Artificial Neural Networks and Deep Architectures,
DD2437

Lab assignment 1a

Classification with a single-layer perceptron

1 Introduction

This exercise is concerned with learning approaches for a single-layer perceptron (also know as a threshold logic unit or McCulloch-Pitts neuron) in simple binary classification problems. The two main learning algorithms considered in this lab are percpetron and delta (Widrow-Hoff) rule.

1.1 Aim and objectives

After completion of the lab assignment, you should be able to

- design and apply perceptrons for simple *classification* tasks
- identify key limitations of single-layer networks and of their learning methods (classical perceptron and delta rule methods)
- configure and monitor the behaviour of learning algorithms for single-layer perceptrons

1.2 Scope

In this short introductory lab you will implement single-layer perceptrons with focus on classical *perceptron* and Widrow-Hoff *delta rule* learning algorithms. You will then study their properties by means of simulations. Since the calculations are naturally formulated in terms of vectors and matrices, the exercise was originally conceived with MATLAB[1] in mind. However, you are free to choose your own programming/scripting language, environment etc. The intention here is that you develop all the code from scratch.

[1]It is also possible to use Octave, the free version of MATLAB.

2 Background

2.1 Data Representation

The data can be effectively represented in matrices (collection of vectors). Since we follow a supervised learning approach (here for classification), our traning data should consist of input patterns (vectors) and the associated output patterns, often called labels (e.g., scalar values for classification and regression). There are two options to perform training - sequential on a sample-by-sample basis and batch. For classical perceptron learning a sequential approach is used whereas Widrow-Hoff rule is often applied in a batch mode, which means that all patterns in the training set are used as a whole at the same time instead of stepping through them one by one and updating weights successively for each sample (input pattern with its associated label/output). Batch learning is better suited for a matrix representation and is significantly more effective given built-in functions for quick matrix operations in most programming/scripting languages. In batch learning, each use of the whole set of available training patterns is commonly referred to as an *epoch* and the entire training process involves many iterative epochs. By a suitable choice of representation, an epoch can be performed with just a few matrix operations.

Further, in problems where binary labels (0/1) are inherent (e.g. bnary classification problems), it is convenient sometimes and practical to rely instead on a symmetric $(-1/1)$ representation of the patterns, sometimes referred to as a bipolar representation. Most importantly, you must make sure that your learning rule is compliant with the representation of your class labels/targets, as discussed in the lecture - typically $(0/1)$ for perceptron rule and $(-1/1)$ for delta rule.

The input patterns (vectors) as well as their corresponding targets/labels (predominantly scalar values) can be represented as columns in two matrixes, X and T, respectively. With this representation, the XOR problem would for instance be described by

$$X = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

If we read the matrices column-wise, we get the pattern $(-1, -1)$ to be associated with the output -1, and the pattern $(1, -1)$ with the output 1 etc.

A single-layer perceptron sums the weighted inputs, adds the *bias* term and produces the thresholded output. If you have more than one output, you have to have one set of weights for each output. These computations become very simple in matrix form. Make sure however that you account for the bias term by adding an extra input signal whose value always is one (and a weight corresponding to the bias value, as shown in the lecture). In the XOR example we thus get an extra column:

$$X_{\texttt{input}} = \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

The weights are stored in matrix W with as many columns as the dimensionality of the input patterns and with the number of rows matching the number of the outputs (dimensionality of the output). The network outputs corresponding to *all* input patterns can then be calculated by a simple matrix multiplication followed by thresholding at zero (since the bias has been already taken into account in the extra column of the weight matrix, provided that an extra entry with the constant value 1 was also included in the formation of the inputs, as explained earlier and discussed in the lecture). Learning with the delta rule aims, with the representation selected, to find the weights W that give the best approximation:

$$W \cdot \begin{bmatrix} -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \approx \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$$

Unfortunately, the XOR problem is one of the classical problems that a single-layer perceptron cannot solve.

2.2 Implementation of the delta rule

Store the training data in variables `patterns` and `targets`. As discussed above, add an extra row to the input patterns with ones corresponding to the extra bias terms in the weight matrix.

The delta rule can be written as:

$$\Delta w_{j,i} = -\eta x_i \left(\sum_k w_{j,k} x_k - t_j \right)$$

where $\bar{x}$ is the input pattern, $\bar{t}$ is the wanted output pattern and $w_{j,i}$ is the connection x_i to t_j. This can be more compactly written in matrix form:

$$\Delta W = -\eta (W\bar{x} - \bar{t})\bar{x}^T$$

The formula above describes how the weights should be changed based on *one* training pattern (and its matching target label). To get the total weight change for the entire epoch, i.e. accounting for all training patterns, the weight update contributions from *all* patterns should be summed. Since we store the patterns as columns in X and T, we get this sum "for free" when the matrixes are multiplied. The total weight change from a whole epoch (batch mode since the learning contributions from each sample are summed and the weight update is performed at once) can therefore be written in this compact way:

$$\Delta W = -\eta(WX - T)X^T$$

Write your code so that the learning according to the formula above can be flexibly repeated `epochs` times (where 20 is a suitable number for a low-dimensional perceptron). Try to avoid loops as much as possible at the cost of powerful matrix operations (especially multiplications). Make sure that your code works for arbitrary sizes of input and output patterns and the number of training patterns. The step length or *learning rate* η should be set to some suitable small value like 0.001. Note: a common mistake when implementing this is to accidentally orient the matrixes wrongly so that columns and rows are interchanged. Make a sketch on paper where you write down the sizes of all components starting by the input and how the dimensionality propagatesto the weights to the output. This will be particularly important in the next part of the lab with a two-layer perceptron.

Before the learning phase can be executed, the weights must be initialised (have initial values assigned). The normal procedure is to start with small random numbers drawn from the normal distribution with zero mean. Construct a function to create an initial weight matrix by using random number generators built into programming/scripting languages. Note that the matrix must have matching dimensions.

2.3 Monitoring the learning process and evaluation

Monitoring the process of learning for multi-layer perceptrons is not as simple as for a single-layer perceptron, which could be done by drawing the line of separation - decision boundary. For multi-layer networks we commonly rely on the output error as a probe for the advancement of the learning process. It is a common practice therefore to plot learning curves with the error estimated either by the mean square error or, in classification tasks, as the total number or proportion of misclassifications (there are also other error measures). Such *learning curves* illustrate the progress made over consecutive epochs (the error is usually estimated for the entire epoch, i.e. across all the training patterns). Keep in mind however that the training error does not provide a good estimate for a generalisation capability of the network.

3 Assignment

3.1 Classification with a single-layer perceptron

3.1.1 Generation of linearly-separable data

In the first place, please generate some data that can be used for binary classification (two classes). To simplify visual inspection, let us work with two-dimensional data. To start with, please draw two sets of points/patterns in 2D from multivariate normal distribution. In MATLAB, one could easily do that in the following way:

```
n = 100;
mA = [ 1.0, 0.5];   sigmaA = 0.5;
mB = [-1.0, 0.0];   sigmaB = 0.5;
classA(1,:) = randn(1,n) .* sigmaA + mA(1);
classA(2,:) = randn(1,n) .* sigmaA + mA(2);
classB(1,:) = randn(1,n) .* sigmaB + mB(1);
classB(2,:) = randn(1,n) .* sigmaB + mB(2);
```

Please choose parameters, `mA`, `mB`, `sigmaA`, `sigmaB`, yourselves to make sure that the two sets are linearly separable (so the means of the two distributions, `mA` and `mB`, should be sufficiently different). You can generate `n`=100 points per class and shuffle samples (`randperm` indexes in MATLAB) so that in your dataset you would not have just two concatenated blocks of samples from the same class. Although this reordering (shuffling) does not matter for bacth learning, it has implications for the speed of convergence for sequential (on-line) learning, where updares are made on a sample-by-sample basis. Please plot your patterns with different colours per class.

3.1.2 Classification with a single-layer perceptron and analysis

In this part, you are requested to implement and apply percpetron learning rules and analyse the results obtained on the generated dataset. It is helpful and fun to visualise the learning process by plotting a separating line (decision boundary) after each epoch of training. For that you could generate a sort of animation by iteratively re-plotting the updated decision boundary (see `drawnow` in MATLAB), in that case remember to keep the scame scaling in the plot every time you update it (in MATLAB you can force the scaling using `axis`). You are not required to demonstrate this animation to the teaching assistant, it is sufficient to show the final decision boundary (decision boundary can be derived from the weights and biases in W, i.e. it is a line defined by $Wx = 0$).

These are the tasks to focus on in the exercise:

1. Apply and compare perceptron learning with the delta learning rule in batch mode on the generated dataset. Adjust the learning rate and study the convergence of the two algorithms.
2. Compare sequential with a batch learning approach for the delta rule. How quickly (in terms of epochs) do the algorithms converge? Please adjust the learning rate and plot the learning curves for each variant. Bear in mind that for sequential learning you should not use the matrix form of the learning rule discussed in section 2.2 and instead perform updates iteratively for each sample. How sensitive is learning to random initialisation?
3. Remove the bias, train your network with the delta rule in batch mode and test its behaviour. In what cases would the perceptron without bias converge and classify correctly all data samples? Please verify your hypothesis by adjusting data parameters, `mA` and `mB`.

Comparisons can be made using some evaluation metrics that could be the number or ratio of misclassified examples and/or the mean squared error at each epoch (iteration through the entire dataset).

3.1.3 Classification of samples that are not linearly separable

In this exercise, you should study how perceptron deals with data samples that are not linearly separable. To this end, please generate first a dataset with such property by adjusting `mA`, `mB`, `sigmaA` and `sigmaB`. In particular, you can make the means of the two multivariate normal distributions more similar and/or increase the spreads. As a result, you should see that the two clouds of points (corresponding to the two classes) overlap when you plot the samples. You can control the amount of overlap by the parameters of the distributions. Apply and compare perceptron learning and the delta rules using either batch or sequential learning mode, similarly as in the previous exercise (p.1, section 3.1.2).

In the second part of this exercise please generate a different version of linearly non-separable dataset in the following way:

```
   ndata = 100;
   mA = [ 1.0, 0.3];    sigmaA = 0.2;
   mB = [ 0.0, -0.1];   sigmaB = 0.3;
   classA(1,:) = [ randn(1,round(0.5*ndata)) .* sigmaA - mA(1), ...
randn(1,round(0.5*ndata)) .* sigmaA + mA(1)];
   classA(2,:) = randn(1,ndata) .* sigmaA + mA(2);
   classB(1,:) = randn(1,ndata) .* sigmaB + mB(1);
   classB(2,:) = randn(1,ndata) .* sigmaB + mB(2);
```

Then apply the delta learning rule in batch mode to this new dataset as well as to different versions of the subsampled data, i.e. before training please remove 25%

of data samples (for two classes with n=100 samples each, remove 50 samples) according to the following scenarios:

- random 25% from each class
- random 50% from classA
- random 50% from classB
- 20% from a subset of classA for which classA(1,:)<0 and 80% from a subset of classA for which classA(1,:)>0

Perform simulations, i.e. subsample your data (from the same dataset generated originally) and perform perceptron learning, a few times. What is the effect of different data subsampling manipulations on the results, i.e. the localisation of the decision boundary as well as the performance? Since the number of samples representing different classes are different for the last three cases, it may be easier to measure the performance using the accuracy rate estimated independently for each class (alternatively, one could quantify *sensitivity* and *specificity*).
In your observations reflect on the implications of subsampling - data points that you remove could be thought of as an unseen test set. How would the resulting sampling bias, both in the context of uneven class representations and non-representative sample distribution, affect the generalisation?