Pattern Classification and Machine Learning

Benoit Rat[1], Jean-Christophe Fillion-Robin[2]
Communication Systems Faculty
EPFL Lausanne
CH-1015 Lausanne, Switzerland

***Abstract* — Using ..., Combining ... This paper highlights and contextualizes our implementation and outlines the crux of the problem in order to motivate future related research.**

I. Introduction & motivation

Pattern classification ....

II. Database

For this work we have started using the segmentation images database. As it was really difficult to observe relevant overfitting for the back-propagation algorithm even using few samples in the training, we decided to use "Wisconsin Diagnostic Breast Cancer data base" (WDBC) which gives more interesting result for a "theoretical flavor" study.

The WDBC data base contains 569 samples. Each of them has an ID, a class type tagged with some medical expertise and 30 different features. Our goal is to know if the subject is malignant (M) or benign (B) using the 30 preprocessed features. We are not going to describe the meaning of all the 30 features since they are available in the references [1].

To perform the classification we separated our database in 3 sets: Training, Validation, Testing. We first processed each line to fit the *MatDatSet* structure: (removing ID field and comma, changing the class type to {0,1} and moving it to the end of the row). Then we took one third of the database to built the testing set which is not changed after. Finally, we designed and implemented a C++ program generating new validation and training set by randomly permuting the two third of samples that left. We also have been cautious to respect more or less the proportion of benign and malignant: $|B|/|M| \approx 1.7$. This random permutation is, in some extent, a sort of cross validation between training and testing when we use more than one iteration over this two sets randomly picked.

III. Back-Propagation

A. Description & Structure.

Back-Propagation in an algorithm which corresponds to a gradient descent on a multi-layer perceptron. As we use the structure of an MLP, we still want to minimize the error function, which is in our case the mean square error (MSE) between the target and the output.

We first focus on the structure of the MLP we should use: This can be done by setting the number of hidden layers and the number of hidden neurones in each layers. In order to connect the different neurones from a layer to another we use a linear weighted function. Then the hidden neurones, and outputs neurones have a transfer function (step, linear, sigmoid or tanh). As we want our output to be smooth and continuous values between 0 and 1, we use the Sigmoid transfer function $g(h)$.

Moreover, the BackProp algorithm has a learning rate parameter which corresponds to the steps of a gradient descent. Finally the criterion to stop the algorithm is given by the final MSE to reach (aka. *end_accuracy*) or in the worst case a maximum number of iterations.

B. Playing with the different parameters.

During the first part of our research on the MLP, we tried different parameters against each others. This is a difficult task to do due to their important numbers. We first set a very low *end_accuracy*: 10^{-6} and the maximum number of iterations to 10000. Then we try with coarse values for the number of hidden units and the learning rate in order to observe the influence of using 1,2 or 3 hidden layers. No significant improvements were observed whereas the computational time has been increased, so we decide to use only one hidden layer.

Afterwards, to obtain the overfitting plot we run our algorithm with different number of hidden neurones {1,5,10,15,25,35,50,100} and different learning rate {0.1,0.01,0.001,0.0001}. We first denote that a learning rate of 0.1 is too high because the learning curbs oscillates until converging to a high MSE value for both training and validation set. We also denote that taking a really small learning rate increases the convergence speed such as the presence of a high number of hidden neurones. Moreover the minimum value of the MSE on the validation is not better than when we use an higher learning rate. Therefore we decided to focus our experiments using the learning rates 0.01 and 0.001.

Report written June 13th 2007.

1. e-mail: benoit.rat@epfl.ch
2. e-mail: jean-christophe.fillion-robin@epfl.ch

We noticed that adding more hidden units give us better overfitting. Finally, after 20 experiments with randomly built set both training and validation, the best compromise between good overfitting curbs and lowest MSE is obtained using the a learning rate = 0.001 and a number of hidden neurones equal to 50.

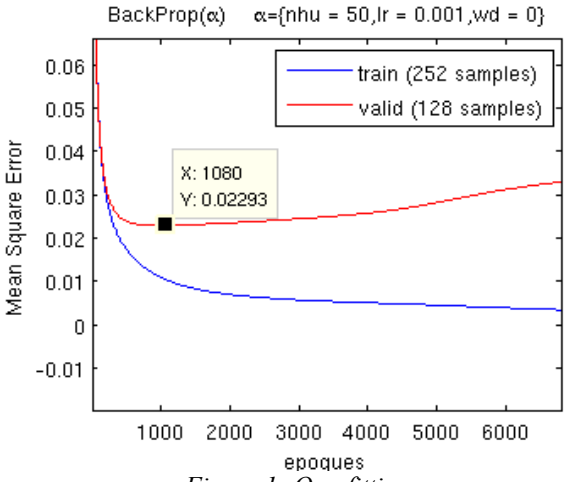
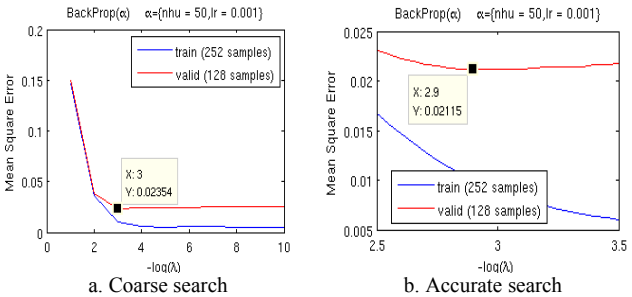


Figure 1: Overfitting.

C. Regularization using weight decay.

Once we have set these different parameters, we notice that, due to the overfitting, the final MSE for validation is higher than the MSE error at epoch 1080 (See Figure 1). In order to compensate overfitting we use a regularization method to control the flexibility of the network.

We test different values of the weight decay λ using a log step in order to find the optimal one. 20 different iterations with random initialization has been used to obtain smooth curbs. The search of the optimal value λ has been started using coarse step and has given its best result around 10^{-3}. Then using a fine step we obtain the optimal value $\lambda = 10^{-2.90} = 0.0013$.



a. Coarse search b. Accurate search

Figure 2: Regularization graph (optimal weight decay)

The final classification curve are showed in Figure 3 :

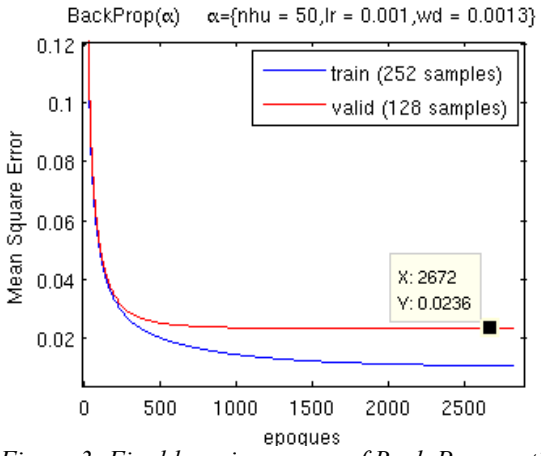


Figure 3: Final learning curves of Back-Propagation

	Training	Validation	Testing
MSE error	0.00552671	0.0236324	0.0238956
Classification error	0.3967%	2,3438%	2.6455%

Table 1: Final result of Back-Propagation

IV. Support Vector Machine

A. Description & Structure.

In order to deal with the fact that the perceptron is not linearly separable we introduce the Support Vector Machine (SVM). The SVM is an optimal perceptron which maximize the security margin using a kernel to project the input space into a linear separable space. The main difficulties in the SVM is to find the best kernel to perform this "linearization".

In this work, we are asked to use a Gaussian kernel to ease the task of selecting the best structure of SVM. In this sense the standard deviation (stdv) of the Gaussian is the parameter to optimize. We could also look at the C parameter which deals with the security margin and the error, but at it is not asked we let the default value *C=100*.

B. Regularization using standard deviation.

To build this regularization plot we have trained the SVM changing the value of the stdv from 1 to 200 with unit step. Doing this operation we denote that the SVM is really fast to learn (maybe due to the purpose of Torch when it was created).

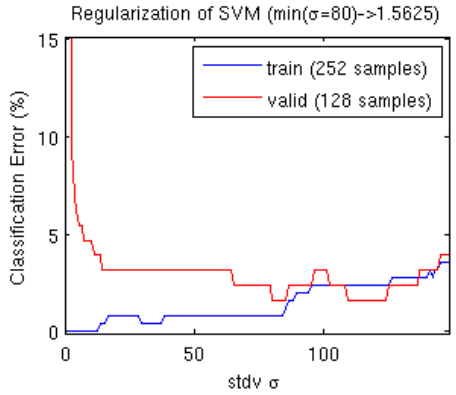


Figure 4: Regularization graph (optimal std deviation)

We finally obtain the regularization graph in Figure 4, and we take the standard deviation equal to $\sigma=80$.

We also notice, that has the set has a small size we have our curves with big step. To compensate this effect we have performed this operation on different random arrangements of training and validation. Applying this method we have noticed that the optimal σ highly depend on how the two set are distributed.

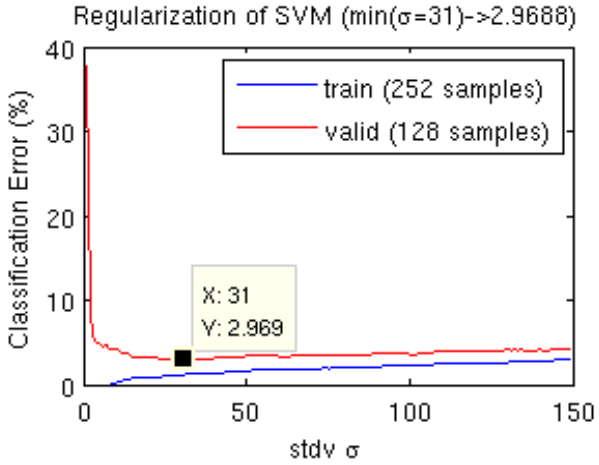


Figure 5: Regularization graph over 20 randomly built set

As Figure 5 shows, the optimal mean value of standard deviation other 20 iterations is $\sigma=31$.

	Training	Validation	Testing
Classification error $\sigma=80$	0.7956%	1,5625%	3.1752%
Classification error $\sigma=31$	0.3968%	2.4617%	3.1658%

This shows that using $\sigma=31$ may be a better idea to perform better classification on unknown data because it is more general (as we use a sort of cross validation).

V. Gaussian Mixture Model

A.Description & Structure.

Since the distribution of the point in our database can't be easily represented by only one probability function, the use of Gaussian Mixture Model (GMM) which is a linear combination of gaussian probability distributions (also know as normal probability distributions) reveals to be an interesting option. The key problem is to determine the number M of probability distributions allowing to elegantly discriminate the inputs and match them to the different classes.

The parameter M fixed, there is an other important task to achieve. Recalling a gaussian probability distribution is managed by two important parameters, the mean μ and the variance σ^2 , the use of the K-Means algorithm [1] allows to discover the parameters for each one of the M gaussian distributions.

Technically, this last operation is done inside the Torch framework through the initialization of *DiagonalGMM object* using a *Kmeans object* preliminary trained to discover the initial gaussian parameters.

B.Expectation-Maximization algorithm

The GMM is trained using the Expectation-Maximization (EM) algorithm [2]. The concept behind the EM strategy is a 2-step mechanism iteratively repeated until convergence. The stopping criterion is defined by the *end_accuracy*. It means the process is stopped when the difference between the iteration n and the iteration $n+1$ is smaller than the *end_accuracy* value. After some trials, a value of 10^{-3} for *end_accuracy* seems to minimize the NLL the best.

The value gathered at each EM iteration can be described as the negative log-likelihood (NLL).

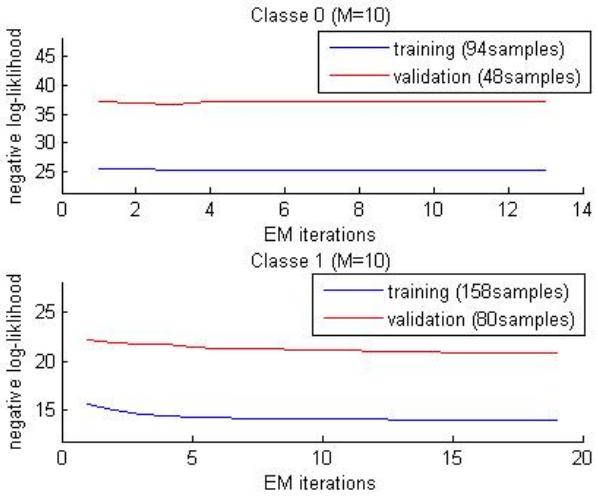


Figure 6: Negative log-likelihood = function(EM iteration) for M = 10 gaussians

Looking at Figure 6, we can't observe any significant overfitting for any of the classes.

C.Regularization using the number of gaussian.

Next, we wanted to understand how the number of gaussians influences the classification error. Before going further in our analysis, let's recall how are generated the different model.

First, each dataset (training, validation and testing sets) are split by classes. Then, using the training data, the GMM model is initialized with the K-Means algorithm and is trained (with an *EMTrainer object*) on each classes independently. Since the database used has 2 classes (C_0 and C_1), we obtained two models gmm^0 and gmm^1 for each dataset.

Later on, we wrote a *classification error function* able to compute the number of misclassified *examples*. An *example* being a set of X inputs. The core of this function is based on the simple comparison of the probability returned by each of the models for a given *example*. For each example, we obtain the probabilities of being in C_0 and in C_1 , if the *example* should be in C_0 and the gmm^1 returns the highest probability, it's a misclassified one. We iterate this comparison for all the examples and we obtain the percentage of classification error. Applying this method for each of the set, we can visualize the efficiency of the GMM model compared to the number of gaussians used (see Figure 7).

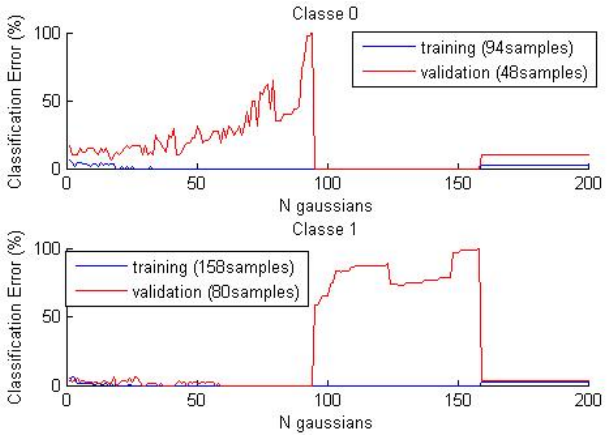


Figure 7: Regularization graph for GMM

Analyzing Figure 7, we can extract the number of gaussians minimizing the classification error for both classes C_0 and C_1 . Setting the parameter $n_{gaussians}$ to 17 seems to be a good compromise (see Table 2).

	Training	Validation	Testing
Classification error C_0 (%)	2.1276	6.2500	8.5714
Classification error C_1 (%)	0.6329	1.2500	3.3613

Table 2: Classification error of C_0 and C_1 for M = 17 gaussians

VI. Discussion

I m working on it .. :)

VII. References

[1] Kmeans - http://en.wikipedia.org/wiki/K-means_algorithm

[2] EM - http://en.wikipedia.org/wiki/Expectation-maximization_algorithm