Assignment 3: Neural Networks

H. Blum, D. Cavezza, A. Paudice and M. Rohbeck
Machine Learning CO395
Imperial College London

November 30, 2015

1 Validation

In our second assignment, we apply neural networks to the emotion recognition problem. We use the Neural Network Toolbox provided by MATLAB to train and compare the performance of different neural networks on the dataset at our disposal, in order to find the best training algorithm along with the best parameter configuration.
We compare four different training algorithm:

- Standard gradient descent backpropagation (`traingd` in MATLAB);
- Gradient descent with adaptive learning rate (`traingda`);
- Gradient descent with momentum (`traingdm`)
- Resilient backpropagation (`trainrp`).

In this section we describe our implementation of:

- selection of the best set of parameters for each algorithm;
- evaluation of NN's performance on unseen data.

1.1 Implementation

In the first part, we use cross- validation to select the best performing algorithm on the dataset and the best parameter configuration for it. Cross-validation is performed by splitting the dataset into 10 folds and using 9 folds for training and 1 for validation; iteratively, each fold is in turn used for validation, and ultimately the algorithm and parameter set that yield the best average performance over the folds are chosen.
For each algorithm, we compare several network topologies and parameter values in all their combinations. The tested values are shown in Table 1
For further details on the criteria that led to those choices, see the answer to Question 1.
The trained network has the default architecture returned by the function `feedforwardnet`: it uses sigmoidal activation functions in the hidden layers and linear activation functions for the output neurons. Therefore, our choice for the performance measure falls onto the Mean Squared Error.
A known problem of the basic training algorithm regards the random choice of the initial weights: a bad choice may lead the MSE to converge to a local minimum different from the global one. To soften the effect of the random choice, in each trial we train the network five times with the same parameters and take as a performance measure the minimum MSE on

Table 1: Parameters tested

Neurons per hidden layer	From 6 to 45	
Hidden layers	[1, 2]	
`traingd`	Learning rate	[5 3 1 0.5 0.3 0.1 0.05 0.03 0.01]
`traingda`	Learning rate	[1 0.1 0.01]
	LR decrease ratio	[0.7 0.07 0.03]
	LR increase ratio	[1.4 2 5]
`traingdm`	Learning rate	[5 3 1 0.5 0.3 0.1 0.05 0.03 0.01]
	Momentum	[0.9 0.95]
`trainrp`	Delta increase	[1.4 1.3 1.2 1.1]
	Delta decrease	[0.7 0.5 0.3 0.07 0.05 0.03]

the validation set.
For splitting the dataset into folds, we use the same function as in the previous exercise, which performs *stratified* cross-validation: each fold contains approximately the same proportion of examples in every class as the whole dataset. It is implemented in the file `getFoldsPartitioning.m`.
The files `crossValidate.m`, `validateNeuralNetwork.m` and `repeatNNTraining.m` implement the cross-validation. The function `crossValidate` requires in input the algorithm to be validated and the dataset. The outputs are:

- `parameters`: a cell array containing all the parameter values tested in the cross-validation; each cell contains an array of values for a single parameter. For instance, in *traingd* the first cell contains the tested values for the number of neurons per hidden layer, the second cell contains the numbers of hidden layers and the third one all the learning rates tested;
- `mserrors`: a multidimensional array containing the average MSE computed over all the validation sets for each combination of parameters; the indices of each element in the array correspond to the indices of the parameter values in `parameters`.

Since the cross-validation requires time to be completed, the function saves the performance computed on every validation fold by every parameter configuration as a precaution.

```
1  function [ parameters, mserrors ] = crossValidate( algorithm, attributes, labels )
2
3  % Get the indices of the 10 folds as a cell array of 10 indices arrays
4  foldsIndices = getFoldsPartitioning(labels,10,true);
5
6  [parameters, numParams] = getParameters(algorithm);
7
8  mserrors = zeros(numParams);
9  for i=1:10
10     disp(['Testing fold ' num2str(i)]);
11     trainingSetIndices = getTrainingSetIndexed(foldsIndices,i);
12     validationSetIndices = foldsIndices{i};
13
14     mserrorsPerFold = validateNeuralNetwork(algorithm,parameters,attributes,labels,
           trainingSetIndices,validationSetIndices);
15     save([algorithm '_msErrorsFold' num2str(i) '.mat'],'parameters','mserrorsPerFold');
16
17     mserrors = mserrors+mserrorsPerFold;
18 end
19 % Average the accuracies
20 mserrors = mserrors./10;
21 save([algorithm '_avgmsErrors.mat'],'parameters','mserrors');
22
23
24 end
```

For each validation fold, the function `validateNeuralNetwork` is called. The main structure of this function has a `switch` statement, which checks the algorithm that is being tested; according to the algorithm, it loops over all the parameters that we need to test and calls the function `repeatNNTraining` for each parameter combination. This function first sets the training and validation sets for the training algorithm,

```
17     net.divideFcn = 'divideind';
18     net.divideParam.trainInd = trainingSetIndices;
19     net.divideParam.valInd = validationSetIndices;
20     net.divideParam.testInd = [];
```

then it configures the input and output layers' sizes and trains the network,

```
22     % Set up input and output layer
23     NN{j} = configure(net, attributesNN, labelsNN);
24     % Train network
25     [NN{j}, trainRecord] = train(NN{j}, attributesNN, labelsNN);
```

and finally keeps the performance value computed on the validation set for the following operations; this value can be retrieved via the field `best_vperf` of the struct `trainRecord`.

```
30     perfs(j) = trainRecord.best_vperf;
```

At the end of the five iterations, the network that performed best on the validation set is returned, along with the error measured. As a technique to avoid overfitting, we rely on the Early Stopping implemented by default in NNTools. This technique stops the training before the maximum number of iterations of gradient descent is performed: when the MSE on the validation set starts rising, the algorithm keeps executing for a maximum of `net.trainParam.max_fail` iterations; if the error does not decrease in these iterations, the algorithm stops and the weights of the network are set to the values that achieved the minimum error on the validation set.

1.2 Selected parameters

2 Performance evaluation

2.1 Implementation

For evaluating the performance of the chosen model on unseen data, we use cross-validation with three sets:

- a training set, used to execute gradient descent;
- a validation set, used to determine the termination condition and avoid overfitting, as well as optimizing the parameters;
- a test set, used for evaluating the performance.

The procedure is aimed at verifying the performance of the model obtained via the procedure of the previous subsection on data that have been used neither for training nor for parameter optimization.
The dataset is divided again in 10 folds. In each of the 10 iterations, one fold is used as test set, one as validation set and the remaining 8 as training set. The network is trained on the training set and the parameters are optimized on the validation set, by using MSE as in the validation above; the resulting network is then tested on the test set in terms of confusion matrix and derived performance measures (accuracy, precision, recall). Since in each iteration the optimization is done on a different validation set, it may happen (and it actually does) that different parameter configurations are tested on different folds: in this way, we are truly estimating the performance of the entire approach we use to train and optimize the network, not just of a single trained network.

2.2 Performance results

3 Questions

3.1 Question 1

We chose the optimal topology and parameters through cross-validation. In detail, in each iteration we tested the performance of the chosen topology on the fold used as validation set; at the end, we averaged the performances of each topology and parameters configuration and chose the setting that showed the best average performance.
We tested topologies with 1 and 2 layers. Topologies with 1 layer can fit any Boolean function, while 2-layer topologies can approximate arbitrarily well any real-valued function.
For each layer, a common practice is to use a number of neurons between the sizes of the input and the output layer. Choosing less neurons than the output leads to a data compression that may cause information loss before reaching the output.
For the algorithms' parameters, we had to trade off the number of tests executed and the total time for testing. We chose learning rates that covered different orders of magnitude where possible.
The optimal parameters for the standard gradient descent are:
Neurons per layer = 18; Number of layers = 2; Learning rate = 0.5; Avg Error = 0.0458
The optimal parameters for the adaptive gradient descent are:
Neurons per layer = 15; Number of layers = 2; Learning rate = 0.1; LR decrease rate = 0.7; LR increase rate = 1.4; Avg Error = 0.0449
The optimal parameters for the gradient descent with momentum are:
Neurons per layer = 42; Number of layers = 1; Learning rate = 1; Momentum coefficient = 0.9; Avg Error = 0.0679
The optimal parameters for resilient backpropagation are:
Neurons per layer = 14; Number of layers = 2; Delta increase = 1.3; Delta decrease = 0.5; Avg Error = 0.0444

3.2 Question 1

3.2.1 Overfitting

Matlab already implemented *early stopping* by default. [some words about early stopping]. We considered using *regularization*, but this would include another paramenter to optimise. Because all the parameters already took us 2 days, we didn't want to double/triple this time. However, theoretically also other techniques [...] are possible.

3.2.2 6 Networks with single output

Disadvantages
First of all this includes to implement a decision function which classifies on basis of the outputs of the 6 networks (like for the decision trees).
The task for each of the 6 networks is to find a boolean decision function. If we assume that we test the networks with the same topologies as the 1 big network, these networks can represent *exactly* one boolean function as long as they have at least 2 layers. This leads to a huge risk of overfitting compared to the 1 network which approximates an arbitrary function and is therefore more robust to noise.

Furthermore, training 6 networks could lead to a higher risk of overfitting, as the parameters are optimized differently for each output class. Also, this is a time factor as long as one does not have access to 6 powerful computers. (It is possible to parallelize the process of cross-validation as the networks are independent).
Advantages
As the networks are trained more specific to recognize one class, this approach could lead to a better performance.
Because the output layer is smaller than for 1 big network, there are less weights to optimize in each network for the same topology. As a result, parallel cross-validation of 6 networks will be faster than the cross-validation of 1 big network.

Combination
The combination of the output of the 6 networks is exactly the problem solved in out last Assignment. We will just look at each discussed strategy:

random choice still possible with a threshold value (output a random choice between each network that outputs a value bigger than the treshold.

score-based decision naturally, the output of the networks is also a score, so this is possible without modifications

depth-based decision the depth of a tree is a specific property of a decision tree and therefore not easy to find a similar property for neural networks. Of course, one could use the number of layers, but as this varies just between two values, the algorithm would probably not perform well.

error-based decision as the error of the networks can be calcualted just as the errors of the trees, this strategy is also possible to use without modification

3.3 Question 2

1)

After performing a t-test, feedforward neural networks showed to have a statistical significantly smaller classification error than our decision tree implementation. However, although for our specific test set performance sample neural networks did better than decision tree, it is not possible to state general conclusion about the relative performance of this two learning algorithm.

First, it should be observed that a t-test just refers to a specific sample of test performance. This implies that the resulting difference between of the two algorithms is just an estimate of the true difference. Thus, any conclusion behind the observed sample is statistically meaningless. Second, in our case we compared the classification error of the algorithms and in general it could be possible that for other performance metrics the resulting t-test can be different.

To summarize then, it is not possibile to establish if a specific learning algorithm is better than an another. Rather, analyzing the performance of the algorithms one should always consider the specific problem domain, the relevant performance metric and the available dataset.

2)

We trained and tested decision trees and neural networks on the same training and test sets which leads to dependent samples. Because of this, we adopted a paired t-test.

3)

Considering the classification error rather than the F1 measure has the advantage to avoid the so called multiple hypothesis comparison problem. Indeed adopting the F1 measure for this multi-class classification problem leads to 6 different measurements per test set, i.e one per each emotions. As a result the sample for the t-test is constituted by 10 six-elements vector. To compare the two algorithms in this way we have to realize 6 different t-test one per each emotions which leads to the multiple hypothesis comparison problem. Testing simultaneously 6 hypothesis with a significance (i.e a probability of rejecting the null hypothesis when it is actually true) $\alpha = 0.05$, leads to an overall probability of error of 0.3 which lead the test much more effective.

4)

Let's first analyze the case of reducing the number of folds. Reducing the number of folds leads to a smaller sample on which to perform the t-test. Then, the *SE* term of the t-statistic increases and the resulting t value get smaller. Moreover, as the number of degree of freedom (df) is directly proportional to the sample size, the corresponding threshold is bigger. As a result, having a smaller statistic to test against a higher threshold the number of false rejection for the null hypothesis increases.

Conversely, increasing the number of fold leads to a bigger t-statistic and a smaller threshold which in turn might lead to accept the null hypothesis when it is actually false. In any case, it should be observed that increasing the number of folds, one should be aware to get fold with at least 30 examples. Indeed, 30 examples is the minimum number of sample to approximate the fold individual differences between the two algorithm as realization of a normal random variable which provide the theoretical ground for the validity of the t-test. In addition, it is important specify that there is no argument to believe that 10 folds represent the optimum for this specific test.

5)

In the case of the decision trees learning, adding new emotions requires to grow additional decision decision trees (in particular as many decision trees as many emotions we add) as well as retrain all the others. It should be observed that retraining is required since adding new emotions to the dataset enrich the base of negative examples of the pre-existing classes. On the other hand, in the case of neural networks it is required add as many additional output layers as new emotions are introduced, retrain and optimize trough cross-validation the network.