Exercise: Training Simple MLP by Backpropagation. Using Netlab.

Petr Pošík

December 11, 2007

1 File list

This document is an explanation text to the following script:

- `demoMLPKlin.m` — script implementing the beckpropagation algortihm derived in Sec. 2. Uses data set `tren.dat` and m-file `mlperc.m` which implements the neural network.
- `demoNetlabKlin.m` — constructs the same neural network as the first script (see Sec. 3.1), this time using Netlab (which must be downloaded separately). Uses data set `tren.dat` and helper function `plotData.m`.
- `demoNetlabXOR.m` — uses Netlab to construct the neural network for the XOR data set, see Sec. 3.2 (`trenXORrect.dat` contains rectangular XOR pattern, `trenXOR.dat` contains rotated version). Uses helper function `plotData.m`.

2 Backpropagation for simple MLP

Suppose we have a training set that exhibits the pattern shown in Fig. 2:

The task is to create a simple neural network classifier that would be able to discriminate between red crosses and blue circles.

2.1 Preliminary questions

2.1.1 Is one simple perceptron sufficient to classify the training points precisely?

No. One perceptron creates only one *linear* decision boundary.

2.1.2 How many linear boundaries do we need here?

At least two.

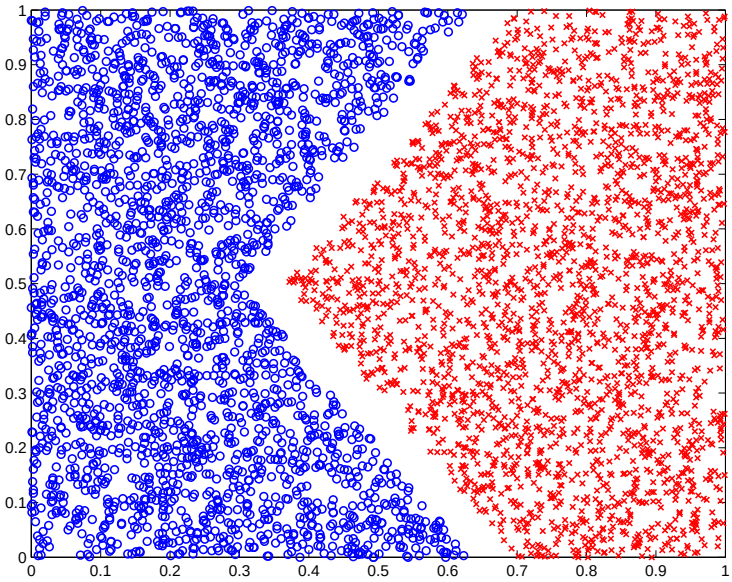

Figure 1: The first training set.

2.1.3 What architecture of the network will be suitable?

Well, we need two linear decision boundaries, i.e. two neurons—variables x_1 and x_2 will be the inputs to both of them.

OK, but this would give us a network that has two outputs. We need additional neuron to combine the outputs of the first two neurons into one output.

2.1.4 We need to tune the NN to the training set. What is actually tuned?

The weights $\mathbf{w}$.

2.2 Questions for our network

2.2.1 What does our network look like?

Our network is depicted in Fig. 2.

The output nonlinear function g is the same for all the three neurons:

$$g(a) = \frac{1}{1 + e^{-\lambda a}} \tag{1}$$

To reiterate, its derivative in point a can be computed as

$$g'(a) = \lambda g(a)(1 - g(a)) \tag{2}$$

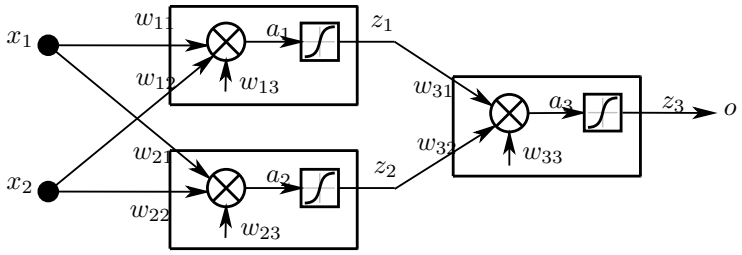

Figure 2: Our neural network.

2.2.2 How many weights have to be tuned in our network?

It seems that 6. For two inputs to neuron 1, two inputs to neuron 2, and 2 inputs for neuron 3. Really?

Well, all the neurons should also have additional input that is connected to value 1 (do you remember the homogeneous coordinates?). That's why we have 3 inputs for each of the 3 neurons which gives 9 weights.

2.2.3 Given an input vector x, how do we compute the output of our NN?

We use forward propagation. The weights related to neuron 1 form a vector $\mathbf{w}_1 = (w_{11}, w_{12}, w_{13})$. Simlarly we have vectors of weights for neurons 2 and 3: $\mathbf{w}_2 = (w_{21}, w_{22}, w_{23})$ and $\mathbf{w}_3 = (w_{31}, w_{32}, w_{33})$.

For a given vector $\mathbf{x} = (x_1, x_2)$, we create a vector $\tilde{\mathbf{x}}$ in homogeneous coordinates:

$$\tilde{\mathbf{x}} = (x_1, x_2, 1). \tag{3}$$

Then we can compute the weighted sums a_1 and a_2 for neurons 1 and 2:

$$a_1 = \mathbf{w}_1^T \tilde{\mathbf{x}}, \tag{4}$$

$$a_2 = \mathbf{w}_2^T \tilde{\mathbf{x}}, \tag{5}$$

and we can also compute the outputs z_1 and z_2 of neurons 1 and 2:

$$z_1 = g(a_1), \tag{6}$$

$$z_2 = g(a_2). \tag{7}$$

Then, we can construct the input vector for the third neuron as

$$\tilde{\mathbf{x}}_H = (z_1, z_2, 1) \tag{8}$$

and pass it through the third neuron

$$a_3 = \mathbf{w}_3^T \tilde{\mathbf{x}}_H, \text{ and} \tag{9}$$

$$z_3 = o = g(a_3). \tag{10}$$

This way we have gained the values of all the z_i in the whole network.

2.2.4 How do we compute the weights of the network?

We define the loss function (optimization criterion) E as

$$E = \frac{1}{2}\sum_{n=1}^{N}\sum_{c=1}^{C}(y_{nc} - o_{nc})^2. \tag{11}$$

i.e. as the sum of squares of errors of all outputs ($c = 1, \ldots, C$) for all training examples ($n = 1, \ldots, N$). However we can consider only the error for one training example:

$$E = \frac{1}{2}\sum_{c=1}^{C}(y_c - o_c)^2. \tag{12}$$

And, since our network has only one output, the error is defined as

$$E = \frac{1}{2}(y - o)^2. \tag{13}$$

Since the output o is actually a function of weights $o(\mathbf{w})$, we can optimize the error function E by modifying the weights. We shall use the gradient descent method. The update rule of the gradient descent method is

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \mathrm{grad}(E(\mathbf{w})) \tag{14}$$

2.2.5 How do we compute the gradient?

The gradient in point $\mathbf{w}$ can be computed by using the backprop. algorithm. In our case, the gradient can be written as a matrix (similarly as $\mathbf{w}$):

$$\mathrm{grad}(E(\mathbf{w})) = \begin{pmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} & \frac{\partial E}{\partial w_{13}} \\ \frac{\partial E}{\partial w_{21}} & \frac{\partial E}{\partial w_{22}} & \frac{\partial E}{\partial w_{23}} \\ \frac{\partial E}{\partial w_{31}} & \frac{\partial E}{\partial w_{32}} & \frac{\partial E}{\partial w_{33}} \end{pmatrix} \tag{15}$$

From the derivation of the backpropagation algrotihm we know that the individual derivatives can be computed as

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i, \tag{16}$$

where δ_j is the error of the neuron on the output of the edge $i \to j$, and z_i is the signal on the input of the edge $i \to j$. Values of z_is are known from the forward propagation phase, we need to compute the δ_js.

Again, from the derivation of the backpropagation algorithm we know, that for the output layer we can write

$$\delta_3 = \underbrace{g'(a_3)}_{\lambda z_3(1-z_3)} \frac{\partial E}{\partial o} = \lambda z_3(1 - z_3) \cdot (-1)(y - o) \tag{17}$$

For the δ_js in the hidden layer, we can write

$$\delta_j = g'(a_j) \sum_{k \in \mathrm{Dest}(j)} w_{kj}\delta_k \tag{18}$$

Consider neuron 1. The set of neurons which are fed with signal from neuron 1 is $\mathrm{Dest}(1) = 3$, so that the sum will have only one element:

$$\delta_1 = g'(a_1) \sum_{k \in \mathrm{Dest}(1)} w_{k1}\delta_k = \tag{19}$$

$$= \lambda z_1(1 - z_1)w_{31}\delta_3 \tag{20}$$

Similarly, for the neuron 2 we can write

$$\delta_2 = \lambda z_2(1 - z_2)w_{32}\delta_3 \tag{21}$$

That completes the backpropagation algorithm for our particular network. We know the z_is from the forward propagation, we have computed the δ_js using the backpropagation algorithm, we can thus compute the gradient of the error function $\mathrm{grad}(E(\mathbf{w}))$ and make the gradient descent step $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \mathrm{grad}(E(\mathbf{w}))$.

2.2.6 How can that be written in MATLAB?

The following code is the MATLAB implementation of our derived procedure. The only difference is that we have derived the on-line version, the code is for batch algorithm.

```
function err = train()
  % W - [3 x 3] matrix of NN weights
  % x - [nin x 2] matrix of coordinates x1 and x2
  % y - [nin x 1] matrix of target values (1 or 0)

  % Propagate training points through network
  nin = size(x,1);
  data = [x ones(nin,1)];
  z1 = perc(W(1,:), data);
  z2 = perc(W(2,:), data);
  data2 = [z1 z2 ones(nin,1)];
  z3 = perc(W(3,:), data2);

  % Compute average error for 1 point
  err = 0.5 * sum((y-z3) .^ 2) / nin;

  % Delta in output layer, neuron 3
  delta3 = lambda * z3 .* (1-z3) .* -(y-z3);
  % Propagate errors back to hidden layer, neurons 1 and 2
```

```
  delta1 = lambda * z1 .* (1-z1) .* delta3 * W(3,1);
  delta2 = lambda * z2 .* (1-z2) .* delta3 * W(3,2);

  % Compute the derivatives
  dEdW1 = delta1' * data;
  dEdW2 = delta2' * data;
  dEdW3 = delta3' * data2;

  % Compile the gradient of the weight matrix
  nablaEw = [dEdW1; dEdW2; dEdW3];

  % Adapt the weights using the gradient descent rule
  W = W - eta * nablaEw;

  % Apply decay factor
  eta = eta*quo;
end
```

2.2.7 How can we interpret the weights after learning?

After repeatedly running the training algorithm for e.g. 1000 iterations, we hopefully get near-optimal weights. The meaning of individual parts of our trained NN can be seen in Fig. 3

3 Using Netlab

Netlab[1] is a set of MATLAB functions that allows us to create simple neural networks (among other things). It was created by Ian Nabney and Christopher Bishop who is the author of the very popular book Neural Networks for Pattern Recognition.

3.1 Netlab and our first data set

3.1.1 Crucial functions of Netlab

First, we apply the Netlab to solve the same problem as in previous section. The important part of the script can be seen here:

```
%% Suppose:
% x - [ntr x 2] input part of training vectors
% y - [ntr x 1] output part of training vectors
% xt - [ntst x 2] input part of testing vectors
% yt - [ntst x 1] output part of testing vectors
```

[1] Download at `http://www.ncrg.aston.ac.uk/netlab/`.

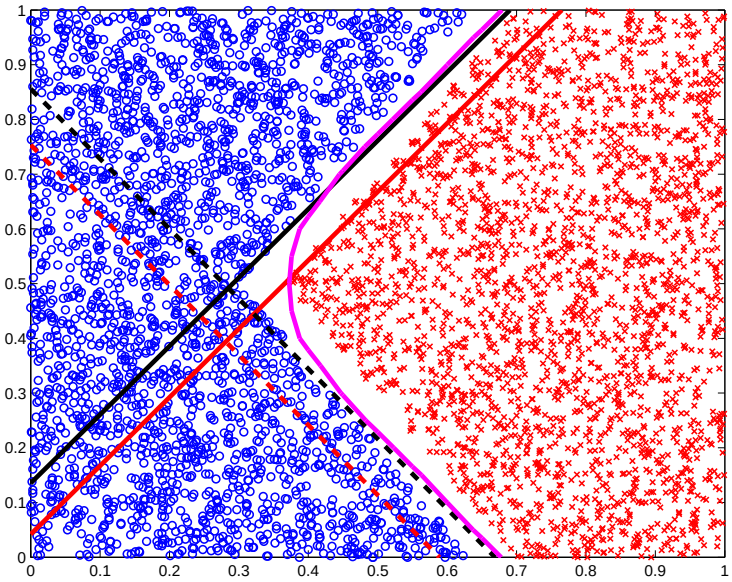

Figure 3: The meaning of the elements in the net. Line — is a set of points for which $a_1(\tilde{\mathbf{x}}) = 0$, line — is a set of points for which $a_1(\tilde{\mathbf{x}}) = 1$, line --- is a set of points for which $a_2(\tilde{\mathbf{x}}) = 0$, line --- is a set of points for which $a_2(\tilde{\mathbf{x}}) = 1$, and line — is the decision boundary of the whole network, i.e. set of points for which $z_3(\tilde{\mathbf{x}}) = 0.5$.

```
%% Create network structure
% 2 inputs, 2 neurons in hidden layer, 1 output.
% The nonlinear function for the output neuron is 'logistic'.
% The nonlinear funtions for the neurons in hidden layer is
% hardwired and is hyperbolic tangent.
net = mlp(2,2,1,'logistic');

%% Set the options for optimization
options = zeros(1,18);
options(1) = 1;         % This provides display of error values.

%% Optimize the network weights
% Train the network in structure 'net' to correctly classify
% input vectors x to outputs y using 'scg' (scaled conjugate
% gradients).
[net, options] = netopt(net, options, x, y, 'scg');

%% Compute the predictions of the trained network
% for the testing data
ypred = mlpfwd(net, xt);
```

```
% Compute errors on testing data
errors = yt - ypred;
```

As can be seen, for our purposes (multilayer perceptron) we need only 3 functions of Netlab:

- `net = mlp(nin, nhid, nout, func)` which creates a data structure holding the NN with `nin` inputs, `nhid` neurons in hidden layer, and `nout` outputs. The nonliner function of output neurons is set to `func`. Further info: `help mlp`.
- `net = netopt(net, options, x, y, 'scg')` which optimizes the NN weights (hidden in structure `net` that was created by function `mlp`) so that the NN will have the least possible error when classifing the training inputs `x` into training outputs `y`. It can use several optimization algorithms, one of them is scaled conjugate gradients, i.e. `scg`. Further info: `help netopt`.
- `ypred = mlpfwd(net, xt)` which provides the predictions that the `net` gives us when new data points `xt` are presented to our NN. Further info: `help mlpfwd`.

3.1.2 Netlab results on our first data set

If we train the same network on the same data as in previous section, we shall expect very similar results. The difference is that (1) the nonlinear functions in hidden layer are not logistic, but hyperbolic tangent, and (2) we do not use gradient descent to optimize the weights, but rather the method of scaled conjugated gradients. The results of training can be seen in Fig. 4.

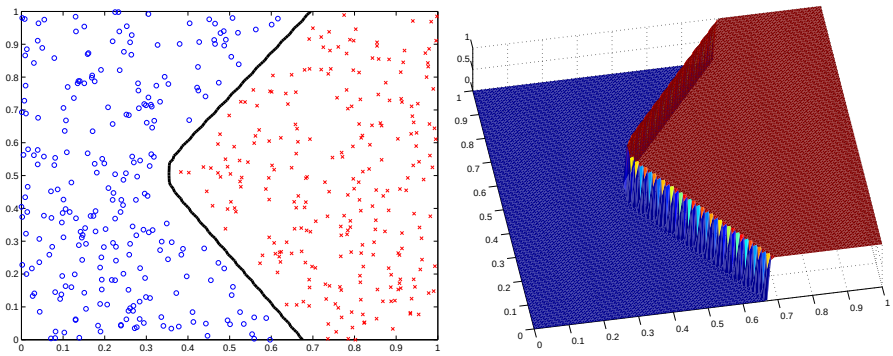

Figure 4: Results for the network from Fig. 2. Left: the decision boundary, right: the discrimination function.

3.2 Netlab and the XOR problem

Consider the pattern of training data points depicted in Fig. 5.

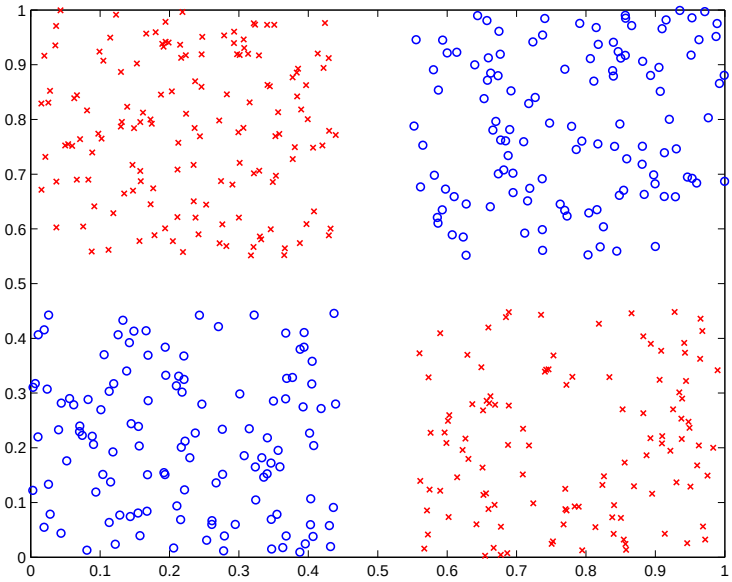

Figure 5: Training data forming the XOR pattern.

Similarly to our first data set, the two classes are not linearly separable. It seems reasonable to use our 2-2-1 network for this data set as well.

3.2.1 Locally optimal solution

In Fig. 6, we can see one possible solution where the NN tries to separate one corner of the training data from the rest of the patterns.

It can be seen that such a network makes pretty large error (see the image title). However, there exists another solution of this problem which is very hard to obtain if the NN already encodes the decision boundary from Fig. 6, i.e. when the NN is in local optimum.

3.2.2 Globally optimal solution

A better solution is depicted in Fig. 7. The title of the left picture clearly shows that the solution is better (makes lower error with respect to the given data set).

Looking at Fig. 7, one can ask where did the second decision boundary come from. The same picture in somewhat larger scale reveals that there is no second boundary, they are just 2 parts of 1 boundary, see Fig. 8.

3.2.3 More complex network

The first and the second datasets were similar in the respect that they are not linearly separable. However, they are different in the aspect that the first data

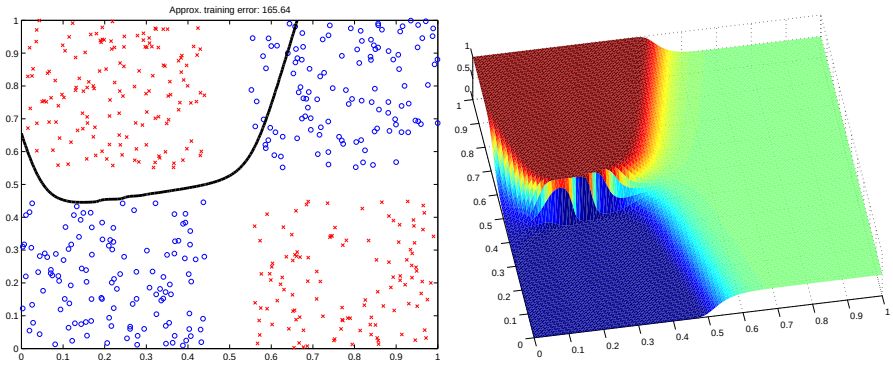

Figure 6: Locally optimal solution of the network from Fig. 2 applied to the XOR pattern. Left: the decision boundary, right: the discrimination function.

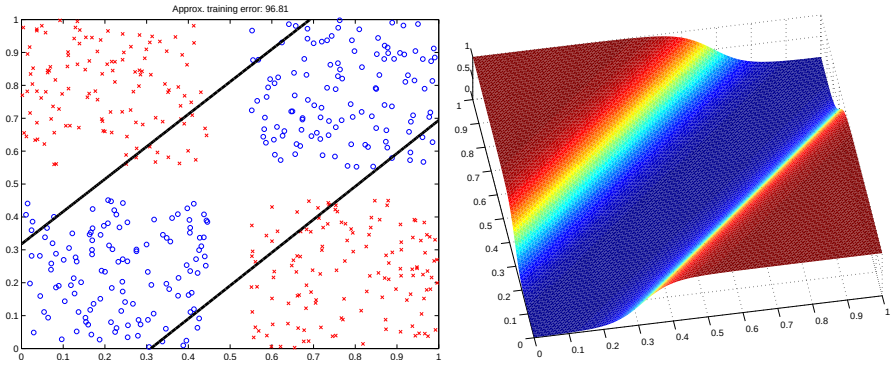

Figure 7: Globally optimal solution of the network from Fig. 2 applied to the XOR pattern. Left: the decision boundary, right: the discrimination function.

set is separable by 1 decision boundary, the XOR pattern is not.

The next step is to add 1 neuron to the hidden layer, which gives a 2-3-1 network. In that case the solution can look like that depicted in Fig. 9. The result seems to be almost perfect solution! All training points are classified correctly, the decision boundaries look like those that would be drawn by a human.

However, if you look at the solution in larger scale (see Fig. 10), you can see that the boundaries are more 'wild' than expected. Moreover, although the lower right corner of the training data points is full of red crosses, the neural network produces there another decision boundary and creates there an area that is classified as blue circles! Without any support in the training data!

This phenomenon can be observed in all types of machine learning models. A particular model has its own particular structural constraints that it imposes on the final form of the classifier. Only at the area that is covered by the training

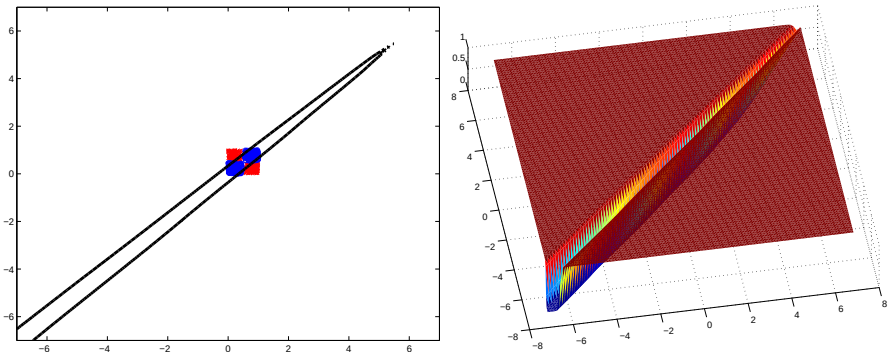

Figure 8: Globally optimal solution of the network in larger scale. Left: the decision boundary, right: the discrimination function.

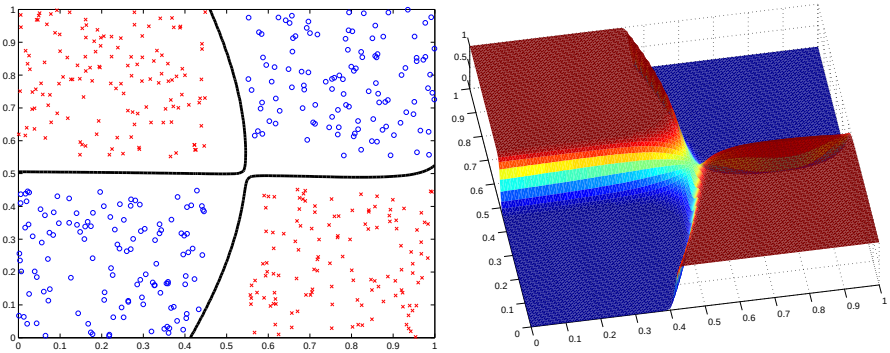

Figure 9: Solution of the 2-3-1 network for the XOR problem. Left: the decision boundary, right: the discrimination function.

data set, the predictions of the model can be influenced in a systematic way; in areas beyond the training set, each model extrapolates the values in its own way—similarly as would be done e.g. by 2nd and 3rd order polynomials when modeling one-dimensional function.

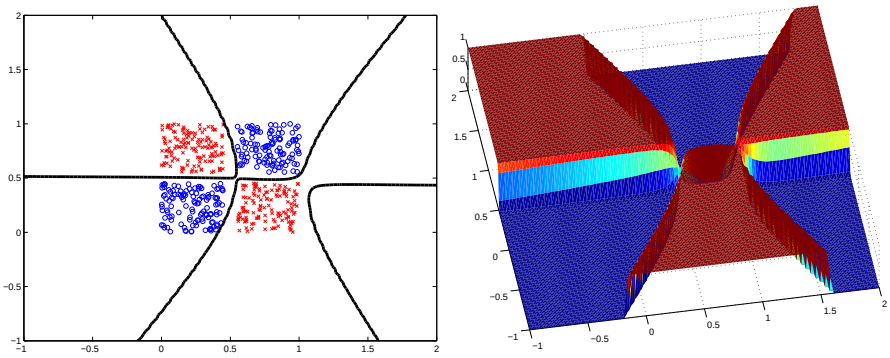

Figure 10: Solution of the 2-3-1 network for the XOR problem in larger scale. Left: the decision boundary, right: the discrimination function.