

Lecture 4: Perceptrons and Multilayer Perceptrons

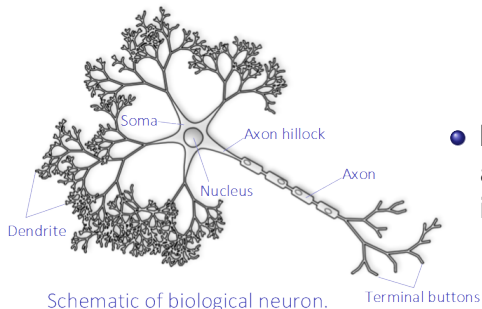
Cognitive Systems - Machine Learning

Part I: Basic Approaches of Concept Learning

Perceptrons, Artificial Neuronal Networks (ANNs), Backpropagation

last change November 26, 2014

Biological Motivation



Schematic of biological neuron.

- biological learning systems are built of complex webs of interconnected neurons

- **motivation:**

- capture kind of highly parallel computation
- based on distributed representation

- **goal:**

- obtain highly effective machine learning algorithms, independent of whether these algorithms fit biological processes (*not cognitive modeling!*)

Biological Motivation

	Computer	Brain
computation units	1 CPU ($> 18^9$ Gates)	10^{11} neurons
memory units	64 GB RAM 10 TB HDD	10^{11} neurons 10^{14} synapses
clock	10^{-10} sec	10^{-3} sec
transmission	$> 10^{11}$ bits/sec	$> 10^{14}$ bits/sec

- Computer: typically serial, quick
- Brain: parallel, slow, robust wrt noisy data

Appropriate Problems

The PERCEPTRON algorithm is an early neuro-inspired approach to machine learning

- instances are represented as many attribute-value pairs
 - input values can be any real values
- target function output is binary
- Hypotheses are linear, that is, hyper-planes which partition the hypothesis space in two areas (concept/not concept)
- When Minsky and Papert showed in their book “Perceptrons” (1969) that perceptrons cannot learn xor problems, research in neuro-inspired approaches was mainly given up until it re-emerged end of the 1980ies.
- The highly efficient learning approach of Support Vector Machines also relies on learning hyper-planes, but it uses the “kernel trick” to deal with data sets which are not linearly separable.

Appropriate Problems

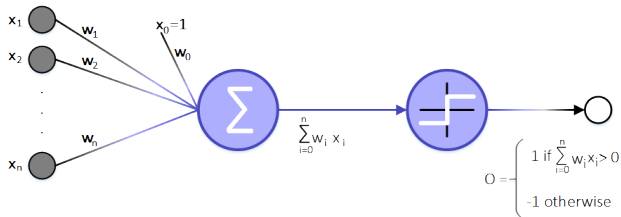
The BACKPROPAGATION algorithm is the most commonly used ANN learning technique with the following characteristics:

- instances are represented as many attribute-value pairs
 - input values can be any real values
- target function output may be **discrete-, real- or vector- valued**
- training examples **may contain errors**
- long training times are acceptable
- fast evaluation of the learned target function may be required
 - many iterations may be necessary to converge to a good approximation
- ability of humans to understand the learned target function is not important
 - learned weights are not intuitively understandable

Outline

- Perceptron
- Backpropagation
- Inductive Bias
- Summary

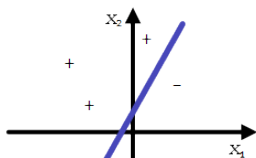
Perceptrons



- takes a vector of real-valued inputs (x_1, \dots, x_n) weighted with (w_1, \dots, w_n)
- calculates the linear combination of these inputs
 - $\sum_{i=0}^n w_i x_i = w_0 x_0 + w_1 x_1 + \dots + w_n x_n$
 - $-w_0$ denotes a threshold value, i.e. that value which must be reached by the linear combination of inputs to cause the perceptron to output 1
 - x_0 is always 1
- outputs **1** if the result is greater than 0, otherwise **-1**

Representational Power

- a perceptron represents a **hyperplane decision surface** in the n -dimensional space of instances
- some sets of examples cannot be separated by any hyperplane, those that can be separated are called **linearly separable**
- many boolean functions can be represented by a perceptron: AND, OR, NAND, NOR



(a)



(b)

Perceptron Training Rule

- **problem:** determine a weight vector \vec{w} that causes the perceptron to produce the correct output for each training example

- **perceptron training rule:**

$$w_i = w_i + \Delta w_i \text{ where } \Delta w_i = \eta(t - o)x_i$$

t target output

o perceptron output

η learning rate (usually some small value, e.g. 0.1)

Algorithm

- 1 initialize \vec{w} to random weights
 - 2 repeat, until each training example is classified correctly
 - ⇒ apply perceptron training rule to each training example
- convergence guaranteed provided *linearly separable* training examples and sufficiently small η

Delta Rule

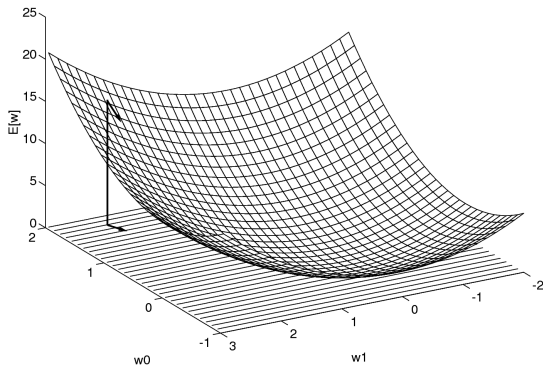
- perceptron rule fails if data is not linearly separable
- delta rule converges toward a **best-fit approximation**
- uses **gradient descent** to search the hypothesis space
 - perceptron cannot be used, because it is not differentiable
 - hence, a **unthresholded linear unit** is appropriate
 - error measure (instead of perceptron training rule):

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

- to understand gradient descent, it is helpful to visualize the entire hypothesis space with
 - all possible weight vectors and
 - associated E values

Error Surface

- the axes w_0, w_1 represent possible values for the two weights of a simple linear unit



⇒ error surface must be **parabolic** with a **single global minimum**

Derivation of Gradient Descent

- **problem:** How calculate the steepest descent along the error surface?
- derivative of E with respect to each component of \vec{w}
- this vector derivative is called *gradient* of E , written $\nabla E(\vec{w})$

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

- $\nabla E(\vec{w})$ specifies the steepest ascent, so $-\nabla E(\vec{w})$ specifies the steepest descent
- **training rule:** $w_i = w_i + \Delta w_i$

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

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

Differentiating E

$$\frac{\partial E}{\partial w_i} =$$

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

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

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

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

$$\sum_{d \in D} (t_d - o_d) (-x_{id})$$

Remember:

Outer and inner derivation for $y = u^2$: $\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$ with $u = t_d - o_d$

Incremental Gradient Descent

- application difficulties of gradient descent
 - convergence may be quite slow
 - in case of many local minima, the global minimum may not be found
- **idea:** approximate gradient descent search by updating weights *incrementally*, following the calculation of the error for *each* individual example

$$\Delta w_i = \eta(t - o)x_i \text{ where } E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

- **key differences:**
 - weights are not summed up over all examples before updating
 - requires less computation
 - better for avoidance of local minima

Gradient Descent

Algorithm

GRADIENT-DESCENT(*training_examples*, η)

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

- 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 $\Delta w_i = \Delta w_i + \eta(t - o)x_i^*$
 - ▶ For each linear unit weight w_i , Do $w_i \leftarrow w_i + \Delta w_i^{**}$

To implement *incremental approximation*, equation ** is deleted and equation * is replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Perceptron vs. Delta Rule

- **perceptron training rule:**

- uses thresholded unit
- converges after a finite number of iterations
- output hypothesis classifies training data perfectly
- linear separability necessary

- **delta rule:**

- uses unthresholded linear unit
- converges asymptotically toward a minimum error hypothesis
- termination is not guaranteed
- linear separability not necessary

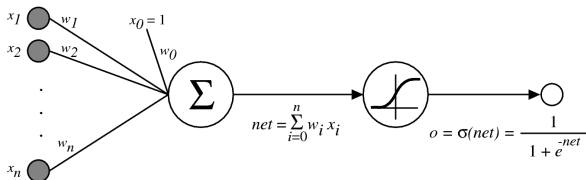
Multilayer Networks (ANNs)

- capable of learning **non-linear decision surfaces**
- normally **directed** and **acyclic** \Rightarrow Feed-forward Network
- based on **sigmoid unit**
 - much like a perceptron
 - but based on a smoothed, **differentiable threshold function**

$$\sigma(net) = \frac{1}{1+e^{-net}}$$

$$\lim_{net \rightarrow +\infty} \sigma(net) = 1$$

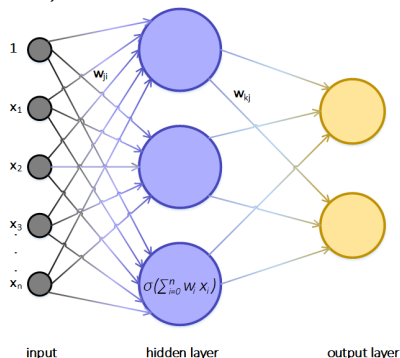
$$\lim_{net \rightarrow -\infty} \sigma(net) = 0$$



Backpropagation

- learns weights for a feed-forward multilayer network with a fixed set of neurons and interconnections
- employs gradient descent to minimize error
- redefinition of E
 - has to sum the errors over all units
 - $E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$

- **problem:**
search through a large H
defined over all possible
weight values for all units in
the network



Backpropagation - Algorithm

BACKPROPAGATION(*training_examples*, η , n_{in} , n_{out} , n_{hidden})

The input from unit i to 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
- Until the **termination condition** is met, Do (EPOCHE)
 - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

Propagate the **input forward** through the network:

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

Propagate the **errors back** 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

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

4. Update each weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \text{ where } \Delta w_{ji} = \eta \delta_j x_{ji}$$

Termination conditions

- fixed number of iterations
- error falls below some threshold
- error on a separate validation set falls below some threshold
- **important:**
 - too few iterations reduce error insufficiently
 - too many iterations can lead to overfitting the data

Adding Momentum

- one way to avoid local minima in the error surface or flat regions
- make the weight update in the n^{th} iteration depend on the update in the $(n - 1)^{\text{th}}$ iteration

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n - 1)$$

Note: $\Delta w_{ji}(n - 1)$ represents the cumulative updates for this weight in the complete last epoch.

$$0 \leq \alpha \leq 1$$

Representational Power

- *boolean functions*:
 - every boolean function can be represented by a two-layer network
- *continuous functions*:
 - every continuous function can be approximated with arbitrarily small error by a two-layer network (sigmoid units at the hidden layer and linear units at the output layer)
- *arbitrary functions*:
 - each arbitrary function can be approximated to arbitrary accuracy by a three-layer network

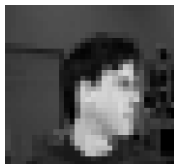
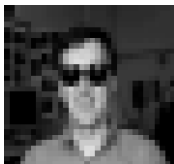
Inductive Bias

- every possible assignment of network weights represents a syntactically different hypothesis

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

- inductive bias:** smooth interpolation between data points
 - Multilayer Networks:** smooth interpolation between data points
 \Rightarrow **Preference bias**
 - Perceptron:** linear separability necessary
 \Rightarrow **Restriction bias**

Illustrative Example - Face Recognition



Task:

- classifying camera image of faces of various people
- images of 20 people were made, including approximately 32 different images per person
- image resolution 120×128 with each pixel described by a grey-scale intensity between 0 and 255
- identifying the direction in which the persons are looking (i.e., left, right, up, ahead)

Illustrative Example - Design Choices

- **input encoding:**

- image encoded as a set of 30×32
- pixel intensity values ranging from 0 to 255 linearly scaled from 0 to 1
- ⇒ reduces the number of inputs and network weights
- ⇒ reduces computational demands

- **output encoding:**

- network must output one of four values indicating the face direction
- *1-of-n* output encoding: 1 output unit for each direction
- ⇒ more degrees of freedom
- ⇒ difference between highest and second-highest output can be used as a measure of classification confidence

Illustrative Example - Design Choices

- **network graph structure:**

- BACKPROPAGATION works with any DAG of sigmoid units
- question of how many units and how to interconnect them
- using *standard design*: hidden layer and output layer where every unit in the hidden layer is connected with every unit in the output layer

⇒ 30 hidden units

⇒ test accuracy of 90%

Advanced Topics

- hidden layer representations
- alternative error functions
- recurrent networks
- dynamically modifying network structure

Summary

- able to learn discrete-, real- and vector-valued target functions
- noise in the data is allowed
- perceptrons learn hyperplane decision surfaces (linear separability)
- multilayer networks even learn non-linear decision surfaces
- **BACKPROPAGATION** works on arbitrary feed-forward networks and uses gradient-descent to minimize the squared error over the set of training examples
- an arbitrary function can be approximated to arbitrary accuracy by a three-layer network
- **Inductive Bias**: smooth interpolation between data points

Learning Terminology

Perceptrons / Multilayer Perceptrons

Supervised Learning	unsupervised learning
----------------------------	-----------------------

Approaches:

Concept / Classification	Policy Learning
symbolic	statistical / neuronal network
inductive	analytical

Learning Strategy: Data: Target Values:	learning from examples categorical/metric features arbitrary (concept, class, value, vector)
---	---