Peter Csiba 10.03.2011

Project number one - Perceptron

Introduction

Our task was to implement Perceptrons for solving tasks. All theory behind was gathered from the slides from the lectures:
http://ii.fmph.uniba.sk/~farkas/Courses/NeuralNets/Slides/perceptron.4x.pdf.

I have chosen Java programming language and make this project Object oriented. Before I started this project I get some advices from older students. On behalf of these advices I built this code (considering that the next project will be a multilayer network).
Note that sometimes I used third person and sometimes first. Usually first person I used where I somehow experiment with constants.

Go to Appendix A for the source code.

Program loads data, creates Perceptron (Continuous / Discrete) and for some alpha and some epoch_count runs the learning process.This is repeated simulation_count times and we take the avarage of some measures.
We were interested in the avarage convergency rate (epoch finished) for DiscretePerceptron and for avarage total error after hundreds of epoch for ContinuousPerceptron.

Part A: Show that binary discrete perceptron can implement logic functions AND and OR.

For demonstration I show the main loop:

```
for(int sim_id=1; sim_id <= simulation_count; sim_id++){
      for(int i=1 ; i <= alpha_count ; i+=1){
            for(epoch_id=1; epoch_id <= epoch_count ; epoch_id++){
                  error = perceptron.train(training_set, true, log);
                  if(error <= 0){
                                    break;
                  }
            }
      }
}
```

I used linear learning function (delta w = alpha * x * (d-y)). This stands for a linear function. Trivial activation function:

```
            return ((y < 0)?0.0 : 1.0);
```

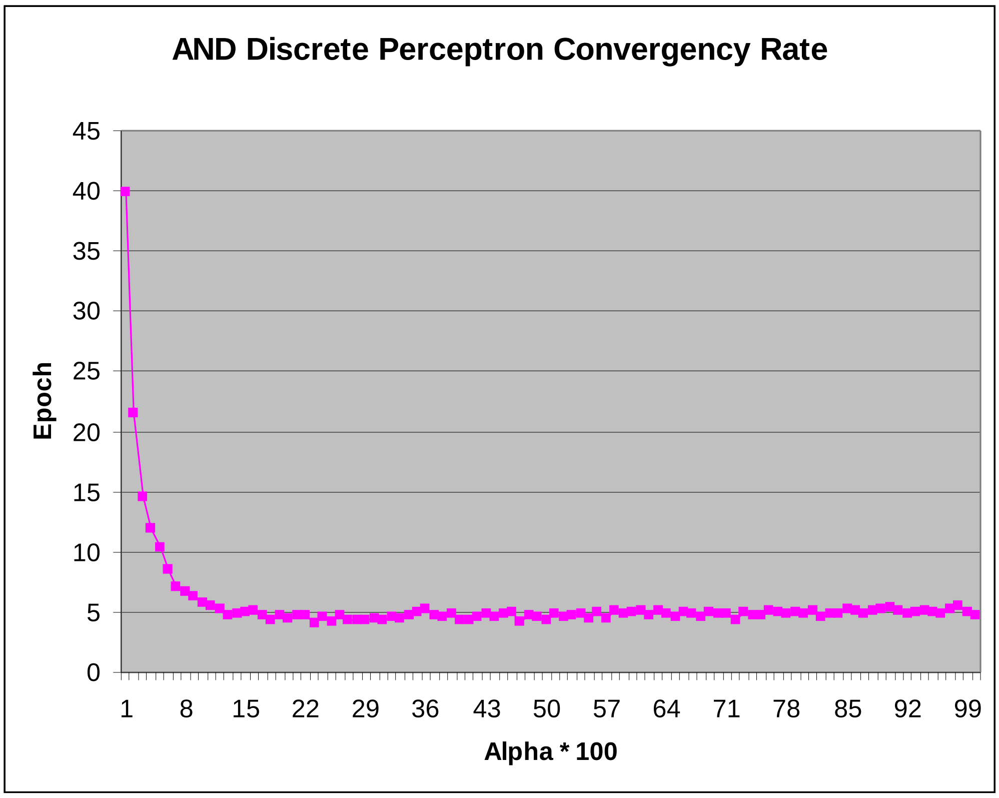
Before a new epoch weights are (re)setted randomly:

```
            for(int i=0; i<this.dimension ; i++){
                  //Next double returns from interval <0,1>
                  this.weight.set(i, r.nextDouble());
```

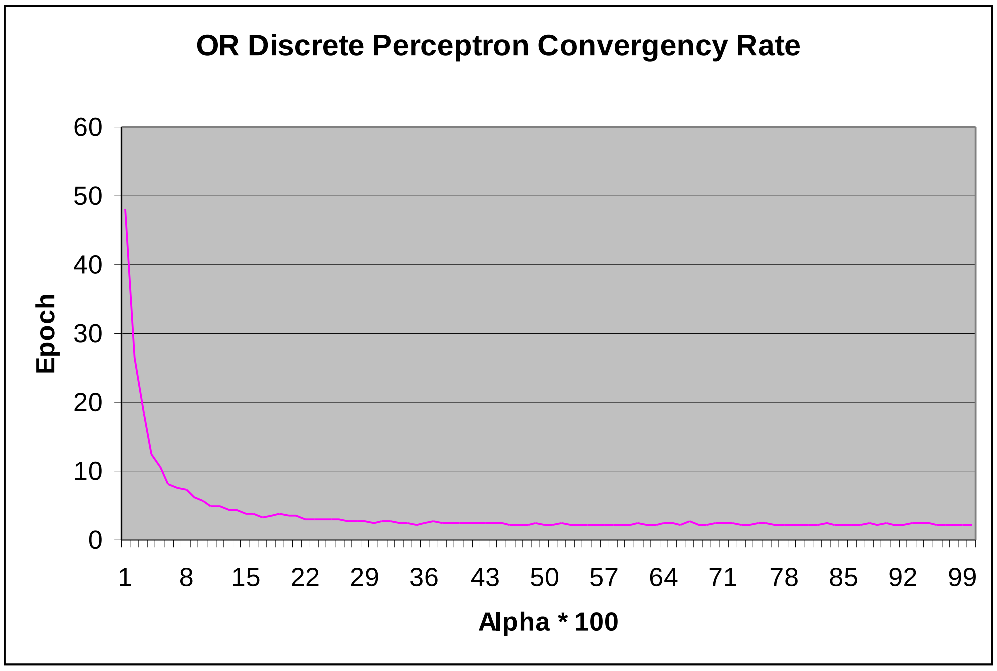
Treshold is set as a constant input (1.0):

```
this.input.set(this.dimension-1, new NetworkConstant(treshold));
```

Log from the learn process can be found in the Appendix B.

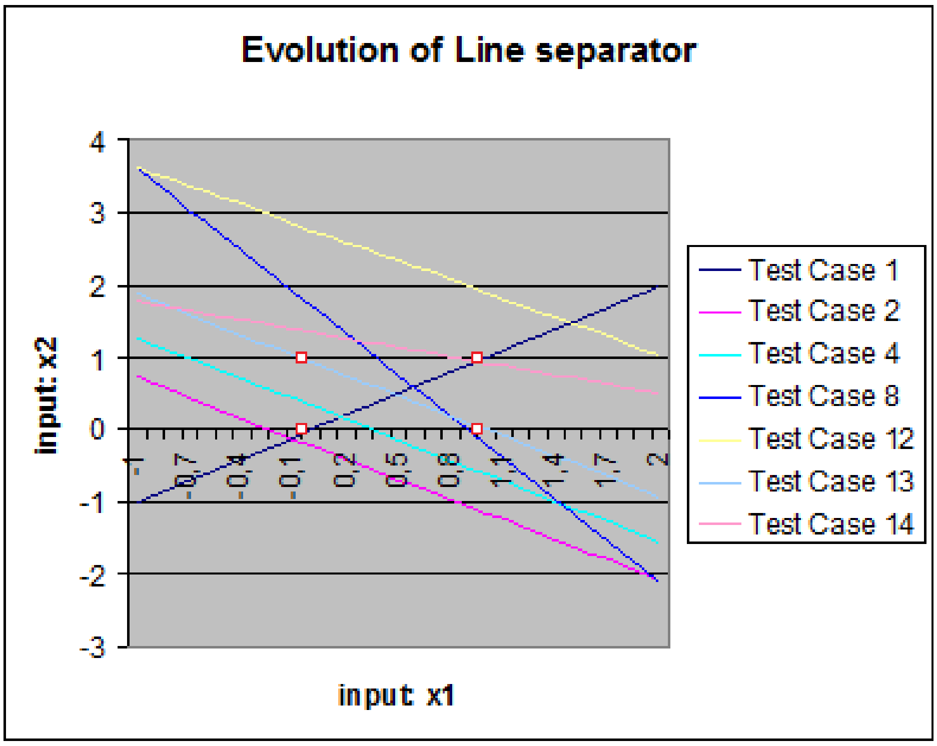


This plot shows avarage convergency rate for a DiscretePerceptron which tries to learn AND. First, alpha was chosen uniformly from <0.0, 1.0>. Then the learning process is repeated simulation_count times. For each simulation we save the epoch in which the simulation ended. After that we compute the avarage.
Constrains used: epoch_count:100, simulation_count:20, 100 different alfa from 0.01 to 1.00, treshold =1.0, not constrained weights, activation function (x < 0):0; (x >= 0) : 1.
Excel file: and.xls



We see that the both plots are exponential functions.
Excel file: or.xls

Now we look in the inards of a train process of the DiscretePerceptron. In Appendix B we publish a log from the learning process.
From lecture we know a theorem which states that Perceptron could separate in any way (and always separates in some way) a R^N space by a R^{N-1} space. Specially it could separate plane with a line.



In this plot we demonstrate the linear separability of Perceptron. For this case we have chosen alpha=0.25. The Perceptron converged in 4 epochs. The red-white points show the four diferent inputs (0,0;0,1;1,0;1,1) (treshold is always 1.0). For transparency we show only the moments when the line changed.

Part A.C: Consider continuous perceptron for this task.

We know that the ContinousPerceptron never returns values 0.0 and 1.0 (except when the float precision is not sufficient). So in this case we must consider some intervals <0, d>, <1-d,1> which we accept as 0, 1 (in my code d is named as ContinuousPerceptron.treshold). Between the singleton Discrete and Continuous Perceptrons resolving these logic problems there is asymptotically no difference (I have tried a simulation).

Resume:
We have seen on this basic model the trivial functionality of a DiscretePerceptron.

Part B: Try if a ContinuousPerceptron could separate a compact shape.
As activation function I have used a not scaled sigmoid (so the alpha values were larger)

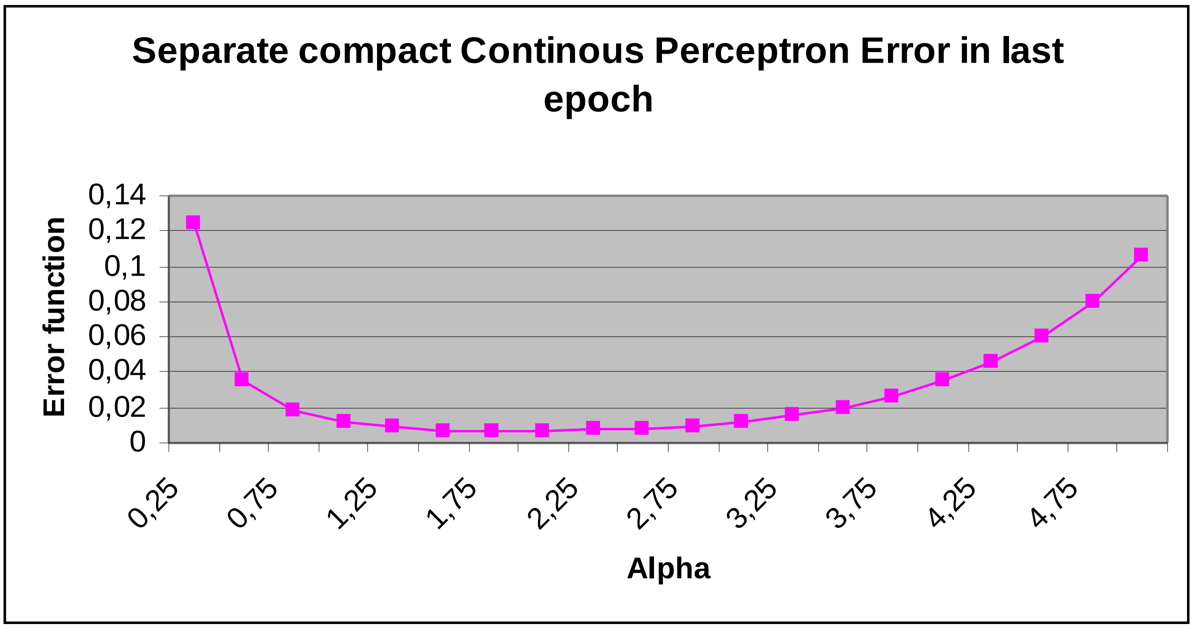
```
      protected double activationFunction(double y) {
                  double x =  1 / (1 + Math.pow(Math.E, -y));
                  if(x >= 1-this.tolerance) return 1.0;
                  if(x <= this.tolerance) return 0.0;
                  return x;
      }:
```

Note that tolerance=0.0.

The learn speed function (how fast are the weights changed) is the derivate of the sigmoid. Because I have a very small network I used the full derivation from wolframalpha.com:

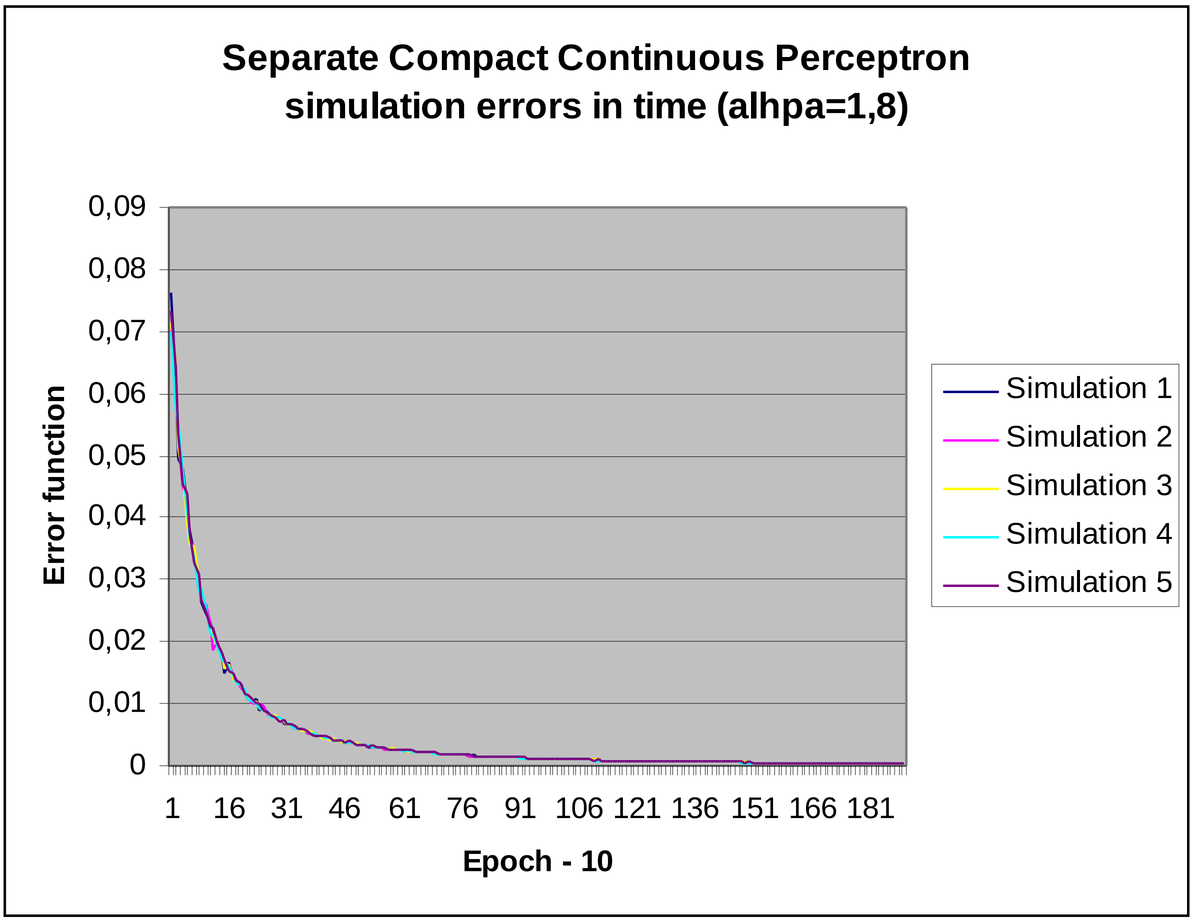
```
return (alpha*Math.pow(Math.E, alpha*x)) / Math.pow(1 + (Math.pow(Math.E,
alpha*x)), 2);
```

In larger networks it is recommended to use a simplified learning function (1-y)(y)*alpha which is very similiar on the neighbourhood of 0.0 to the derivate of the sigmoid.



In this plot we see the average error rate after 100 learning epochs of the ContinousPerceptron on the SeparateCompact task. I have used no tolerance in the activation function, because I wanted to see in which cases is the input learned best.
I have considered 20 different alphas from 0.25 to 5.00. Here we show the average of 5 simulations for each alpha. The dimension of the input was 25 and the chosen constant treshold was 1.0 (treshold input has also a weight).
Excel file: separate_compact_errors.xls

Based on this data I have chosen a convenient alpha=1,8 and ran 5 simulations with this alpha.



In the plot above we see the error rate in 5 different simulations through epochs 11-190. It is clear that it exponentially falls. Also we see that the error rates are almost the same, so it doesn`t depends much on the initial inputs in the long run. Also i we generate a plot for the first ten steps, there is no bigger difference (See Apendix C).
We used the same constraints as in the plot before.
Excel file: sep_comp_18_errors.xls

Resume:
I have learned how to implement a basic ContinuousPerceptron. I wonder if the Perceptron I have trained could classify other problem instances as in the TrainingSet.
We have learned that bigger alpha doesn`t mean better convergence rate. Also we see that the precision of ContinuousPerceptron decision falls down exponentially.
In bigger networks we should use y(1-y) learnSpeedFunctions to speed up our network little bit.

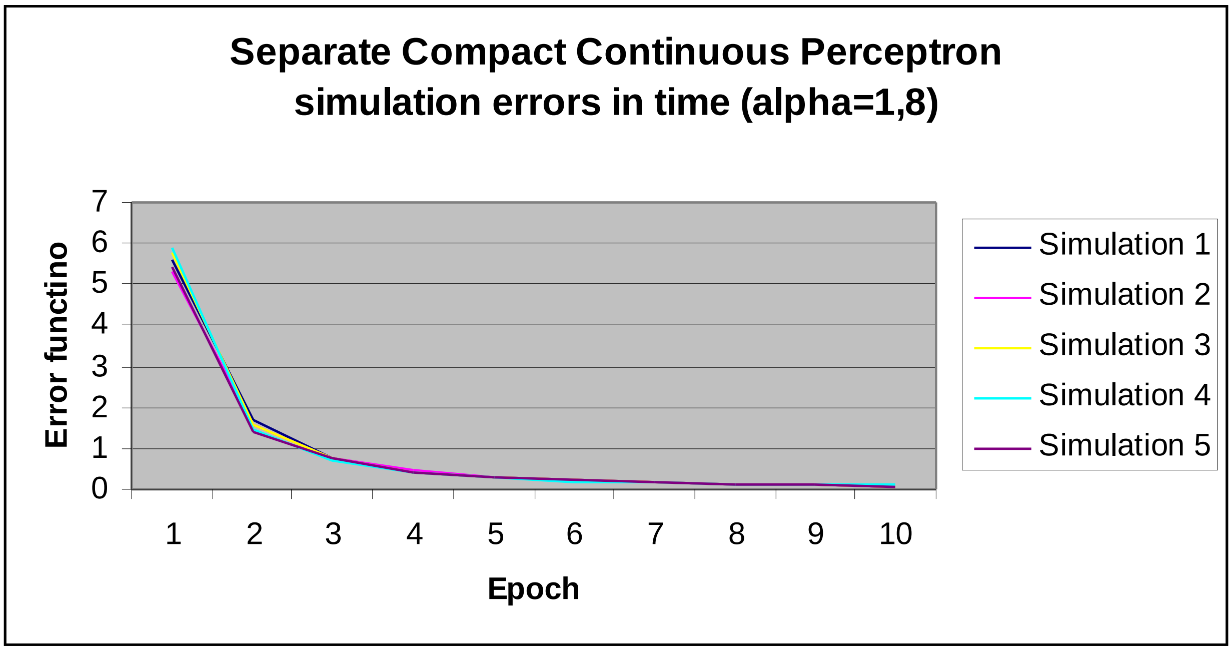
Appendix A:
Project data are also available on my homepage: http://www.csip.sk/?page_id=497

Appendix B:

Log from one simulation of a DiscretePerceptron with alpha=0,25

```
=====NEW ALPHA:0.25=====
===NEW EPOCH===
=NEW TEST CASE=
Input: 1.0 0.0 1.0 : 0.0
y=1.0 d=0.0
Weights: 0.45660029929967705 0.4881280243838163 -0.1577195337708719
=NEW TEST CASE=
Input: 1.0 1.0 1.0 : 1.0
y=1.0 d=1.0
Weights: 0.45660029929967705 0.4881280243838163 -0.1577195337708719
=NEW TEST CASE=
Input: 0.0 1.0 1.0 : 0.0
y=1.0 d=0.0
Weights: 0.45660029929967705 0.2381280243838163 -0.4077195337708719
=NEW TEST CASE=
Input: 0.0 0.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.45660029929967705 0.2381280243838163 -0.4077195337708719
alpha=0.25 ,error=1.0
===NEW EPOCH===
=NEW TEST CASE=
Input: 1.0 1.0 1.0 : 1.0
y=1.0 d=1.0
Weights: 0.45660029929967705 0.2381280243838163 -0.4077195337708719
=NEW TEST CASE=
Input: 0.0 0.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.45660029929967705 0.2381280243838163 -0.4077195337708719
=NEW TEST CASE=
Input: 1.0 0.0 1.0 : 0.0
y=1.0 d=0.0
Weights: 0.20660029929967705 0.2381280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 0.0 1.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.2381280243838163 -0.6577195337708719
alpha=0.25 ,error=0.5
===NEW EPOCH===
=NEW TEST CASE=
Input: 0.0 0.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.2381280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 0.0 1.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.2381280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 1.0 1.0 1.0 : 1.0
y=0.0 d=1.0
Weights: 0.45660029929967705 0.4881280243838163 -0.4077195337708719
=NEW TEST CASE=
Input: 1.0 0.0 1.0 : 0.0
y=1.0 d=0.0
Weights: 0.20660029929967705 0.4881280243838163 -0.6577195337708719
alpha=0.25 ,error=1.0
===NEW EPOCH===
=NEW TEST CASE=
Input: 0.0 1.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.4881280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 1.0 0.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.4881280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 0.0 0.0 1.0 : 0.0
y=0.0 d=0.0
Weights: 0.20660029929967705 0.4881280243838163 -0.6577195337708719
=NEW TEST CASE=
Input: 1.0 1.0 1.0 : 1.0
y=1.0 d=1.0
Weights: 0.20660029929967705 0.4881280243838163 -0.6577195337708719
alpha=0.25 ,error=0.0
```

Appendix C


Separate Compact Continuous Perceptron simulation errors in time (alpha=1,8)
Error functino
Epoch
Simulation 1
Simulation 2
Simulation 3
Simulation 4
Simulation 5