First find the outputs O_I, O_{II}
In order to do this, propagate the inputs forward.
First find the outputs for the neurons of hidden layer

$$O_1 = O\left(\sum_i W_i X_i\right) = O(W_{10} + W_{11}X_1 + W_{12}X_2)$$
$$O_2 = O(W_{20} + W_{21}X_1 + W_{22}X_2)$$
$$O_3 = O(W_{30} + W_{31}X_1 + W_{32}X_2)$$

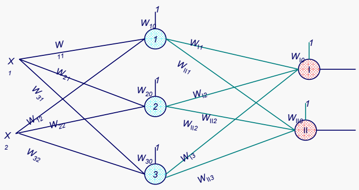
Example: Learning addition



Then find the outputs of the neurons of hidden layer

$$O_I = O(W_{I0} + W_{I1}O_1 + W_{I2}O_2 + W_{I3}O_3)$$
$$O_{II} = O(W_{II0} + W_{II1}O_1 + W_{II2}O_2 + W_{II3}O_3)$$

Example: Learning addition



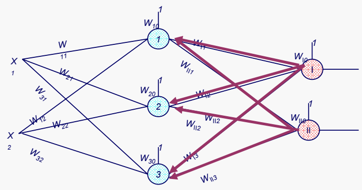
Now propagate back the errors. In order to do that first find the errors for the output layer, also update the weights between hidden layer and output layer

$$\delta_I = O_I(1 - O_I)(t_I - O_I)$$
$$\delta_{II} = O_{II}(1 - O_{II})(t_{II} - O_{II})$$

$$\Delta W_{I0} = \eta\delta_I$$
$$\Delta W_{I1} = \eta\delta_I O_1$$
$$\Delta W_{I2} = \eta\delta_I O_2$$
$$\Delta W_{I3} = \eta\delta_I O_3$$

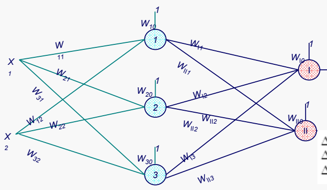
$$\Delta W_{II0} = \eta\delta_{II}$$
$$\Delta W_{II1} = \eta\delta_{II} O_1$$
$$\Delta W_{II2} = \eta\delta_{II} O_2$$
$$\Delta W_{II3} = \eta\delta_{II} O_3$$

Example: Learning addition



And backpropagate the errors to hidden layer.

Example: Learning addition



And backpropagate the errors to hidden layer.

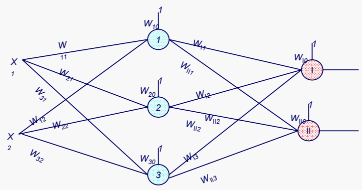
$$\delta_1 = O_1(1 - O_1)\textstyle\sum_k W_{k1}\delta_k = O_1(1 - O_1)(W_{I1}\delta_I + W_{II1}\delta_{II})$$
$$\delta_2 = O_2(1 - O_2)(W_{I2}\delta_I + W_{II2}\delta_{II})$$
$$\delta_3 = O_3(1 - O_3)(W_{I3}\delta_I + W_{II3}\delta_{II})$$

$$\Delta W_{10} = \eta\delta_1 X_0 = \eta\delta_1$$
$$\Delta W_{11} = \eta\delta_1 X_1$$
$$\Delta W_{12} = \eta\delta_1 X_2$$

$$\Delta W_{20} = \eta\delta_2 X_0$$
$$\Delta W_{21} = \eta\delta_2 X_1$$
$$\Delta W_{22} = \eta\delta_2 X_2$$

$$\Delta W_{30} = \eta\delta_3 X_0$$
$$\Delta W_{31} = \eta\delta_3 X_1$$
$$\Delta W_{32} = \eta\delta_3 X_2$$

Example: Learning addition



Finally update weights!!!!

$$W_{10} = W_{10} + \Delta W_{10}$$
$$W_{11} = W_{11} + \Delta W_{11}$$
$$W_{12} = W_{12} + \Delta W_{12}$$

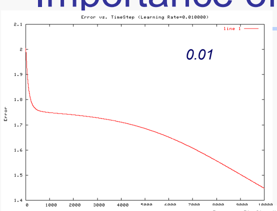
$$W_{20} = W_{20} + \Delta W_{20}$$
$$W_{21} = W_{21} + \Delta W_{21}$$
$$W_{22} = W_{22} + \Delta W_{22}$$

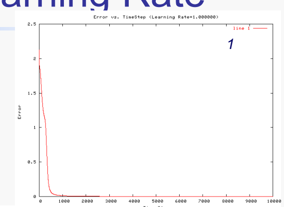
$$W_{30} = W_{30} + \Delta W_{30}$$
$$W_{31} = W_{31} + \Delta W_{31}$$
$$W_{32} = W_{32} + \Delta W_{32}$$

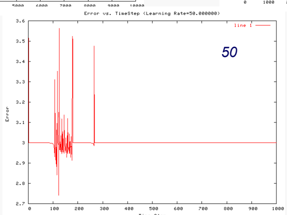
$$W_{I0} = W_{I0} + \Delta W_{I0}$$
$$W_{I1} = W_{I1} + \Delta W_{I1}$$
$$W_{I2} = W_{I2} + \Delta W_{I2}$$
$$W_{I3} = W_{I3} + \Delta W_{I3}$$

$$W_{II0} = W_{II0} + \Delta W_{II0}$$
$$W_{II1} = W_{II1} + \Delta W_{II1}$$
$$W_{II2} = W_{II2} + \Delta W_{II2}$$
$$W_{II3} = W_{II3} + \Delta W_{II3}$$

Importance of Learning Rate





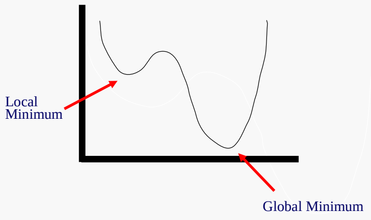


Generalization of the Backpropagation

Backpropagation Using Gradient Descent

- Advantages
  - Relatively simple implementation
  - Standard method and generally works well
- Disadvantages
  - Slow and inefficient
  - Can get stuck in local minima resulting in sub-optimal solutions

Local Minima



Alternatives To Gradient Descent

- Simulated Annealing
  - Advantages
    - *Can* guarantee optimal solution (global minimum)
  - Disadvantages
    - May be slower than gradient descent
    - Much more complicated implementation

Alternatives To Gradient Descent

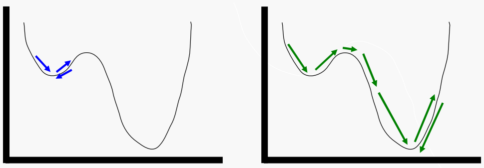
- Genetic Algorithms/Evolutionary Strategies
  - Advantages
    - Faster than simulated annealing
    - Less likely to get stuck in local minima
  - Disadvantages
    - Slower than gradient descent
    - Memory intensive for large nets

Alternatives To Gradient Descent

- Simplex Algorithm
  - Advantages
    - Similar to gradient descent but faster
    - Easy to implement
  - Disadvantages
    - Does not guarantee a global minimum

Enhancements To Gradient Descent

- Momentum
  - Adds a percentage of the last movement to the current movement



Enhancements To Gradient Descent

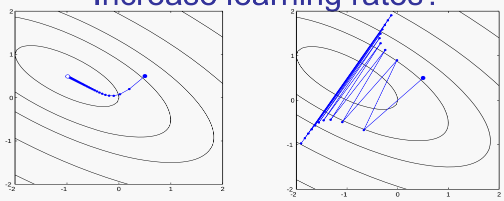
- Momentum
  - Useful to get over small bumps in the error function
  - Often finds a minimum in less steps
  - $\Delta w_{ji}(t) = -\eta * \delta_j * x_{ji} + \alpha * w_{ji}(t-1)$

Backpropagation Drawback

- Slow convergence

improve

Increase learning rates?



Bias

- Hard to characterize
- Smooth interpretation between data points

Overfitting

- Use a validation set, keep the weights for most accurate learning
- Decay weights
- Use several networks and use voting

K-fold cross validation:

1. *Divide input set to K small sets*
2. *For k=1..K*
3. *use Set_k as validation set, and the remaining as the test set*
4. *find the number of iterations i_k to optimal learning for this set*
5. *Find the average of number of iterations for all sets*
6. *Train the network with that number of iterations….*

Despite its popularity backpropagation has some disadvantages

- Learning is slow
- New learning will rapidly *overwrite* old representations, unless these are interleaved (i.e., repeated) with the new patterns
- This makes it hard to keep networks up-to-date with new information (e.g., dollar rate)
- This also makes it very implausible from as a psychological model of human memory

Good points

- Easy to use
  - Few parameters to set
  - Algorithm is easy to implement
- Can be applied to a wide range of data
- Is very popular
- Has contributed greatly to the ‘new connectionism’ (second wave)

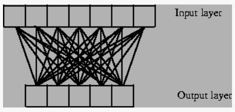
Deficiencies of BP Nets

- Learning often takes a **long time** to converge
  - Complex functions often need hundreds or thousands of epochs
- The net is essentially a **black box**
  - If may provide a desired mapping between input and output vectors (***x, y***) but does not have the information of why a particular ***x*** is mapped to a particular ***y.***
  - It thus cannot provide an intuitive (e.g., causal) explanation for the computed result.
  - This is because the hidden units and the learned weights do not have a semantics. What can be learned are operational parameters, not general, abstract knowledge of a domain
- Gradient descent approach only guarantees to reduce the total error to a **local minimum**. (***E*** may be be reduced to zero)
  - Cannot escape from the local minimum error state
  - Not every function that is represent able can be learned

- How bad: depends on the shape of the error surface. Too many valleys/wells will make it easy to be trapped in local minima
- Possible remedies:
  - Try nets with different # of hidden layers and hidden units (they may lead to different error surfaces, some might be better than others)
  - Try different initial weights (different starting points on the surface)
  - Forced escape from local minima by random perturbation (e.g., simulated annealing)

- **Generalization** is not guaranteed even if the error is reduced to zero
  - Over-fitting/over-training problem: trained net fits the training samples perfectly (E reduced to 0) but it does not give accurate outputs for inputs not in the training set
- Unlike many statistical methods, there is no theoretically well-founded way to **assess the quality** of BP learning
  - What is the confidence level one can have for a trained BP net, with the final E (which not or may not be close to zero)

Kohonen



Every neuron of the output layer is connected with every neuron of the input layer. While *learning, the closest neuron to the input data (the distance between its weights and the input vector is minimum) and its neighbors (see below) update their weights. The distance is defined as follows:*

$$d_{out} = \sum_{in} (w_{in,out} - x_{in})^2$$

The formula for the Kohonen map tends to bring the connections closer to the input data:

$$w_{in,out} = w_{in,out} + \eta(x_{in} - w_{in,out})$$

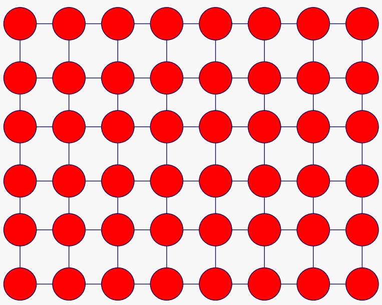
Kohonen

For each training data

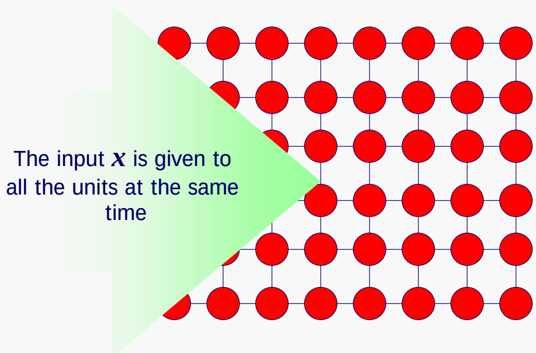
Find the winner neuron using $d_{out} = \sum_{in} (w_{in,out} - x_{in})^2$

Update the weights of the neighbors $w_{in,out} = w_{in,out} + \eta(x_{in} - w_{in,out})$

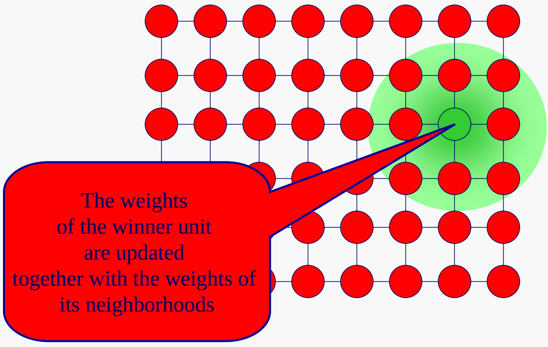
Kohonen Maps



Kohonen Maps



Kohonen Maps



NETtalk (Sejnowski & Rosenberg, 1987)
Killer Application

- The task is to learn to pronounce English text from examples.
- Training data is 1024 words from a side-by-side English/phoneme source.
- Input: 7 consecutive characters from written text presented in a moving window that scans text.
- Output: phoneme code giving the pronunciation of the letter at the center of the input window.
- Network topology: 7x29 inputs (26 chars + punctuation marks), 80 hidden units and 26 output units (phoneme code). Sigmoid units in hidden and output layer.

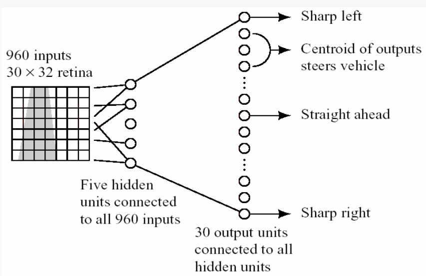
NETtalk (contd.)

- Training protocol: 95% accuracy on training set after 50 epochs of training by full gradient descent. 78% accuracy on a set-aside test set.
- Comparison against Dectalk (a rule based expert system): Dectalk performs better; it represents a decade of analysis by linguists. NETtalk learns from examples alone and was constructed with little knowledge of the task.

Steering an Automobile

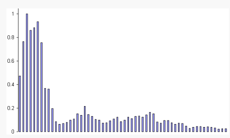
- ALVINN system [Pomerleau 1991,1993]
  - Uses Artificial Neural Network
    - Used 30*32 TV image as input (960 input node)
    - 5 Hidden node
    - 30 output node
  - Training regime: modified “on-the-fly”
    - A human driver drives the car, and his actual steering angles are taken as correct labels for the corresponding inputs.
    - Shifted and rotated images were also used for training.
  - ALVINN has driven for 120 consecutive kilometers at speeds up to 100km/h.

Steering an Automobile- ALVINN network



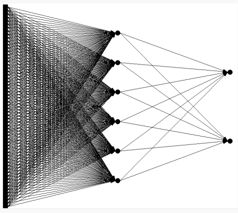
Voice Recognition

- Task: Learn to discriminate between two different voices saying “Hello”
- Data
  - Sources
    - Steve Simpson
    - David Raubenheimer
  - Format
    - Frequency distribution (60 bins)



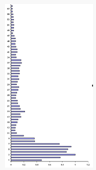
- Network architecture

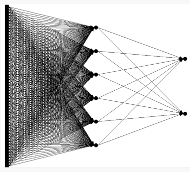
  - Feed forward network
    - 60 input (one for each frequency bin)
    - 6 hidden
    - 2 output (0-1 for “Steve”, 1-0 for “David”)



- Presenting the data

Steve





David