Notes on Deep Feedforward Network (MLP)

Dr. Hao Wang

Late May-Early June, 2016

The task is to construct a proper computing method to obtain the gradient needed for very much a graph that describes the interactions of the perceptrons, also widely known as multilayer perceptron. We visualise this type of deep feedforward network as shown in Figure 1. Given m training sample data, a final cost/loss function without regularization term can be expressed as

$$J_A = \frac{1}{m}\sum_{i=1}^{m} J(\boldsymbol{x}, y; \boldsymbol{W}, \boldsymbol{b}).$$

A gradient descendant method has one key step in that parameters like W is updated each iteration by

$$W^{\text{new}} = W^{\text{old}} - \alpha\frac{\partial J_A}{\partial W},$$

where α is usually termed as *learning rate*. Here the cost function takes a 2-norm squared form i.e.

$$J = \frac{1}{2}\left\|\hat{y}(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}) - y\right\|^2.$$

1 The Flow of Back Propagation

Before we compute the gradient using BP method, we need do a feedforward step by some randomly initialised weights and biases so that we have information about $\boldsymbol{a}^{(i)}$ and $\boldsymbol{h}^{(i)}$. The full flow of the backward propagation scheme is shown in Figure 2. Here, theoretically the chain rule can be applied without much pain provided we stick to the *numerator layout* convention for vector/matrix derivatives. (For general purpose, we encounter a 3rd order tensor in the derivatives, which can be mitigated using index computing. We shall see this in the next section.) Also here the case is for a single training sample. Therefore,

$$\begin{aligned}\frac{\partial J}{\partial \boldsymbol{W}^{(n_L)}} &= \frac{\partial J}{\partial \boldsymbol{h}^{(n_L)}}\frac{\partial \boldsymbol{h}^{(n_L)}}{\partial \boldsymbol{a}^{(n_L)}}\frac{\partial \boldsymbol{a}^{(n_L)}}{\partial \boldsymbol{W}^{(n_L)}} \\ \frac{\partial J}{\partial \boldsymbol{b}^{(n_L)}} &= \frac{\partial J}{\partial \boldsymbol{h}^{(n_L)}}\frac{\partial \boldsymbol{h}^{(n_L)}}{\partial \boldsymbol{a}^{(n_L)}}\frac{\partial \boldsymbol{a}^{(n_L)}}{\partial \boldsymbol{b}^{(n_L)}}\end{aligned}. \tag{1}$$

and similar results can be obtained for other terms.

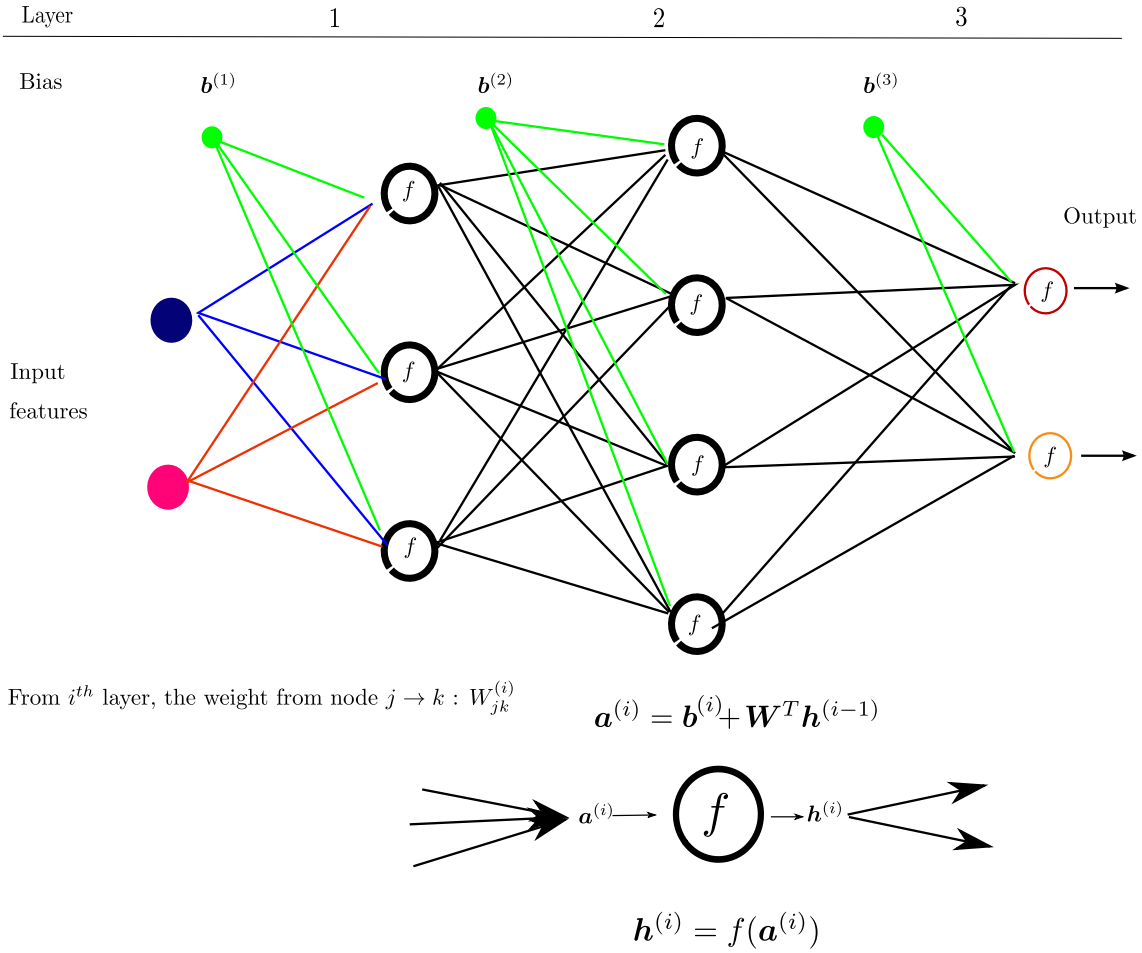

Figure 1 Illustration of deep feedforward networks

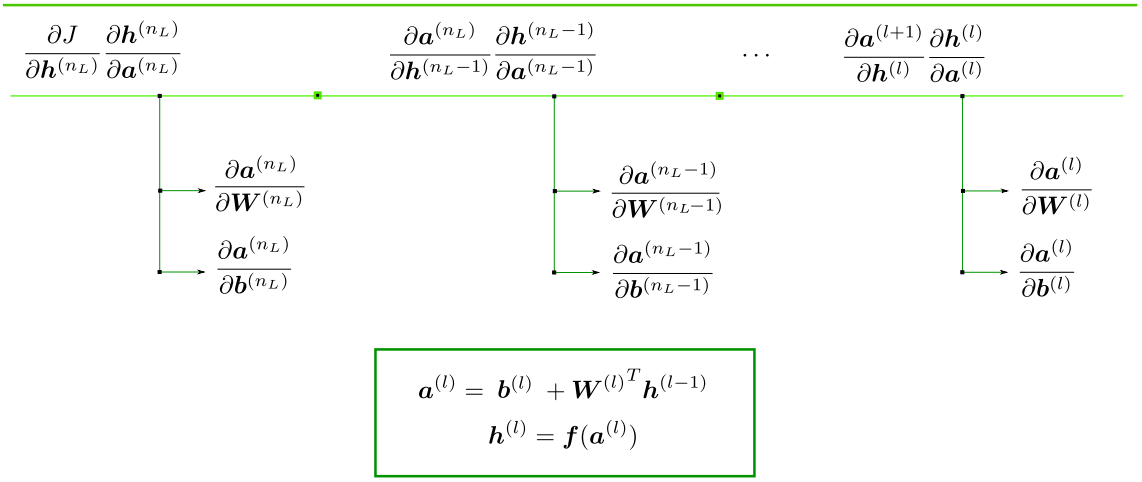

Figure 2 Illustration on how to compute the relevant gradients backwards, the squared nodes indicate multiplication action.

2 The Proposed BP Algorithm

First we compute a few terms occur in the above theoretic framework in Figure 2. Assume all the input and output are column vectors, and we adopt numerator layout convention for derivative computing and the indices in parentheses are indicator of the layer number. For i^{th} sample at the output layer, we have

$$\frac{\partial J}{\partial \boldsymbol{h}^{(n_L)}} = \left(\boldsymbol{h}^{(n_L)} - y_i\right)^T, \tag{2}$$

which by the way is a *row* vector.

Besides we know that the activation function $\boldsymbol{f}$ is acting element-wise, so

$$\frac{\partial \boldsymbol{h}^{(l)}}{\partial \boldsymbol{a}^{(l)}} = \mathrm{Diag}(\boldsymbol{f}'(\boldsymbol{a}^{(l)})) \tag{3}$$

Let us denonate $\boldsymbol{g}$ as the product of (2) and (3) (let $l = n_L$), then the computing can be simplified as

$$\boldsymbol{g} = \frac{\partial J}{\partial \boldsymbol{h}^{(n_L)}} \frac{\partial \boldsymbol{h}^{(n_L)}}{\partial \boldsymbol{a}^{(n_L)}} = \left(\boldsymbol{h}^{(n_L)} - y_i\right)^T \odot \boldsymbol{f}'(\boldsymbol{a}^{(n_L)})^T, \tag{4}$$

where operator $\odot$ represents element-wise multiplication, so $\boldsymbol{g}$ is also a row vector.

Using numerator convention, we can easily verify that

$$\frac{\partial \boldsymbol{a}^{(l+1)}}{\partial \boldsymbol{h}^{(l)}} = W^{(l+1)^T} \tag{5}$$

and

$$\frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{I}, \tag{6}$$

where $\boldsymbol{I}$ is the identity matrix. The slightly tricky bit is $\dfrac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{W}^{(l)}}$ since according to definition we end up with a 3^{rd} order tensor. For notational simplicity, we omit the upper script (l) and work on a general gradient term $\dfrac{\partial J}{\partial \boldsymbol{W}}$. With a proper row vector $\boldsymbol{g}$, we can have

$$\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{g}\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{W}}.$$

In the following, we use the summation convention, where the identical indices imply summation. So

$$\frac{\partial J}{\partial W_{jk}} = g_i \frac{\partial W_{qi} h_q}{\partial W_{jk}} = g_i h_q \delta_{qj} \delta_{ik} = g_k h_j, \tag{7}$$

transforming into a matrix form (also in numerator convention) we have

$$\frac{\partial J}{\partial W} = (\boldsymbol{h}\boldsymbol{g})^T.$$

($\boldsymbol{h}$ is a column vector, and for actual updating W, no need to transpose as shown in the algorithm.) The other identity:

$$\frac{\partial J}{\partial \boldsymbol{b}} = \boldsymbol{g}\boldsymbol{I} = \boldsymbol{g}.$$

Putting in column vector form, we can transpose $\boldsymbol{g}$. The algorithm can be constructed now as in Algorithm 1. (Note that all algorithm index starts with zero.)

Algorithm 1 Back propagation algorithm to compute gradients (single training sample)

Require: One training sample $(\boldsymbol{h}_i^{(0)}, y^i)$; Initialised $\boldsymbol{W}, \boldsymbol{b}$; Activation function $\boldsymbol{f}$
Ensure: Gradients $\partial J/\partial \boldsymbol{W}$ and $\partial J/\partial \boldsymbol{b}$

procedure BP PROCEDURE
  First run a feedforward pass, to get $\boldsymbol{a}$ and $\boldsymbol{h}$
  $\boldsymbol{g} := \left(\boldsymbol{h}^{(n_L-1)} - y^i\right)^T \odot \boldsymbol{f}'(\boldsymbol{a}^{(n_L-1)})^T$ ▷ *Starts from the output layer*
  $\partial J/\partial W^{(n_L)} := \boldsymbol{h}^{(n_L-1)}\boldsymbol{g}$
  $\partial J/\partial b^{(n_L)} := \boldsymbol{g}^T$
  for $l = n_L - 2, n_L - 2, \cdots, 0$ **do**
    $\boldsymbol{g} := \left(\boldsymbol{g}\left[W^{(l+1)}\right]^T\right) \odot \boldsymbol{f}'(\boldsymbol{a}^{(l)})^T$
    $\partial J/\partial \boldsymbol{W}^{(l)} := \boldsymbol{h}^{(l-1)}\boldsymbol{g}$
    $\partial J/\partial \boldsymbol{b}^{(l)} := \boldsymbol{g}^T$
  end for
end procedure

3 Gradient Descendant Algorithm

This section is try to incorporate the former BP Procedure to obtain the full version gradient descendant (optimization) algorithm. Everything derived so far is for **single training data** only. We might add a regularization term for the weights $W_{ij}^{(l)}$ to the loss/cost function J_A, so it becomes

$$J_A = \frac{1}{m}\sum_{i=1}^{m} J_i(\boldsymbol{x}, y; \boldsymbol{W}, \boldsymbol{b}) + \frac{\lambda}{2}\sum_{i,j,l}(W_{ij}^{(l)})^2. \tag{8}$$

The overall gradients $\partial J_A/\partial W$ is known by (the form breaks the numerator convention, no transpose, to simplify computation)

$$\frac{\partial J_A}{\partial W} = \frac{1}{m}\sum_{i}^{m} \nabla_W J_i + \lambda W. \tag{9}$$

The gradient descendant algorithm is shown in Algorithm 2.

4 Implementation Tricks: Not Reinventing the Wheels

The scikit-learn machine learning library suggests to call the standard scipy optimization library `lbfgs` (Limited memory BFGS) algorithm, which ought to perform better than purely gradient descendant, since Hessian information is employed. All left to do is to transform the deep learning (network) optimization problem to the conventional one.

In BFGS, we need two piece of information:

1. the underlying objective function $f(x)$,
2. the gradient to the objective function $\nabla f(x)$,

which are cost function J_A and derivatives (gradient) $\dfrac{\partial J_A}{\partial \boldsymbol{W}}$ and $\dfrac{\partial J_A}{\partial \boldsymbol{b}}$ under our notation. Next we deal with 1-feature(classifier) output problem, and the row vector $\boldsymbol{g}$ To account for all m training

Algorithm 2 Full gradients descendant algorithm

Require: All training samples $(\boldsymbol{x}, \boldsymbol{y})$; Initialised $\boldsymbol{W}, \boldsymbol{b}$; Activation function $\boldsymbol{f}$; Learning rate α, regularization parameter λ
Ensure: Trained weights and biases: $\boldsymbol{W}, \boldsymbol{b}$

procedure GRADIENT DESCENDANT PROCEDURE
  while Cost J_A is not small enough **do**
    Set $\Delta \boldsymbol{b}^{(l)} := \boldsymbol{0}, \quad \Delta \boldsymbol{W}^{(l)} := \boldsymbol{0}$ for all layers l
    for $l = 0, 2, \cdots, m-1$ **do**
      Call BP PROCEDURE to get $\partial J / \partial \boldsymbol{W}^{(l)}$ and $\partial J / \partial \boldsymbol{b}^{(l)}$
      $\Delta \boldsymbol{b}^{(l)} := \Delta \boldsymbol{b}^{(l)} + \partial J / \partial \boldsymbol{b}^{(l)}$
      $\Delta \boldsymbol{W}^{(l)} := \Delta \boldsymbol{W}^{(l)} + \partial J / \partial \boldsymbol{W}^{(l)}$
    end for
    $\boldsymbol{b}^{(l)} := \boldsymbol{b}^{(l)} - \dfrac{\alpha}{m} \Delta \boldsymbol{b}^{(l)}$
    $\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \alpha \left[\dfrac{1}{m} \Delta \boldsymbol{W}^{(l)} + \lambda \boldsymbol{W}^{(l)} \right]$
    Compute the current cost $J_A(\boldsymbol{W}^{(i)}, \boldsymbol{b}^{(l)})$
  end while
end procedure

data, we can arrange the input/output at each node as a $p \times m$ and $q \times m$ matrix.

$$\begin{aligned} {\boldsymbol{a}^{(l)}}_{q \times m} &= {\boldsymbol{b}^{(l)}}_{q \times 1} \boldsymbol{I}_{1 \times m} + {W^{(i)}}^{T}_{q \times p} {\boldsymbol{h}^{(i-1)}}_{p \times m} \\ {\boldsymbol{h}^{(l)}}_{q \times m} &= \boldsymbol{f}\left({\boldsymbol{a}^{(l)}}_{q \times m} \right) \end{aligned}, \tag{10}$$

where $\boldsymbol{I}_{1 \times m}$ is all-ones row vector. Cost function J_A is calculated by (8). After some careful manipulations we can have the following algorithm to compute $\dfrac{\partial J_A}{\partial \boldsymbol{W}}$ and $\dfrac{\partial J_A}{\partial \boldsymbol{b}}$:

Algorithm 3 Algorithm to compute gradients (all training sample)

Require: Training sample $(\boldsymbol{x}, \boldsymbol{y})$; Initialised $\boldsymbol{W}, \boldsymbol{b}$; Activation function $\boldsymbol{f}$
Ensure: Gradients $\partial J_A / \partial \boldsymbol{W}$ and $\partial J_A / \partial \boldsymbol{b}$

procedure REVISED BP PROCEDURE
  First run a feedforward pass based on (10)
  for $l = n_L - 1, n_L - 1, \cdots, 0$ **do**
    if $l == n_L - 1$ **then**
      $\boldsymbol{g} := \left(\boldsymbol{h}^{(l)} - \boldsymbol{y} \right)^T \odot \boldsymbol{f}'(\boldsymbol{a}^{(l)})^T$ ▷ *Starts from the output layer,* $\boldsymbol{g}$: $m \times q$
    else
      $\boldsymbol{g} := \left(\boldsymbol{g} \left[W^{(l+1)} \right]^T \right) \odot \boldsymbol{f}'(\boldsymbol{a}^{(l)})^T$ ▷ *for other layers,* $\boldsymbol{g} : q_l \times m$
    end if
    $\partial J_A / \partial W^{(l)} := \frac{1}{m} \boldsymbol{h}^{(l-1)} \boldsymbol{g} + \lambda W^{(l)}$
    $\partial J_A / \partial \boldsymbol{b}^{(l)} := \frac{1}{m} (\boldsymbol{I}\boldsymbol{g})^T$ ▷ $\boldsymbol{I} : 1 \times m$
  end for
end procedure

5 Experimental Run

The training dataset and test set are shown in Figure 3. The classified results is shown in Figure 4–Figure 8.

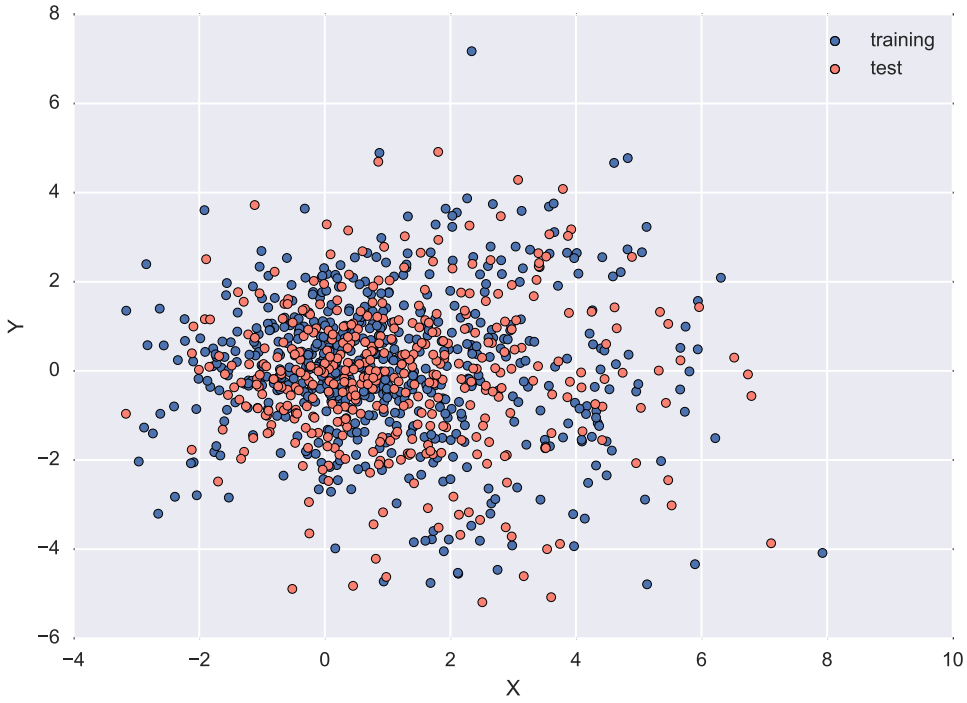

Figure 3 A Gaussian distributed data sample, divided by training and test set.

Figure 4 A trained network with two hidden layers $(100, 4)$, regularization parameter $\lambda = 0.01$

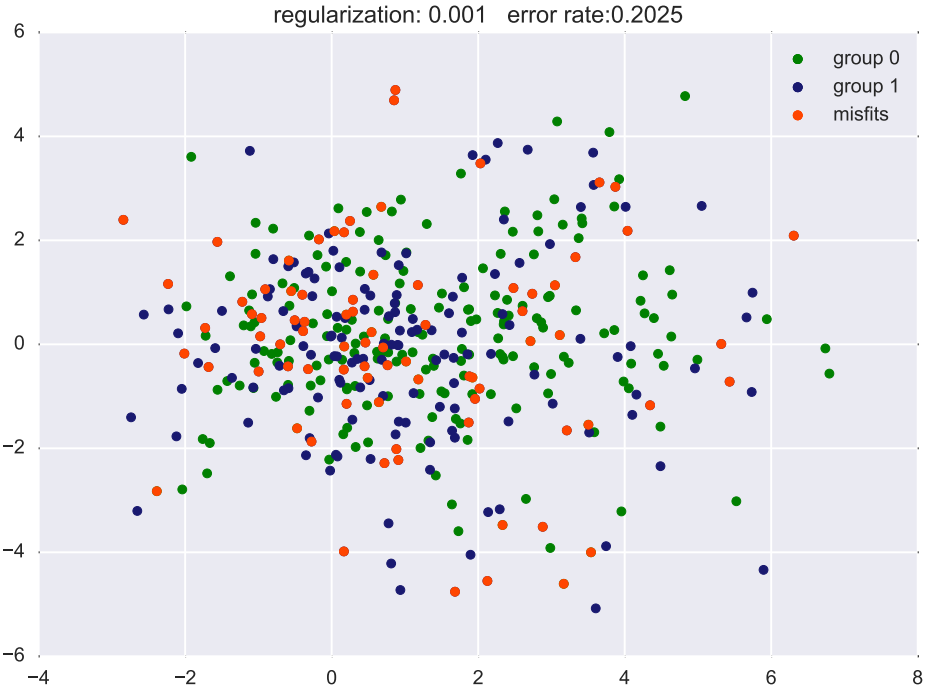

Figure 5 A trained network with two hidden layers $(100, 4)$, regularization parameter $\lambda = 0.001$

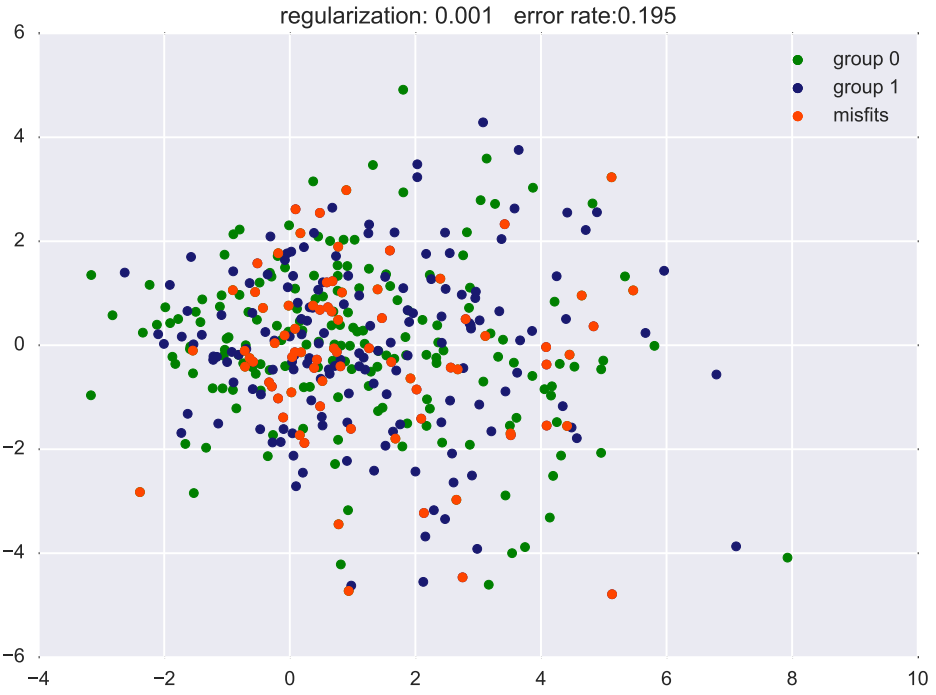

Figure 6 A trained network with two hidden layers $(10, 4)$, regularization parameter $\lambda = 0.001$

Figure 7 A trained network with two hidden layers $(10, 400)$, regularization parameter $\lambda = 0.001$

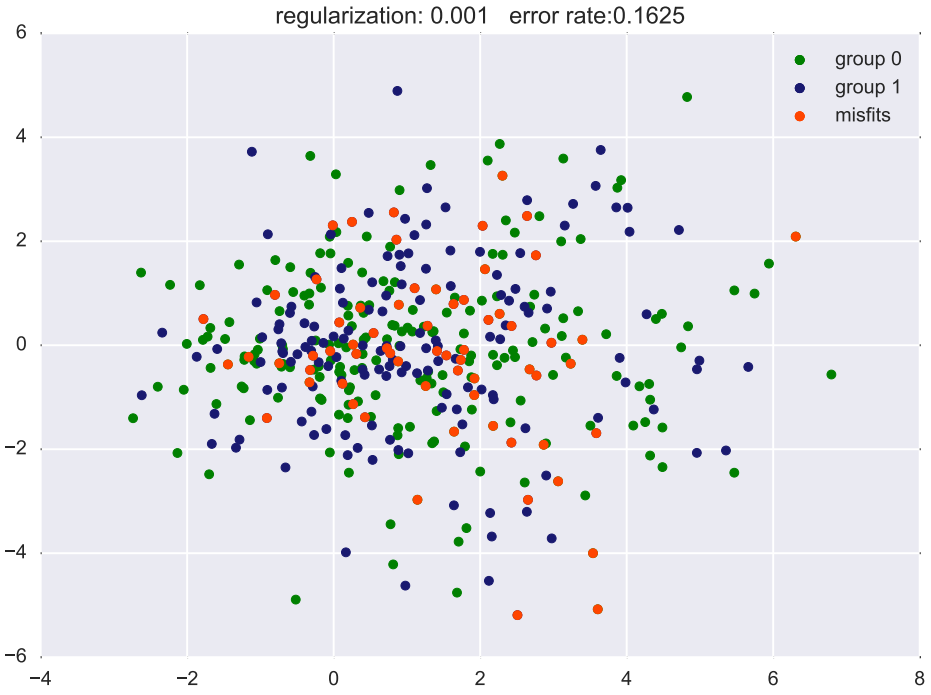

Figure 8 A trained network with three hidden layers $(10, 400, 2)$, regularization parameter $\lambda = 0.001$