Assignment 3: Neural Networks

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

December 1, 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

The parameters that reach the minimum average MSE over the 10 validation sets are:

- **Algorithm:** Resilient backpropagation (`trainrp`)
- **Neurons per layer:** 14
- **Layers:** 2
- **Delta increase:** 1.3
- **Delta decrease:** 0.7

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 cross-validation 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; in this phase the performance measure used, both for training and validation, is MSE as in the previous section. As before, we repeat the training 5 times and take the network that achieved the minimum error on the validation set. 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.
The implementation is provided in the file `performanceEstimation.m`. The function first sets up the training, validation and test sets, and then loops over the training algorithms. For each algorithm, it calls `validateNeuralNetwork` (line 36) to obtain the MSE values for all the configuration parameters. The minimum MSE is recorded for each algorithm and finally the best MSE over all the algorithms on the given validation set is found.

```
1  function [confusionMatrix, accuracy, precision, recall, f1] = performanceEstimation(attributes,
       labels)
2
3  algorithms = {'traingd', 'traingda', 'traingdm', 'trainrp'};
4  n_alg = length(algorithms);
5  confusionMatrix = zeros(6);
6  accuracyFolds = zeros(10,1);
7  precisionFolds = zeros(10,6);
8  recallFolds = zeros(10,6);
9  f1Folds = zeros(10,6);
10
11 % Get the indices of the 10 folds as a cell array of 10 indices arrays
12 foldIndices = getFoldsPartitioning(labels,10,true);
13
14 for i = 1:10 % iterate over 10 test folds
15     % Split: Test- 1 Fold (i) Validation- 1 Fold (k) Training-8
16     testSetIndices = foldIndices{i};
```

```
    k = mod(i,10)+1;
    validationSetIndices = foldIndices{k};
    trainingSetIndices = [];
    for j=1:10
        if j~=i && j~=k
            trainingSetIndices = [trainingSetIndices foldIndices{j}];
        end
    end

    % find optimal Parameters with training and validation set
    bestPerformingAlgorithm = 1;
    bestMSE = Inf;
    optimalParameters = [];

    for k = 1:n_alg
        parameters = getParameters(algorithms{k});

        disp(['Fold ' num2str(i) ' Algorithm: ' algorithms{k}]);

        mserrorsAlgorithm = validateNeuralNetwork(algorithms{k}, parameters, ...
            attributes, labels, trainingSetIndices, validationSetIndices);
        % idx is the index of the minimum in the linearized
        % mserrorsAlgorithm
        [minMSE, idx] = min(mserrorsAlgorithm(:));
        % Get the corresponding indices in the multidimensional array
        idxParameters = cell(1,length(parameters));
        [idxParameters{:}] = ind2sub(size(mserrorsAlgorithm), idx);

        if(minMSE<bestMSE)
            bestMSE = minMSE;
            bestPerformingAlgorithm = k;
            optimalParameters = zeros(length(parameters),1);
            for j=1:length(parameters)
                optimalParameters(j) = parameters{j}(idxParameters{j});
            end
        end
    end

    optimalAlgorithm = algorithms{bestPerformingAlgorithm};
```

The optimized network is used to perform a test on the test set and the corresponding confusion matrix is recorded. Both the optimal network and the confusion matrix are saved to the file `confMatrixFold<i>.mat` for subsequent use.

```
    % Configure the best training algorithm with the optimal parameter
    % configuration
    net = configureNeuralNetwork(optimalAlgorithm,optimalParameters);
    % Train the network 5 times and get the best network
    [attributesNN,labelsNN] = ANNdata(attributes,labels);
    [~,net] = repeatNNTraining(net,attributesNN,labelsNN,trainingSetIndices,validationSetIndices)
        ;

    % Compute performance on test set
    predictions = NNout2labels(sim(net,attributesNN(:,testSetIndices)));
    confMatrixFold = getConfusionMatrix(labels(testSetIndices),predictions,6);
    save(['confMatrixFold' num2str(i)],'net','confMatrixFold');

    accuracyFolds(i) = sum(diag(confMatrixFold))/length(testSetIndices); % The sum of all the
        elements in confMatrixFold is
                                                                          % equal to the size of
                                                                              the test set
    precisionFolds(i,:) = diag(confMatrixFold)./sum(confMatrixFold,1)';
    recallFolds(i,:) = diag(confMatrixFold)./sum(confMatrixFold,2);
    for j=1:6
        if(precisionFolds(i,j)+recallFolds(i,j)==0)
            f1Folds(i,j)=0;
        else
            f1Folds(i, j) = 2* precisionFolds(i,j).*recallFolds(i,j) ...
                ./ (precisionFolds(i,j) + recallFolds(i,j));
        end
    end
    confusionMatrix = confusionMatrix+confMatrixFold;
end

accuracy = mean(accuracyFolds);
precision = mean(precisionFolds,1);
```

True\Predicted	*anger*	*disgust*	*fear*	*happiness*	*sadness*	*surprise*
anger	83	12	9	4	23	1
disgust	14	158	3	5	15	3
fear	0	2	85	3	10	19
happiness	0	4	5	201	6	0
sadness	11	18	7	7	86	3
surprise	1	0	10	3	5	188

(a) Confusion matrix

Class	Precision	Recall	F_1	Accuracy
anger	69.8%	63.0%	65.5%	
disgust	81.6%	79.8%	80.3%	
fear	72.4%	71.4%	70.8%	79.8%
happiness	90.6%	93.1%	91.6%	
sadness	60.8%	65.1%	62.3%	
surprise	88.4%	90.9%	89.4%	

(b) Performance metrics

Figure 1: Performance of Neural Network on clean data

True\Predicted	*anger*	*disgust*	*fear*	*happiness*	*sadness*	*surprise*
anger	6	16	21	9	32	4
disgust	2	151	12	7	12	3
fear	1	13	133	11	14	15
happiness	3	8	14	173	6	5
sadness	2	14	16	5	61	12
surprise	1	4	18	7	10	180

(a) Confusion matrix

Class	Precision	Recall	F_1	Accuracy
anger	27.5%	6.9%	10.2%	
disgust	74.6%	80.7%	77.0%	
fear	62.5%	71.2%	66.5%	70.4%
happiness	81.9%	82.8%	81.8%	
sadness	45.6%	55.5%	49.6%	
surprise	82.3%	81.8%	81.9%	

(b) Performance metrics

Figure 2: Performance of Neural Network on noisy data

```
86  recall = mean(recallFolds,1);
87  f1 = mean(f1Folds,1);
88
89  end
```

To compute the total confusion matrix, we sum the confusion matrices of the folds. To compute the performance statistics of the whole network, we average the performance statistics over the folds.

2.2 Performance results

The performance results of the network trained and tested on clean data are shown in Figure 1

3 Questions

3.1 Part A: Questions

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. In the end, we averaged the performances of each

topology and parameters configuration and chose the setting that showed the best average performance.
In the following our reasons behind the set of tested parameters:

- **Number of Layers:** We tested topologies with one and two layers. Topologies with one layer can fit any boolean function, while two-layer topologies can approximate arbitrarily well any real-valued function.
- **Number of Neurons:** 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 parameters that covered different orders of magnitude where possible.

- **Learning Rate:** As usual values, the lecture slides give 0.1 and 0.01. Because this is an essential parameter, we tested a larger set of different values at 3 different orders of magnitude, on higher than 0.1 and 1 lower than 0.1. As the `traingda` algorithm has 3 parameters to optimize, we had to reduce the number of tested learning rates here.
- **LR Decrease Ratio:** As typical values, 0.5 and 0.7 are given in the lecture slides. [...]
- **LR Increase Ratio:** As typical values, 1.05 and 1.1 are given in the lecture. [...]
- **Momentum:** We tested the both values given as typical in the lecture slides.
- **Delta Increase:** As a typical value, 1.2 is given in the lecture slides. As this value has to be bigger than 1, we tested values around this. We didn't consider higher orders of magnitude, as this would just lead to exploding update values and therefore most probably to a lot of oscillation and long converging times. In the end, our optimal parameter was close by the given value.
- **Delta Decrease:** As a typical value, 0.5 is given in the lecture slides. Again, we tested different values around this number and this time also tested values of smaller magnitude. Bigger magnitues were not possible as the parameter has to be smaller than 1. Our optimal parameter actually was 0.5.

The optimal parameters for the different learning techniques are the following.

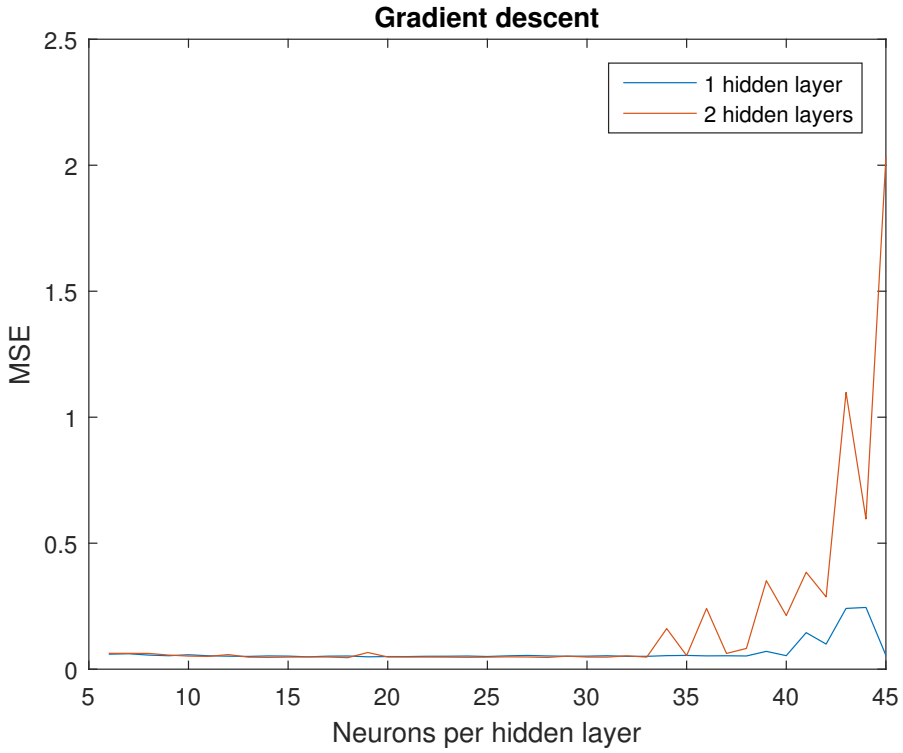
- **standard gradient descent:**
  Neurons per layer = 18, Number of layers = 2, Learning rate (LR) = 0.5, Avg. Error = 0.0458
- **adaptive gradient descent:**
  Neurons per layer = 15, Number of layers = 2, Learning rate = 0.1, LR decrease rate = 0.7, LR increase rate = 1.4, Avg. Error = 0.0449
- **gradient descent with momentum:**
  Neurons per layer = 42, Number of layers = 1, Learning rate = 1, Momentum coefficient = 0.9, Avg. Error = 0.0679
- **resilient backpropagation:**
  Neurons per layer = 14, Number of layers = 2, Delta increase = 1.3, Delta decrease = 0.5, Avg. Error = 0.0444

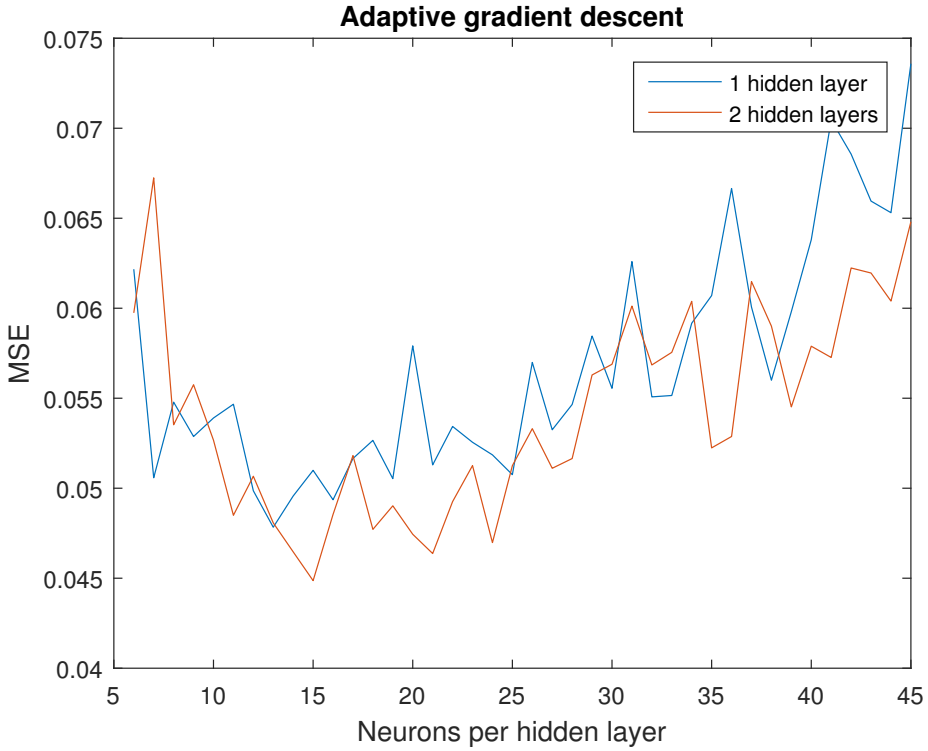
Question 2

With the exception of the Momentum-Algorithm, the plots show increasing error for very large number of neurons in the hidden layer(s). For all but the standard gradient descent, the error becomes also bigger for very small numbers of hidden layer neurons. Our explanation is as follows.

In general, the number of neurons in the hidden layers defines the degrees of freedom in the approximation of the classification function. Therefore, very few hidden neurons lead to very few degrees of freedom, which would explain higher classification errors as the ANN may not be able to fit the training data to a suitable degree. On the other hand, a high degree of freedom increases the possibilities for overfitting, which again will cause a higher classification error on the test set.

Especially interesting is the performance in relation to the number of hidden layers. This is linked to the size of the searched hypothesis space as 2-layer-ANNs can only approximate continuous functions whereas 3-layer-ANNs can approximate every function. This can now have 2 contrary effects. First, with a bigger hypothesis space the ANN might be able to find a better fitting hypothesis and therefore perform better. Second, this is linked to a greater possibility of overfitting, as the hypotheses can be more specific. Both aspects are shown in the plots. For the standard gradient descent, the network with 1 hidden layer becomes increasingly bad for large numbers of neurons, but not nearly as bad as for a 2-layer-network, where there is a much higher possibility of overfitting. On the other Hand, we can see that the resilient backpropagation performs in general better with 2 layers than with 1. It is also the overall best performing algorithm and therefore it is better able to handle a large hypothesis space.


Gradient descent
1 hidden layer
2 hidden layers
MSE
Neurons per hidden layer
2.5
2
1.5
1
0.5
0
5
10
15
20
25
30
35
40
45


Adaptive gradient descent
1 hidden layer
2 hidden layers
MSE
Neurons per hidden layer
0.075
0.07
0.065
0.06
0.055
0.05
0.045
0.04
5
10
15
20
25
30
35
40
45

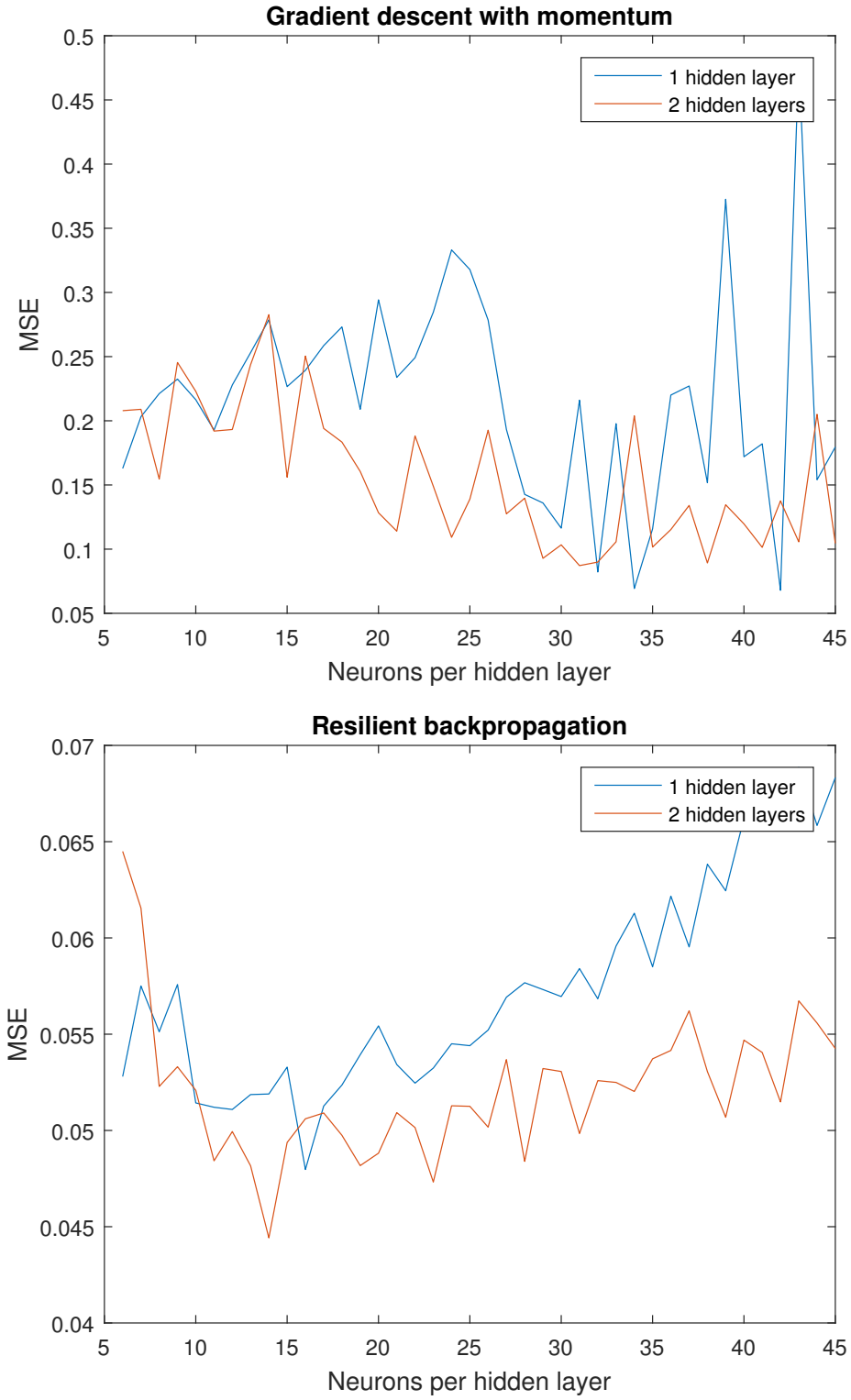


Figure 3: Topology-Performance plots per algorithm

Question 3

Matlab already implemented *early stopping* by default. Therefore the available data is divided into three subsets: the training, validation and test set. The first one is used to find the weights for the network. If the error on the validation set starts to increase for several iterations, which is an indication for overfitting, the process stops and the the weights which produce the minimum error on the validation set are returned.
Early stopping is not a 100% safe method as it is possible that the error on the validation set will decrease again after an unknown number of steps, but as it also reduces the training time and therefore comes somehow with negative cost, it is especially a good first approach for our (limited) computing power.

Furthermore, we considered to use *regularisation*. Regularisation is a mechanism to reduce the occurence of large weights. As large weights imply that some particular paths through the network are much more important than others, it could be more likely for overfitting to occur. Therefore, a function taking the input weights to a perceptron as an input is added to the original Error-Function that is minimized in the training. For example, in L2-Regularisation the squares of the weights are added together:

$$E_{new} = E_{original} + \lambda \sum_{all\ weights} w^2$$

λ is a parameter that sets the balance between the original minimization of the error and the minimization of the weights-function. Therefore, optimisation would include another parameter to optimise and therefore a multiplication of the computing time. Because the optimisation of our other parameters already took approximately two days, we could not spend triple/quadrupel/... of time.

Theoretically also other techniques, for example *dropout*, are possible.
In the dropout-method, random network nodes are dropped out in every training iteration with a constant probability for each node. Therefore, the network architecture is randomized and every architecture is just trained for a small random subset of the training data. This intuitively reduces the chance that the network is overfitted to the training data set as actually the network with n nodes represents an average of all the before trained dropout-architectures (out of the set of 2^n possible networks), where each network node is only fitted to parts of the training data. On the other hand, with small amounts of training data, the dropout-technique can actually reduce the classification. In particular, the paper from Srivastava et al. referenced in the lecture slides shows that dropouts only reduce the classification error for MNIST classification training data sets of size 5000 or bigger, for 100 training examples the classification error was bigger than without dropout. Therefore, we are not confident that the dropout-technique would improve our performance whereas the implementation effort would be big.

Finally, one could also consider *Data Augmentation* by using the noisy dataset additionally to the clean dataset.

Question 4

Disadvantages
First of all this leads to a necessary implementation of a decision function, which classifies on the basis of the outputs of the six networks. As in the assignment before, there is no obvious best solution for this problem and therefore the choice of decision function is another parameter to optimize.
The task for each of the six networks is to find a boolean classifying function. If we assume that we test the networks with the same topologies as the first big network, each network can represent *exactly* one boolean function as long as it has at least two layers. This leads to a huge risk of overfitting compared to the one network which only *approximates* an arbitrary function and is therefore more robust to noise. Furthermore, as the parameters are optimised differently for each output class, each network is more specifically fit to the training data and this might be another risk of overfitting.
Moreover, training six networks requires more resources than training one slightly (only the output layer size differs) bigger network. In general, this corresponds to a significantly longer training time. Of coures, the time consumption can be reduced by parallelizing the process of cross-validation because the networks are independent (see advantages).

Advantages
As the networks are trained more specificly to recognize one class, this approach could lead to a better performance.
Because the output layer is smaller than for one big network, there are less weights to optimize in each network for the same topology. As a result, parallel cross-validation of six networks will be faster than the cross-validation of one big network, as long as one has the required computing power.

Combination of Outputs
The combination of the output of the six networks is exactly the same problem which was solved in our last assignment. These could be reused with small modifications:

- **random choice:** This is still possible by using a threshold value. Every output with a bigger value than the threshold is considered for the random pick.
- **score-based decision:** Naturally, the output of the networks is also a score, i.e. this is algorithm can be used without any modifications.
- **depth-based decision:** The depth of a tree is a specific property of a decision tree and therefore it is not easy to find a similar and meaningful property for neural networks. Of course you could use the number of layers, but as this varies just between two values, the algorithm would probably not perform well as there would be a lot of ties.
- **error-based decision:** As the error of the networks can be calculated just as the errors of the trees, this strategy is also possible to use without modification.

3.2 Part B: Questions

Question 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 these two learning algorithms.

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 would be different.

To summarize, it is not possible to say whether a specific learning algorithm is better than an another based on the performance for a given problem. Rather, analyzing the performance of the algorithms one should always consider the specific problem domain, the relevant performance metric and the available dataset.

Question 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.

Question 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 emotion. As a result, the sample for the t-test is constituted by 10 six-elements vectors. To compare the two algorithms in this way we have to apply 6 different t-tests (one per emotion), 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.

Question 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 gets smaller. Moreover, as the degree of freedom (df) is directly proportional to the sample size, the corresponding threshold will be 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 folds leads to a bigger t-statistic and a smaller threshold which in turn might lead us 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 folds 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 to specify that there is no argument to believe that 10 folds represent the optimum for this specific test.

Question 5

In the case of the decision trees learning, adding new emotions requires to grow additional decision trees (in particular as many decision trees as emotions added). It should be observed that retraining is required since adding new emotions to the dataset enriches the set of negative examples of the pre-existing classes. On the other hand, in the case of neural

networks it is required to add as many additional output layers as new emotions, to retrain the network and to optimize it again with cross-validation.