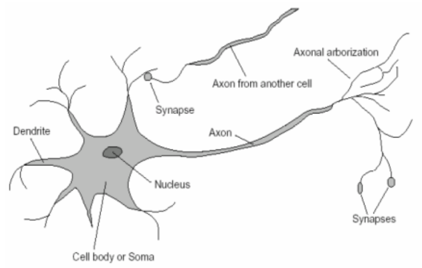
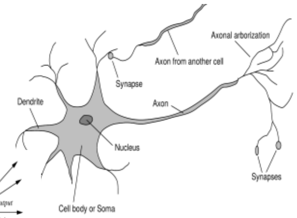
Artificial Neural Network

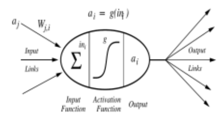
Neurons in the Brain



–

10^{11} neurons, 10^{14} connections





$$a_i = g(\sum_j W_{j,i} a_j)$$

Neural Networks

- Artificial neural network (ANN) is a machine learning approach that models human brain and consists of a number of artificial neurons.
- Each neuron in ANN receives a number of inputs.

Neural Networks

- An Artificial Neural Network is specified by:
  - **neuron model**: the information processing unit of the NN
  - **an architecture**: a set of neurons and links connecting neurons. Each link has a weight
  - **a learning algorithm**: used for training the NN by modifying the weights in order to model a particular learning task correctly on the training examples.

Perceptron

–

Introduced in the late 50s – Minsky and Papert.

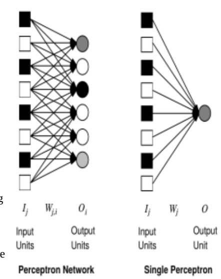
Classifies a linearly separable set of inputs.

Multi-layer perceptrons – found as a "solution" to represent nonlinearly separable functions – 1950s.

Many local minima – Perceptron convergence theorem does not apply.

1950s - Intuitive Conjecture was: There is no learning algorithm for multi-layer perceptrons.

Perceptron convergence theorem Rosenblatt 1962: Perceptron will learn to classify any linearly separable set of inputs.

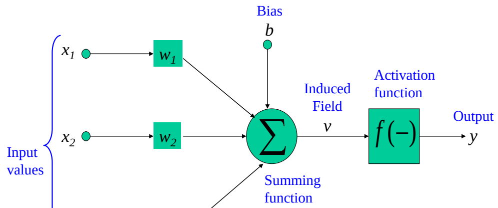


Neuron

- The neuron is the basic information processing unit of a NN. It consists of:
  1. A set of links, describing the neuron inputs, with weights $W_1, W_2, \ldots, W_m$
  2. An adder function (linear combiner) for computing the weighted sum of the inputs: (real numbers) $u = \sum_{j=1}^{m} w_j x_j$
  3. Activation function f for limiting the amplitude of the neuron output. Here 'b' denotes bias.

$$y = f(u + b)$$

The Neuron Diagram



$$y = f\left(b + \sum_{j=1}^{m} w_j x_j\right)$$

One Neuron as a Network

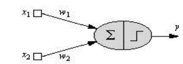


Fig1: an artificial neuron

- x1 and x2 are normalized attribute value of data.
- y is the output of the neuron , i.e the class label.
- x1 and x2 values multiplied by weight values w1 and w2 are input to the neuron
- Given that
  - w1 = 0.5 and w2 = 0.5
  - Say value of x1 is 0.3 and value of x2 is 0.8,
  - So, weighted sum is :
  - sum= w1 * x1 + w2 * x2 = 0.5 x 0.3 + 0.5 x 0.8 = 0.55

One Neuron as a Network

- The neuron receives the weighted sum as input and calculates the output as a function of input as follows :
- y = f(x) , where f(x) is defined as
- f(x) = 0 { when x< 0.5 }
- f(x) = 1 { when x >= 0.5 }
- For our example, weighted sum is 0.55, so y = 1 ,
- That means corresponding input attribute values are classified in class 1.
- If for another input values , x = 0.45 , then f(x) = 0,
- so we could conclude that input values are classified to class 0.

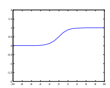
Activation functions

- The choice of activation function φ determines the neuron model.

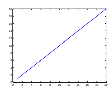
Examples:

- step function: $f(v) = \begin{cases} a & \text{if } v < c \\ b & \text{if } v > c \end{cases}$

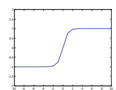
- Logistic (Sigmoid function) $f(v) = \dfrac{1}{1+\exp(-v)}$



Activation functions



Linear
$y = v$



Hyperbolic tangent
$$y = \frac{\exp(v) - \exp(-v)}{\exp(v) + \exp(-v)}$$

Bias of a Neuron

- The bias ***b*** has the effect of applying a transformation to the weighted sum ***u***

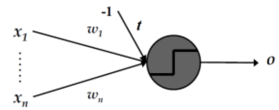
  v = u + b
- The bias is an external parameter of the neuron. It can be modeled by adding an extra input.
- ***v*** is called **induced field** of the neuron

$$v = \sum_{j=0}^{m} w_j x_j$$
$$w_0 = b$$

Bias of a Neuron

- Really, the threshold t is just another weight (called the bias):

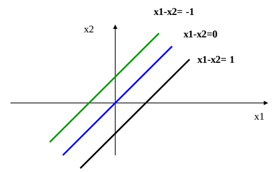
$$(w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) \geq t$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) - t \geq 0$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) + (t \times \text{-}1) \geq 0$$



Bias of a Neuron : Geometric Interpretation

- The bias value added to the weighted sum $\sum_j w_j x_j$ so that we can transform it from the origin.

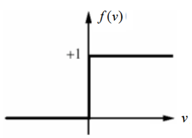
  $v = \sum_j w_j x_j + b$, *here b is the bias*



Linear Threshold Unit
Simple Perceptron Unit
Threshold Logic Unit

Use "hard-limiting" squashing function

$$f(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$$



Boolean interpretation: 0 ⇔ false, 1 ⇔ true

Perceptron for Classification

- The perceptron is used for binary classification.
- First train a perceptron for a classification task.
  - Find suitable weights in such a way that the training examples are correctly classified.

- The perceptron can only model linearly separable classes.
- Given training examples of classes C_1, C_2 train the perceptron in such a way that :
  - *If the output of the perceptron is +1 then the input is assigned to class C_1*
  - *If the output is -1 then the input is assigned to C_2*

Perceptron Training

Learning Procedure:

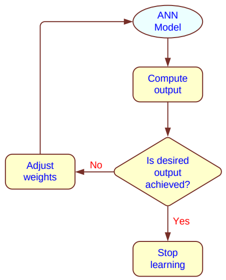
Randomly assign weights (between 0-1)

Present inputs from training data

Get output O, modify weights to gives results toward our desired output T

Repeat; stop when no errors, or enough epochs completed

A Supervised Learning Process



Three-step process:

1. Compute temporary outputs
2. Compare outputs with desired targets
3. Adjust the weights and repeat the process

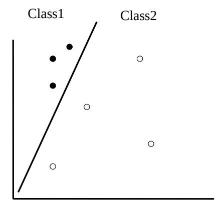
Perceptron algorithm

$\mathbf{w} \leftarrow \mathbf{0}$ (any initial values ok)
repeat
  for r=1 to R
    $\mathbf{w} \leftarrow \mathbf{w} + \eta(d^r - y^r)\mathbf{x}^r$
until no errors

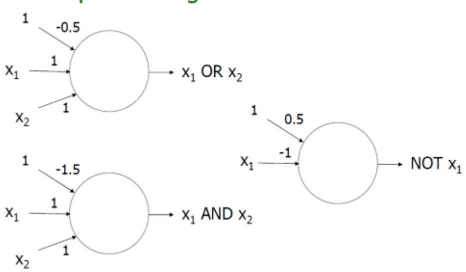
$\eta > 0$ is the learning rate
It can be taken to be 1 when inputs are 0 and 1

Perceptrons

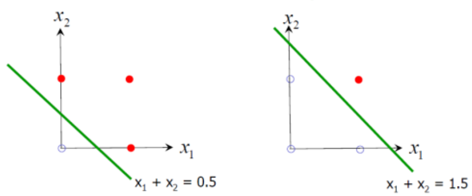
- Essentially a linear discriminant
- Perceptron theorem: If a linear discriminant exists that can separate the classes without error, the training procedure is guaranteed to find that line or plane.



Implementing Boolean Functions



Boolean examples



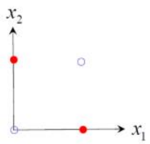
x_1 OR x_2
$w_1 = 1$
$w_2 = 1$
$w_0 = -0.5$

x_1 AND x_2
$w_1 = 1$
$w_2 = 1$
$w_0 = -1.5$

Perceptron algorithm (cont.)

- Convergence theorem: For any linearly separable training data, the algorithm converges to a solution (as long as the learning rate is suitably small). But if the data is not linearly separable, the weights loop indefinitely.

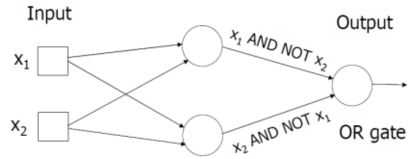
But ...



x_1 XOR x_2

- Not linearly separable
- XOR and its negation are the only Boolean functions of two arguments that are not linearly separable
- However, for larger and larger n, the number of linearly separable Boolean functions grows much more slowly than the number of possible Boolean functions

Implementing XOR with simple perceptron units



- Suffices to use one intermediate stage of simple perceptron units
- Approach generalizes to any Boolean function: write it in DNF, use one intermediate unit for each disjunct, then use an OR gate for output
- Proves that any Boolean function is realizable by a network of simple perceptron units

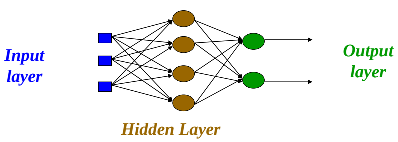
Different non linearly separable problems

Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shapes*
Single-Layer	*Half Plane Bounded By Hyperplane*	A B B A	B A	
Two-Layer	*Convex Open Or Closed Regions*	A B B A	B A	
Three-Layer	Abitrary (Complexity Limited by No. of Nodes)	A B B A	B A	

Neural Networks – An Introduction Dr. Andrew Hunter

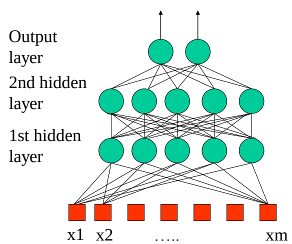
Multi layer feed-forward NN (FFNN)

- FFNN is a more general network architecture, where there are hidden layers between input and output layers.
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.
- FFNNs overcome the limitation of single-layer NN.
- They can handle non-linearly separable learning tasks.



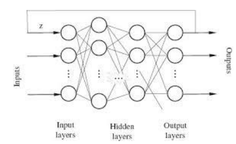
3-4-2 Network

Feed Forward Neural Networks



- The information is propagated from the inputs to the outputs
- Time has no role (NO cycle between outputs and inputs)

Recurrent Neural Networks

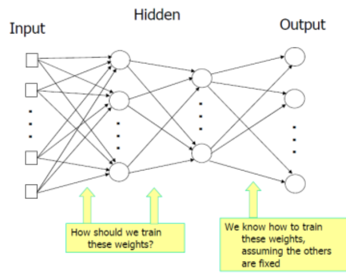


- Can have arbitrary topologies
- Can model systems with internal states (dynamic ones)
- Delays are associated to a specific weight

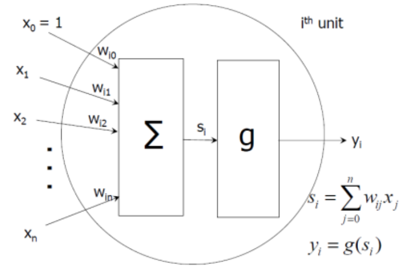
Hidden Layers

- In some cases, there may be many independencies among the input variables and adding an extra hidden layer can be helpful
- MLP with two hidden layers can approximate any non-continuous functions

Multilayer Networks

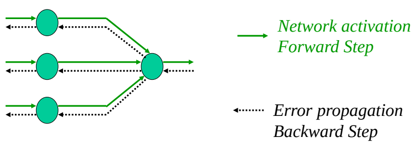


Expanded notation: necessary since using multiple units

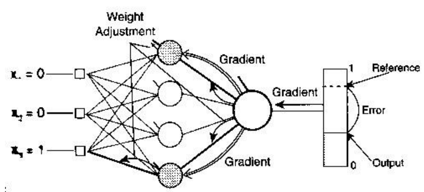


Backpropagation

- Back-propagation training algorithm



- Backpropagation adjusts the weights of the NN in order to minimize the network total mean squared error.



Backpropagation Algorithm

- The Backpropagation algorithm learns in the same way as single perceptron.
- It searches for weight values that minimize the total error of the network over the set of training examples (training set).

Given: set of input-output pairs
Task: compute weights for n-layer network to minimize the total error of the network

Backpropagation Algorithm

1.Determine the number of neurons required

2.Initialize weights to random values

3.Set activation values for threshold units

Backpropagation Algorithm

4.Choose an input-output pair and assign activation levels to input neurons

5.Propagate activations from input neurons to hidden layer neurons for each neuron

$$h_i = 1/(1 + e^{-\Sigma w1_{ij} x_j})$$

6.Propagate activations from hidden layer neurons to output neurons for each neuron

$$o_k = 1/(1 + e^{-\Sigma w2_{ki} h_i})$$

Backpropagation Algorithm

7.Compute error for output neurons by comparing pattern to actual

8.Compute error for neurons in hidden layer

9.Adjust weights in between hidden layer and output layer

10.Adjust weights between input layer and hidden layer

11.Go to step 4

Backprop learning algorithm (incremental-mode)

n=1;
initialize **weights** randomly;
while (stopping criterion not satisfied or n <max_iterations)
  for each example ($\boldsymbol{x}^r$)
  - run the network with input x and compute the output y
  - update the weights in backward order starting from those of the output layer:

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

  with Δw_{ji} computed using the (generalized) Delta rule
  end-for
  n = n+1;
end-while;

Total Mean Squared Error

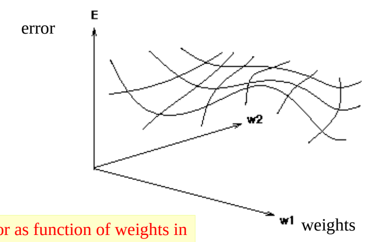
$$E[w] = \frac{1}{2}\sum(d_k^r - y_k^r)^2$$

d_k^r and y_k^r are desired and actual output of k*th* unit for training example *r*.

Where E[w] is the sum of squared errors for the weight vector w, and r ranges over examples in the training set.

Derivation of Back-propagation

Error Surface



Error as function of weights in multidimensional space

Properties of Activation Function

- Trying to make error decrease the fastest
- We need a derivative in activation function
- Activation function must be **continuous**, differentiable, non-decreasing, and easy to compute

Can’t use Step-Function

- To effectively assign credit / blame to units in hidden layers, we want to look at the first derivative of the activation function
- Sigmoid function is easy to differentiate and easy to compute forward

~~Linear Threshold Units~~ → Sigmoid function

Derivative of squashing function

- If the squashing function is the logistic function

$$g(s_i) = \frac{1}{1+e^{-s_i}}$$

the derivative has the convenient form

$$g'(s_i) = g(s_i)(1-g(s_i)) = y_i(1-y_i)$$

Exercise:
Prove this

- Another popular choice of squashing function is tanh, which takes values in the range (-1,1) rather than (0,1)
  - requires plugging a different g′ into the algorithm

Advantages

- Good mathematical foundation
- If solution exists it can be found
- Deals well with noise

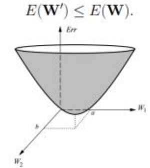
Hidden Neurons

- Choosing the number of neurons in the hidden layer often depends on the AI designer’s intuition and experience
- As the number of dimensions grows the complexity of the decision surface (path through hidden layer) increases

Why Back-propagation doesn’t work always

Producing a new vector $\mathbf{W}'$ by adding to each W_{ji} in $\mathbf{W}$ the opposite of E's slope along W_{ji} guarantees that

$$E(\mathbf{W}') \leq E(\mathbf{W}).$$

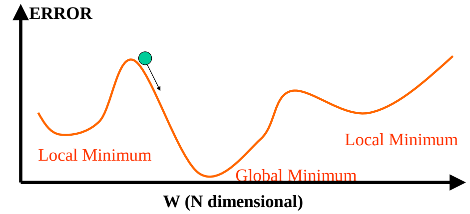


In general, however, the error surface may contain local minima.

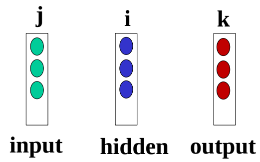
Hence, convergence to an *optimal* set of weights is not guaranteed in back-propagation learning (contrary to perceptron learning).

Training Process of the MLP

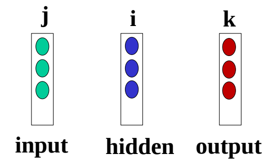
- The training will be continued until the error (RMS) is minimized.



Learning of MLP



Learning of MLP



Error in r[th] sample

$$E^r = \frac{1}{2}\sum(d_k^r - y_k^r)^2$$

d_k^r and y_k^r are desired and actual output of k*th* unit for training example r.

Overall error $E = \sum_r E^r$

Learning of MLP

Overall error $E = \sum_r E^r$

$$\frac{\partial E}{\partial w_{ik}} = \sum_r \frac{\partial E^r}{\partial w_{ik}}$$

$$-\frac{\partial E^r}{\partial w_{ik}} = -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial w_{ik}}$$

$$= \delta_k \times x_i$$

$$S_k = \sum w_{ik} \times x_i$$

$$\frac{\partial S_k}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}}\left(\sum w_{ik} \times x_i\right) = x_i$$

Learning of MLP

Overall error $E = \sum_r E^r$

$$\frac{\partial E}{\partial w_{ik}} = \sum_r \frac{\partial E^r}{\partial w_{ik}}$$

$$-\frac{\partial E^r}{\partial w_{ik}} = -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial w_{ik}}$$

$$= \delta_k \times x_i$$

$$y_k = g(S_k)$$

$$\frac{\partial y_k}{\partial S_k} = g'(S_k)$$

$$\delta_k = -\frac{\partial E^r}{\partial S_k} = -\frac{\partial E^r}{\partial y_k} \times \frac{\partial y_k}{\partial S_k} = \epsilon_k \times g'(S_k)$$

Learning of MLP

$$\delta_k = -\frac{\partial E^r}{\partial S_k} = -\frac{\partial E^r}{\partial y_k} \times \frac{\partial y_k}{\partial S_k} = \epsilon_k \times g'(S_k)$$

At each output node,

$$\epsilon_k = -\frac{\partial E^r}{\partial y_k} = -\frac{\partial}{\partial y_k}\left[\frac{1}{2}\sum(d_k^r - y_k^r)^2\right]$$

$$= d_k - y_k$$

$$\delta_k = -\frac{\partial E^r}{\partial S_k} = -\frac{\partial E^r}{\partial y_k} \times \frac{\partial y_k}{\partial S_k} = \epsilon_k \times g'(S_k)$$

At each hidden node,

$$\epsilon_i = -\frac{\partial E^r}{\partial y_i} = -\frac{\partial E^r}{\partial x_k}$$

WHY?
Output y_i of unit *i* is the input x_k of unit *k*.

$$= -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial x_k}$$

$$= \delta_k \times w_{ik}$$

$$\frac{\partial S_k}{\partial x_k} = \frac{\partial}{\partial x_k}(w_{ik} \times x_k) = w_{ik}$$

$$\delta_i = \delta_k \times w_{ik} \times g'(S_i)$$

Back Propagation Algorithm

1. Place input vector at input nodes and propagate forward.
2. At each output node *i*, compute

$$\delta_i = \epsilon_i \times g'(S_i)$$
$$= g'(S_i) \times (d_i - y_i)$$

3. At each hidden node *i*, compute

$$\delta_i = g'(S_i) \times \sum \delta_k \times w_{ik}$$

4. For each weight w_{ij} compute $\delta_i \times x_j$

$$w_{ij} \leftarrow w_{ij} + \eta\, \delta_i^r \times x_j^r$$

We need derivative. Activation function must be continuous, differentiable, non-decreasing and easy to compute.