Single Layer Neural Networks

Hiroshi Shimodaira[*]

10, 13 March 2015

We have shown that if we have a pattern classification problem in which each class c is modelled by a pdf $p(\mathbf{x}|c)$, then we can define discriminant functions $y_c(\mathbf{x})$ which define the decision regions and the boundaries between classes. If the classes have Gaussian pdfs and all share the same covariance matrix, then the discriminant functions are linear.

If we are concerned with performing pattern classification, we do not have to first estimate the pdfs for each class, and then derive the discriminant functions. We can find the discriminant functions directly. We will start by fitting linear discriminant functions. We will then move on to more complex discriminant functions, suitable for use with generative processes that would be hard to model directly. The technology we will use to represent the discriminant functions will be 'neural networks'.

The neural networks that we will consider are simply functions that take a d-dimensional input, and return an output (often a scalar). These functions have free-parameters, known as 'weights and biases'. Learning a neural network model entails fitting the weight and bias parameters so that the input-output mapping is useful for some task. In these notes, we will learn functions that give low classification error when used as discriminant functions.

1   Network representation

Consider the set of linear discriminant functions for a K class problem:

$$
\begin{aligned}
y_1(\mathbf{x}) &= \mathbf{w}_1^T \mathbf{x} + w_{10}, \\
&\vdots \\
y_K(\mathbf{x}) &= \mathbf{w}_K^T \mathbf{x} + w_{K0},
\end{aligned}
\tag{1}
$$

This can be converted into a matrix-vector form:

$$
\begin{pmatrix} y_1 \\ \vdots \\ y_K \end{pmatrix} = \begin{pmatrix} w_{10} & w_{11} & \dots & w_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K0} & w_{K1} & \dots & w_{Kd} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{pmatrix}
\tag{2}
$$

or

$$
\mathbf{y} = W\dot{\mathbf{x}},
\tag{3}
$$

where $y_c = y_c(\mathbf{x})$, $\mathbf{y} = (y_1, \dots, y_K)^T$, $W = (\dot{\mathbf{w}}_1, \dots, \dot{\mathbf{w}}_K)^T$, $\dot{\mathbf{w}}_c = (w_{c0}, w_{c1}, \dots, w_{cd})^T = (w_{c0}, \mathbf{w}_c^T)^T$, and $\dot{\mathbf{x}} = (1, \mathbf{x}^T)^T$, which is an augmented vector of $\mathbf{x}$. Hereafter, we sometimes use $\mathbf{x}$ to denote $\dot{\mathbf{x}}$ for simplicity.

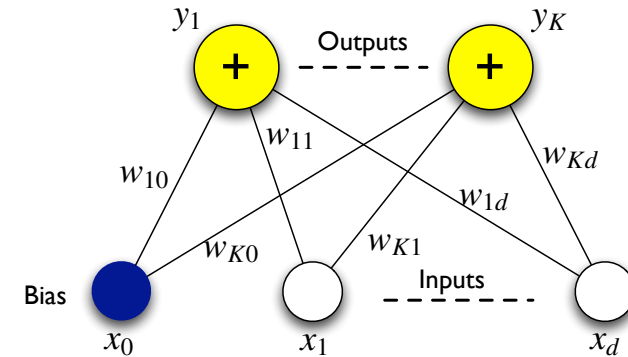[*]Heavily based on notes inherited from Steve Renals and Iain Murray.

Figure 1: Single layer neural network with d-dimensional input vector $\mathbf{x} = (x_1, \dots, x_d)^T$ and output vector $\mathbf{y} = (y_1, \dots, y_K)^T$ corresponding to K classes. The input vector is augmented with an additional variable $x_0 = 1$ which is the bias node. The model is parameterised by the weight matrix $\mathbf{W}$, whose elements w_{ki} are the weights from input x_i to output y_k (and whose rows are the discriminant functions $\mathbf{w}_k$). The bias of output y_k is given by w_{k0}.

We can regard this mapping from $\mathbf{x}$ to $\mathbf{y}$ as a *single-layer neural network*, in which the weight vector for class c_k connects the input vector ($\mathbf{x}$) to an output corresponding to class c_k. We can collect the weight vectors together as the rows of a $K \times (d + 1)$ weight matrix $\mathbf{W}$, in which element w_{ki} connects input x_i to output y_k. To avoid treating the biases w_{k0} separately, we define an additional input x_0, which always takes the value 1 (hence we have a $(d + 1)$ dimension input vector). The bias for class k, w_{k0}, is the weight between x_0 and y_k. The resulting network is illustrated in figure 1.

The equation of such a single-layer neural network (in matrix-vector form) is:

$$
\mathbf{y} = \mathbf{W}\mathbf{x}
\tag{4}
$$

or, in terms of the individual components:

$$
y_k = \sum_{i=0}^{d} w_{ki} x_i.
$$

We can summarise the terminology we have just introduced:

Input vector $\mathbf{x} = (x_0, x_1, \dots, x_d)^T$

Output vector $\mathbf{y} = (y_1, \dots, y_K)^T$

Weight matrix $\mathbf{W}$: w_{ki} is the weight from input x_i to output y_k

2   The training problem

How do we train the weight matrix in a single layer neural network defined by equation (4)?

The network is trained using a *training set* that contains N input/output pairs $\{(\mathbf{x}_n, \mathbf{t}_n) : 1 \le n \le N\}$, where $\mathbf{t}_n = (t_{n1}, \ldots, t_{nK})$ is the *target output* vector for input vector $\mathbf{x}_n$. For a classification problem, if the correct class is c_k, then:

$$t_{nk} = 1$$
$$t_{n\ell} = 0 \qquad \forall \ell \ne c .$$

Representing the target label in a vector in this way is called a "1-from-K" coding.

The single-layer neural network training problem may be stated as follows:

Given a training set, what values should the elements of the weight matrix $\mathbf{W}$ take, so as to *minimise* an *error function* defined in terms of the outputs $\mathbf{y}_n$?

The error function should measure the distance of the output vectors $\mathbf{y}_n$ from the corresponding target vectors $\mathbf{t}_n$ for all n. A natural way to do this is by taking the (squared) Euclidean distance, and we define the *sum-of-squares* error function which computes the sum of squared Euclidean distances between $\mathbf{t}_n$ and $\mathbf{y}_n$ for all members of the training set $1 \le n \le N$. In matrix form we can write:

$$E(\mathbf{W}) = \frac{1}{2} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{t}_n\|^2$$

Or equivalently, we can write the error in terms of the components:

$$E(\mathbf{W}) = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} (y_{nk} - t_{nk})^2$$
$$= \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(\sum_{i=0}^{d} w_{ki} x_{ni} - t_{nk} \right)^2 . \tag{5}$$

This error function $E(\mathbf{W})$ is a smooth function of the weights. We train the network parameters $\mathbf{W}$ so as to minimise the value of the error function $E(\mathbf{W})$ given the training set.

To find the minimum of a function we look for the point where its gradient is 0. In this case we look for where the derivative of E with respect to the weights equals 0. Since E is a quadratic function of the weights, the derivatives will be linear functions of the weights. There are two ways of solving the equation to find the optimal weight matrix:

- An exact approach (*pseudoinverse* solution)

- Iterative approaches such as
  - Iteratively reweighted least squares (IRLS), which is an application of the Newton–Raphson algorithm
  - Gradient descent

We will consider *gradient descent*: although IRLS is preferable for single-layer neural networks (it is much faster), gradient descent can also be used in more complicated settings (such as training multi-layer neural networks).

3   Gradient descent

Gradient descent is an important optimisation technique, that may be used whenever it is possible to compute the derivatives of the error function with respect to the parameter to be optimised. For single layer neural networks, this means taking the derivative of the error function $E(\mathbf{W})$ with respect to the weight matrix $\mathbf{W}$.

The idea of gradient descent is that to minimise an error function with respect to the parameters, we want to take small steps in a *downhill* direction. We take small steps because the gradient is not uniform, and if we take too big a step we may end up going uphill again! When considering this form of optimisation, we are considering another multidimensional space, *weight space*. This is a $K \cdot (d + 1)$ dimension space, and a specific weight matrix $\mathbf{W}$ corresponds to a point in weight space. The error function evaluates the error value for a point in weight space (given the training set).

The gradient of E given $\mathbf{W}$ is written as $\nabla_{\mathbf{W}} E$, the vector of partial derivatives of E with respect to the elements of $\mathbf{W}$:

$$\nabla_{\mathbf{W}} E = \left(\frac{\partial E}{\partial w_{10}}, \ldots, \frac{\partial E}{\partial w_{ki}}, \ldots, \frac{\partial E}{\partial w_{Kd}} \right)^T .$$

Descending in weight space means adjusting the weight matrix $\mathbf{W}$ by moving a small direction down the gradient, which is the direction along which E decreases most rapidly. This means adjusting the weight factor in the direction of $-\nabla_{\mathbf{W}} E$, or adjusting each weight w_{ki} by adding a factor $-\eta \cdot \partial E / \partial w_{ki}$, where η is a small constant called the *step size* or *learning rate*.

The operation of gradient descent is as follows:

1. Start with a guess for the weight matrix $\mathbf{W}$ (e.g., small randomly chosen weights).

2. Update the weights by adjusting the weight matrix by a small distance in the direction in weight space along which E decreases most rapidly: i.e., in the direction of $-\nabla_{\mathbf{W}} E$.

3. Recompute the error, and goto 2, terminating either after a fixed number of steps, or when the error stops decreasing by more than a threshold.

If we write the value of a weight at iteration τ as w_{ki}^{τ}, then its updated value is given by:

$$w_{ki}^{\tau+1} = w_{ki}^{\tau} - \eta \frac{\partial E}{\partial w_{ki}} . \tag{6}$$

The learning rate η specifies how much the parameters should be adjusted along the direction of the gradient.

4   Gradient descent for a single-layer neural network

To apply gradient descent to a single-layer neural network, in which we minimise E with respect to $\mathbf{W}$, it is necessary to differentiate E (equation (5) with respect to each weight w_{ki}:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \left(\sum_{j=0}^{d} w_{kj} x_{nj} - t_{nk} \right) x_{ni}$$
$$= \sum_{n=1}^{N} (y_{nk} - t_{nk}) \, x_{ni} .$$

Define δ_{nk} as the difference between the network output and the target for class k [1], considering the n'th example:

$$\delta_{nk} = y_{nk} - t_{nk}.$$

We can think of δ_{nk} as the error on output k for the n'th training example.

We may thus write the partial derivatives in terms of the δs:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \delta_{nk} x_{ni}. \tag{7}$$

So the derivative for the weight connecting input i to output k is calculated using the product of the error at output k, δ_{nk}, and the input value x_{ni}, summed over all the training set. This derivative is 'local' to the weight concerned: it does not require knowledge of other weights in the weight matrix, nor other elements of the input, output or target vectors.[2]

Combining the expression for the derivatives (7) with the expression for gradient descent update (6), we obtain:

$$w_{ki}^{\tau+1} = w_{ki}^{\tau} - \eta \sum_{n=1}^{N} \delta_{nk} x_{ni}. \tag{8}$$

This is the gradient descent *learning rule* for the weights of a single-layer neural network. (This gradient descent rule is also known as the delta rule, the Widrow–Hoff rule and LMS learning.)

We may write the algorithm for gradient descent training as follows:

```
 1: procedure GRADIENTDESCENTTRAINING(X, T, W)
 2:     initialise W to small random numbers
 3:     while not converged do
 4:         for all k, i: Δw_ki ← 0
 5:         for n ← 1, N do
 6:             for k ← 1, K do
 7:                 y_nk ← Σ_{i=0}^{d} w_ki x_ni
 8:                 δ_nk ← y_nk − t_nk
 9:                 for i ← 1, d do
10:                     Δw_ki ← Δw_ki + δ_nk · x_ni
11:                 end for
12:             end for
13:         end for
14:         for all k, i: w_ki ← w_ki − η · Δw_ki
15:     end while
16: end procedure
```

4.1 The bias parameter

We previously saw a geometric interpretation of the bias parameter for a two-class linear discriminant: the normal distance of the separating hyperplane from the origin. Here we provide another interpretation of the bias.

---

[1] Note that δ_{nk} here is not the Kronecker delta!

[2] Local computations are convenient and scale well. The locality of computation has also been used to argue for the 'biological plausibility' of neural network models of learning. Although any relation to biological neurons is abstract; real neurons are complicated.

If we set the derivatives to 0, as above, and explicitly consider only the bias parameter, then we can write:

$$\frac{\partial E}{\partial w_{k0}} = \sum_{n=1}^{N}\left(\sum_{j=1}^{d} w_{kj} x_{nj} + w_{k0} - t_{nk}\right) = 0.$$

If we write the average inputs and targets as follows:

$$\bar{x}_i = \frac{1}{N}\sum_{n=1}^{N} x_{ni}$$

$$\bar{t}_k = \frac{1}{N}\sum_{n=1}^{N} t_{nk}.$$

Then we may write the solution for the bias as

$$\sum_{j=1}^{d} w_{kj}\bar{x}_j + w_{k0} - \bar{t}_k = 0$$

$$w_{k0} = \bar{t}_k - \sum_{j=1}^{d} w_{kj}\bar{x}_j.$$

This means that we may interpret the bias as compensating for the difference in the mean of the targets and the network outputs, averaged over the training set.

5 Logistic discriminants

We can generalise linear discriminants by applying a nonlinear function to them. Consider a two-class linear discriminant, to which we apply an *activation function*, g:

$$y(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + w_0). \tag{9}$$

If g is monotonically increasing, then (9) may be regarded as a linear discriminant, since the decision boundary will still be linear. We may represent this as a two-class, single output, single layer neural network (figure 2).

For convenience we can define an *activation value* a:

$$y(\mathbf{x}) = g(a)$$

$$a = \mathbf{w}^T\mathbf{x} + w_0.$$

5.1 Logistic sigmoid activation function

If we have a two class problem, with classes c_1 and c_2, then we can express the posterior probability of c_1 using Bayes' theorem:

$$P(c_1|\mathbf{x}) = \frac{p(\mathbf{x}|c_1)\,P(c_1)}{p(\mathbf{x}|c_1)\,P(c_1) + p(\mathbf{x}|c_2)\,P(c_2)}.$$

If we divide top and bottom of the right hand side by $p(\mathbf{x}|c_1)\,P(c_1)$, then we obtain:

$$P(c_1|\mathbf{x}) = \frac{1}{1 + \frac{p(\mathbf{x}|c_2)\,P(c_2)}{p(\mathbf{x}|c_1)\,P(c_1)}}. \tag{10}$$
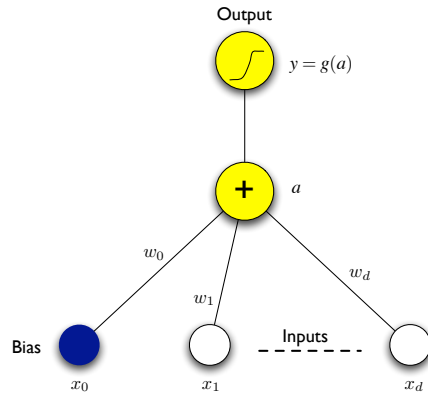
Figure 2: Single layer neural network representation of a generalised linear discriminant. In this case the output activation function is a logistic sigmoid.

If we define a as the ratio of log posterior probabilities (log odds):

$$a = \ln \frac{p(\mathbf{x}|c_1)\,P(c_1)}{p(\mathbf{x}|c_2)\,P(c_2)} \tag{11}$$

and substitute into (10) we obtain:

$$P(c_1|\mathbf{x}) = g(a) = \frac{1}{1 + \exp(-a)}\,. \tag{12}$$

Here, $g(a)$ is the *logistic sigmoid activation function*, plotted in figure 3. Sigmoid means 'S'-shaped: the function maps $(-\infty, \infty)$ onto $(0, 1)$ — it is a "squashing function". If $|a|$ is small then $g(a)$ is approximately linear: so a network with logistic sigmoid activation functions approximates a linear network when the weights (and hence the inputs to the activation function) are small. As a increases, $g(a)$ saturates to 1, and as a decreases to become large and negative $g(a)$ saturates to 0.

For a single layer neural network:

$$a = \mathbf{w}^T\mathbf{x} + w_0\,. \tag{13}$$

If we have a single-layer neural network, with one output, and a logistic sigmoid activation function g on the output node (9), then from (12) and (13) we see that the posterior probability may be written:

$$P(c_1|\mathbf{x}) = g(a) = g(\mathbf{w}^T\mathbf{x} + w_0)\,.$$

This corresponds to the single layer neural network of equation (9) and figure 2.

Therefore, for a two class problem (which may be represented with a single output), a single layer neural network with a logistic sigmoid activation function on the output may be regarded as providing a posterior probability estimate. This is an interesting result, since it means we may obtain a posterior probability estimate by training the weights of a single-layer neural network by gradient descent, with no need to estimate (or assume) Gaussian pdfs. The optimal setting of the weights of such a network, given a training set, need not be the same as those obtained using a Gaussian classifier with a shared covariance matrix.

This single layer neural network, giving $P(c_1|\mathbf{x}) = 1/(1 + \exp(-\mathbf{w}^T\mathbf{x} - w_0))$, is known as the *logistic regression* binary classifier. Logistic regression is a 'work-horse' of practical machine learning: it's widely used, and important to understand.
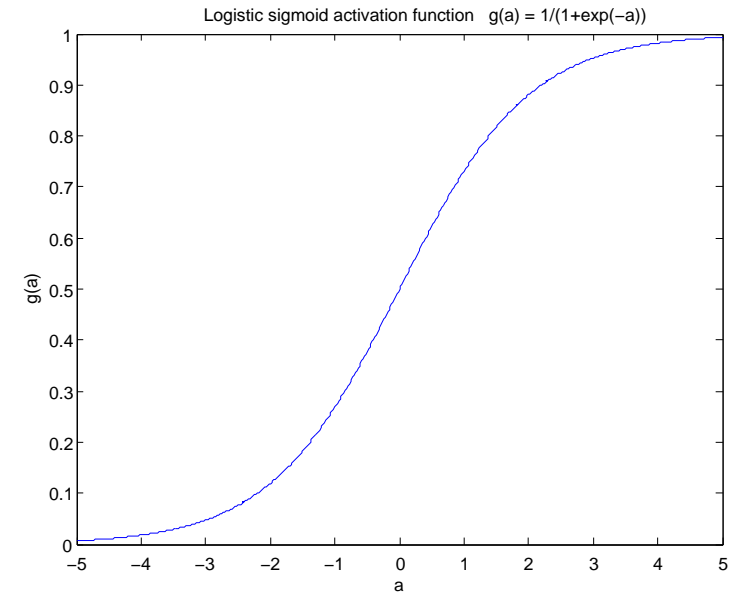
Figure 3: Logistic sigmoid function, $g(a) = \dfrac{1}{1 + e^{-a}}$

5.2 Gradient descent training

Gradient descent training of a single-layer neural network with a logistic sigmoid activation function is similar to training a linear network. The principal difference is that we must take of the contribution of the activation function to the gradient.

Consider a network with sigmoid activation functions on the outputs. We can write the "forward" equations of the network as:

$$y_{nk} = g(a_{nk})$$

$$a_{nk} = \sum_{i=0}^{d} w_{ki}x_{ni}\,.$$

In this case, the sum-of-squares error function is given by:

$$E = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(y_{nk} - t_{nk})^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(g(a_{nk}) - t_{nk})^2\,.$$

The partial derivative of the error function with respect to weight w_{ki} for pattern n is:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial E_n}{\partial y_{nk}} \frac{\partial y_{nk}}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial w_{ki}}$$

$$= \delta_{nk} \, g'(a_{nk}) \, x_{ni}$$

where $\delta_{nk} = (y_{nk} - t_{nk})$, as before. $g'(a)$ is the derivative of the sigmoid activation function. It turns out that

$$g'(a) = g(a)(1 - g(a)) \,.$$

(You should check this!)

Therefore the total gradient is:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E_n}{\partial w_{ki}}$$

$$= \sum_{n=1}^{N} g'(a_{nk}) \, \delta_{nk} \, x_{ni} \,.$$

6 Softmax

If we have more than two classes then a suitable discriminant function for class c_k is related to the log posterior probability:

$$a_k = \ln p(\mathbf{x}|c_k) \, P(c_k) \,,$$

where a_k is called the activation value of output k.

If we substitute the activation into Bayes' theorem, we find that the posterior probability is given by the following expression:

$$P(c_k|\mathbf{x}) = \frac{\exp(a_k)}{\sum_{\ell=1}^{K} \exp(a_\ell)} \,,$$

which is sometimes referred to as the *softmax* or normalised exponential.

We can use the softmax as the output function for a multi-class single layer neural network:

$$y_k = \frac{\exp(a_k)}{\sum_\ell \exp a_\ell}$$

$$a_k = \sum_{i=0}^{d} w_{ki} x_i \,.$$

This form of output function guarantees that the K output values will sum to 1, a necessary condition for probability estimates.

We can perform gradient descent training when there is a softmax activation function on the outputs. We have to be a little more careful when estimating the derivatives, since the output y_k depends on the other outputs y_ℓ, through the normalisation term in the denominator. Therefore we need to consider the

partial derivative of y_k with respect to all the activation values a_ℓ when computing the gradient:

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial E_n}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial w_{ki}}$$

$$\frac{\partial E_n}{\partial a_{nk}} = \sum_{\ell=1}^{K} \frac{\partial E_n}{\partial y_{n\ell}} \frac{\partial y_{n\ell}}{\partial a_{nk}}$$

$$= \sum_{\ell=1}^{K} \delta_{n\ell} \frac{\partial y_{n\ell}}{\partial a_{nk}}$$

$\partial E_n/\partial y_{n\ell}$ equals δ_{nk}, as before. The other derivative turns out to be:

$$\frac{\partial y_{n\ell}}{\partial a_{nk}} = y_\ell I_{k\ell} - y_\ell y_k$$

where $I_{k\ell} = 1$ if $k = \ell$, and $I_{k\ell} = 0$ otherwise.

7 "Online" gradient descent

The error function is usually calculated by summing over all input patterns, where E_n is the contribution to the error from a single pattern:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \,.$$

It is possible to use the gradient of E_n to update the weights one pattern at a time:

$$w_{ki}^{(\tau+1)} = w_{ki}^{(\tau)} - \eta \frac{\partial E_n}{\partial w_{ki}} \,. \tag{14}$$

This is referred to as *online gradient descent*. It is usually preferable to choose patterns randomly when training online. In *stochastic gradient descent* a training pattern is randomly chosen, the error and its derivatives computed for that pattern, and the weights are updated using (14). Online gradient descent can be useful for real-time, adaptive applications. Stochastic gradient descent is an efficient procedure when using large data sets.

8 Matlab: Training single layer neural networks

It is straightforward to use single-layer neural networks in Matlab, using the Netlab toolbox. To define a single layer neural network, with 2 inputs, 2 outputs and a linear output function, use the function `glm`:

```
slnet = glm(2,2,'linear');
```

To train this network with gradient descent, we need to set a few options (look at the help page on `graddesc` to learn more):

```
options=foptions;
options(1)=1; %% display/log error values
options(3)=0.01; %% convergence criterion
options(14)=100; %% maximum number of training iterations
options(18)=0.001; %% learning rate (eta)
```

Training a network in Netlab is done using the netopt function which takes the specific training algorithm to be used as an argument. To train this net using gradient descent, where `xtrain` is the matrix of training data, and `ttrain` contains the corresponding targets, one row per pattern.

```
[slnet options errlog] =
      netopt(slnet, options, xtrain, ttrain, 'graddesc');
plot(errlog);
```

For comparison try using the IRLS trainer (`glmtrain`).

The second statement plots the graph of the error versus learning rate. Gradient descent is sensitive to the value of the learning rate: if it is too high then the step down the gradient is too big, and the error function can increase instead of decreasing, and is likely to become unstable, increasing rapidly in later iterations. On the other hand, if the step size is too small, the error rate will decrease more slowly than possible.

8.1 Testing

Once the network is trained, the function `glmfwd(net,x)` can be used to run the test data `xtest` through the network:

```
testOut=glmfwd(slnet,xtest);
```

If `ttest` contains the targets for the test data (i.e., the correct classification), the following piece of Matlab can be used to compute the classification error:

```
    [maxOut, classified] = max(testOut,[],2);
    [tmpOut, answerClasses] = max(ttest,[],2);
    numberMisclassified = size(find(classified-answerClasses));
    percentError = 100.0 * numberMisclassified / size(answerClasses);
```

MATLAB tricks:

- `max(testOut,[],2);` means work along dimension 2 — i.e., returns the maximum value for each row.

- `[maxOut, classified] = max(testOut,[],2);` returns the max values for each row in `maxOut`, and the index in `classified`.

- `find(classified-answerClasses)` returns the indices of non-zero elements (i.e., elements where the output class is different to the target class).

- `size(find(classified-answerClasses))` thus returns the number of misclassified patterns.

To train a single-layer neural network with a logistic sigmoid activation function in Matlab. If `xtrain` and `ttrain` contain the input vectors and targets, then:

```
options=foptions;
options(1)=1;
options(18)=0.01;
```

```
ttrain = [ta;tb];

slnet=glm(2,1,'logistic')
[slnet options errlog] =
      netopt(slnet, options, xtrain, ttrain, 'graddesc');
```

For the softmax activation function, the third argument to glm should be `'softmax'`.

Technical note: (For a good reason) Netlab uses a different error function for logistic sigmoid output compared with linear output. This means that the error values produced by `netopt` for the logistic and linear activation functions are not comparable.

8.2 Tasks: Single Layer neural networks

The data for this lab consists of two files `ecoli-train.netlab` and `ecoli-test.netlab`; these are available from:
http://www.inf.ed.ac.uk/teaching/courses/inf2b/labs/ecoli-train.netlab
http://www.inf.ed.ac.uk/teaching/courses/inf2b/labs/ecoli-test.netlab

The lab is concerned with classifying a real data set. The data consists of 5 real-valued inputs and a discrete class. There are four classes (numbered 1–4). There are 230 training patterns in total (107 for class 1, 58 for class 2, 39 for class 3 and 26 for class 4). There are also 77 test patterns. The data is to do with molecular biology, but for this example you don't need to be concerned with that (for information the original README file that corresponds to the data set is available on the lab3 web page). The data has been reformatted into a form suitable for Netlab. Download the files `ecoli-train.netlab` and `ecoli-test.netlab`, and use the method `datread` to load them, e.g.:

```
[ecxtrain, ecttrain, nin, nout, ndata] = datread('ecoli-train.netlab');
[ecxtest, ecttest, nin, nout, ndata] = datread('ecoli-test.netlab');
```

Carry out the following tasks.

1. Train a single-layer neural network on this data using gradient descent. Observe how the error decreases per iteration. Experiment with various learning rates. In each case compute the error rate on the test set.

2. Compare IRLS training with gradient descent training.

3. Repeat the experiments using a single layer neural network with a softmax output activation function.

9 Summary

This chapter has introduced several important concepts:

- The representation of a set of discriminant functions as a single-layer neural network.

- Direct training of the parameters of a single-layer neural network.

- Minimisation of error function by gradient descent in parameter space.

- The logistic sigmoid and softmax activation functions, and their relation to posterior probabilities.

- Online or stochastic gradient descent.