Machine Learning: Neural Networks

Alexandros Mousafeiris, Charles Gbadamosi, Nikolas Vayias

November 18, 2014

Contents

1 Introduction

This project's aim is to use machine learning techniques combined with a neural network model and training method in order to discern between six human emotions on facial expressions. This is done based on forty-five distinct muscle groups used as attributes for the decision tree model. The information gain metric will then be used to compute the trees, and several metrics will be computed to assess the trees' performance over the clean and noisy/erroneous training sets provided.

2 Implementation Details

The majority of the network simulation and training is done by using the Neural Networks toolbox provided by MATLAB. Any changes made to the parameters used to create our neural networks will be discussed further ahead.

2.1 Cross-Validation

Ten-fold cross validation is done to resemble the specification as closely as possible. The data set is first split into ten segments. One of the segments is **fixed**, and is used for validation (optimising NN parameters and avoiding overfitting). The other 9 are then further split into the training and testing examples. A network is created on the training set, optimised using the validation set, and then tested using the testing fold provided each time. This returns the error rate for the fold.

2.2 Choosing Optimal Parameters

We found the optimal node number, training function, training parameters for each fold and saved a network with that configuration into a list of networks. After cross validation, we select the network with the lowest error, i.e. the highest performing one.

2.3 Optimising Functions

Optimisation for every parameter is stored separately, as the strategy for optimising each one may be different. The main difference being parameters that are non-monotonic over a changing error-rate, and thus necessitate iteration through increasing discrete values. We accomplish this by discretizing the domain we're optimising over. For monotonic functions such as **traingda**, we employed binary search instead.

3 Performance Measures

We calculated our classifiers various performance measures including Error Rate across all data and various performance metrics per label. Cross validation involved dividing our data set in 10 folds initially into a training/test set and a fixed validation set. The remaining on 90% of the data, and measuring performance on the remaining 10%.

3.1 Error Rate

Cross validation yielded an error rate of: **18.20%** when **clean** data was used for training and validation, and a significantly increased **30.67%** when using **noisy** data. The final networks used were the ones from the fold that were trained to yield the minimum MSE, **12.36%** and **21.35%** respectively.

3.2 Confusion Matrices

Below are the confusion matrices for the **clean** and **noisy** data sets respectively. This is the resulting of averaging the confusion matrix produced by each k-fold of the cross-validation process.

A \P	1	2	3	4	5	6
1	7.6	1.6	0.5	0.6	1.1	0.2
2	0.7	14.9	0.1	1	0.6	0.3
3	0.3	0.5	8.8	0.1	0.2	0.9
4	0	0.7	0.3	17.7	0.1	0.2
5	0.8	2.5	0.1	0.4	7.4	0.4
6	0	0.2	0.9	0.5	0.4	16.4

A \P	1	2	3	4	5	6
1	0.1	1.8	2.1	1.2	2	0.9
2	0.1	13.8	1.3	0.6	0.4	0.6
3	0.4	1.7	10.6	1.1	1	1.4
4	0.1	1.2	0.8	15.4	0.4	0.5
5	0.4	1.3	1.4	0.3	5	1.6
6	0	0.5	0.8	0.7	0.7	16.8

3.3 Precision, Recall, G_1

These are the performance metrics calculated by the confusion matrices given above. Every metric is label-specific, and we've sectioned them by label to make the noisy-clean comparison easier. All of the figures are rounded to the second decimal point.

3.3.1 Happiness - Label 1

Clean

- **Precision Rate:** 80.85%
- **Recall Rate:** 65.52%
- F_1: 72.4%

Noisy

- **Precision Rate:** 9.09%
- **Recall Rate:** 1.23%
- F_1: 2.17%

3.3.2 Sadness - Label 2

Clean

- **Precision Rate:** 73.04%
- **Recall Rate:** 84.66%
- F_1: 78.42%

Noisy

- **Precision Rate:** 67.98%
- **Recall Rate:** 82.14%
- F_1: 74.39%

3.3.3 Surprise - Label 3

Clean

- **Precision Rate:** 82.24%
- **Recall Rate:** 81.48%
- F_1: 81.86%

Noisy

- **Precision Rate:** 62.35%
- **Recall Rate:** 65.43%
- F_1: 63.86%

3.3.4 Disgust - Label 4

Clean

- **Precision Rate:** 87.19%
- **Recall Rate:** 93.15%
- F_1: 90.08%

Noisy

- **Precision Rate:** 79.79%
- **Recall Rate:** 83.70%
- F_1: 81.70%

3.3.5 Fear - Label 5

Clean

- **Precision Rate:** 75.51%
- **Recall Rate:** 63.79%
- F_1: 69.16%

Noisy

- **Precision Rate:** 52.63%
- **Recall Rate:** 50.00%
- F_1: 51.28%

3.3.6 Anger - Label 6

Clean

- **Precision Rate:** 80.61%
- **Recall Rate:** 78.22%
- F_1: 79.40%

Noisy

- **Precision Rate:** 77.06%
- **Recall Rate:** 86.15%
- F_1: 81.36%

4 Questions and Considerations

4.1 Question 1 - Topology and Parameter Optimisation

4.1.1 Topology

The topology of a feed-forward neural network refers to two things. The number of layers in the network (not counting the input layer), and the number of hidden neurons in each (hidden) layer.

A neural network with one layer is not adequate to train a classifier if a data-set is not linearly separable. As this is very commonly not the case, we assumed that our domain was also not linearly separable. This called for the use of at least one hidden layer (and one output layer). Our targets consist of six binary outputs, or six **boolean functions**. One hidden layer is enough to perfectly model any boolean function. While we could also model the function with two or more hidden layers, but this would greatly increase the training times. Due to the time constraints for the task and the nature of the classifier, we decided to choose one hidden layer.

The number of hidden neurons is very vague as a training parameter. Because of the biological inspiration for the model, there is no intuitive way to tell why a lower or higher number of hidden neurons would be appropriate or desirable for any specific problem domain. We also assumed the the error rate changing as a function of a changing number of hidden neurons is non-monotonic, so we had to test linearly over a set of values. The way our implementation did this was to cross-validate our neural networks for an increasing number of hidden neurons in the hidden layer. Then, it would average the error rate of the resulting neural network from each fold, and compare the average error rates for each number of hidden neurons. It will then pick the network trained in the fold having the lowest error rate, and thus highest performance. This is the same procedure by which all our optimal training parameters were picked.

4.1.2 Training Functions and parameters

We decided to optimise over four training functions and their training parameters. All four functions are different variations of **gradient descent** coupled with the **backpropagation training algorithm**.The MATLAB functions are **traingd**, **traingda**, **traingdr**.

For the simple gradient descent we had to optimise the learning rate factor η. This represents the step size for the function, which tries to approach the a minimum by going in the opposite direction to the rate of error increase $\frac{\delta E}{\delta w}$. We run into a few difficulties before realising that this rate of change is non-monotonic. Once we knew this, we started iterating through values step wise. In order to do this, we had to discretise our domain to be able to iterate. We used the default discretisation function provided, with the domain $[0.0001, 0.1]$. The final value for our optimal network was 0.09.

In order to save time we made the, admiteddly not well-founded, assumption that our optimal learning rate for **traingd** would be optimal for the other variants of gradient descent. We made this assumption since the other gradient descent variants are mostly about **adjusting** lr rather than defining it, so we were content with being at a slightly wrong starting position. The cost of that is ending within local minima in the error plane (instead of real minima).

For **traingdm**, we tried to optimise the momentum factor. The main motivation for using this variant is to converge to the minimum faster as gradient descent "travels" very slowly in valleys in the error plane. Unfortunately, we didn't get any consistent values to use.

When optimising **traingda** and **trainnrp** we used a slightly different approach since we reasoned that unlike the other two variants, these two are not monotonic. Subsequently, we decided on binary search to get $O(\log n)$ complexity on our optimising functions. The ranges used for each parameter are the following:

- **delta_inc:** $[1.1, 1.3]$
- **delta_dec:** $[0.4, 0.6]$
- **lr_inc:** $[1.0, 1.5]$, stabilised around 1.1.
- **lr_dec:** $[0.5, 1]$, stabilised around 0.7, with greater variance.

It should be noted that there is two properties of the resulting neural networks which we did not alter. The first one is the pre-processing and post-processing functions, as the default options provided all the desired functionality. Specifically **mapminmax**, by mapping our inputs and outputs to $[-1, 1]$ prevented the issue of vanishing/exploding gradients and mitigated overfitting. The other property we didn't alter was the network's activation functions.

4.2 Overfitting

Overfitting is described when a function ends up describing the underlying noise in a set of values instead of the relationship between them in a statistical model. In the context of this project, this is a problem that occurs when the error on the validation/testing set starts increasing (after having decreased to a, not necessarily non-local, minimum), despite still descending on the training set. As the definition would suggest, this usually happens because of the training examples not being representative of the whole problem domain. This is especially obvious with noisy data sets, as will be shown in the next section.

One of the most common methods used to avoid this problem, which is what we employ here, is the **early stopping** technique. Instead of allowing the neural network to optimise its parameters for the defined number of epochs (iterations), the training instead stops when the performance measure starts getting worse. "Starts getting worse" when using mean squared error implies that the training will stop after n times that the error on the validation set has increased.

Unfortunately, this method has the disadvantage of not recognising local minima. If n is given too small a value, then the training might stop too soon and not actually reach the true minimum for the error in the validation set. While training, we noticed that this did happen for a few folds. However, we decided against raising n as it would directly increase the time required to train the network, and the classifier seemed to be performing relatively well on the noisy data set, with some exceptions.

Another measure we took is limiting the number of epochs for each training session to 25000 epochs, although the primary motivation for doing this was to put an upper limit on the computation time required.

Another method to avoid overfitting is to regularise the inputs/weights of our network. While we scaled our inputs using **mapminmax**, we did not normalise their distribution.

4.3 Noisy and Clean Datasets

There is a very clear difference in the performance of our classifiers when using the two different example sets. As was mentioned in the Performance Measures section, there was about a 70% increase in the error rate when using the noisy set. This was somewhat mitigated by choosing the *optimal* networks from each dataset, but the clean data-set-trained classifier was still ahead by about 50%.

If we look at the confusion matrices we can see that both classifiers are performing quite well, with the clean data-trained one being ahead as predicted. One of the most prominent differences is the ability in classifying label 1 (happiness) correctly. The F_1 measure drops from 72.4% to 2.17%. Over the two data sets. Our neural network seemed completely unable to generalise over the rules for label 1 prediction. If we look at the actual predictions, it seems like it was being predicted almost randomly, with a bias for sadness, surprise and disgust. This points to either a fault in our code, or simply a very non-representative data set for the first label. A great decrease in performance was noted in the Decision Trees part of the assignment as well, so we assumed that the problem did not lie within our code.

Because of the relatively even distribution of false positives and negatives this is also unlikely to be an overfitting problem, as there was no erroneous generalisation. Rather, there was no attempt at generalisation.

The clean qualifier has also confused a significant amount of label 5 examples (fear) for label 2 (sadness). This is most probably a result of overfitting. Due to the added noise in the second data set, this problem is not as pronounced. The distribution of falsely classified examples is more equal.

While the clean qualifier has better measures for every single measure, label 6 predictions are slightly better for the noisy data set. This might be a result of the classifier being unable to classify label 1 examples properly. There is a chance a lot of those examples were actually label 6 examples, skewing the statistics given.

Overall, the discrepancies between the clean and noisy measures are as predicted. There is a decrease in accuracy for every label (except 6), but most pairs have relatively close values with the exception of label 1.

Another conclusion that can be drawn is that most of the error increases in the noisy set are due to simply erroneous predictions within the set, and not overfitting.

4.4 Question 3 - Six NNs or one NN

If we created six neural networks, one for each label, there would only be three cases for each prediction. In the first one, only one network classified the example positively. In that case, we could simply choose the only positive classification. The other two cases are slightly more complex. In the second one, more than one network give us a positive classification.

There are numerous ways to deal with this, but a sensible one would be to compare the performance of the networks giving a positive classification and picking the network with the best metrics. If we were interested in the precision or recall depending on our application, we could pick based on the network giving the highest one. Otherwise, we can just pick the network resulting in the lowest MSE, as that classification is the most likely one to be correct. In the third case, none of our six networks managed to classify our example. In such a case, we can do two things. The first method is to discard the example completely, exclude it from the stats and include a classification rate metric which gives the percentage of examples our networks are able to classify at all. The second option presents multiple sub-options. We will consider two. We can just classify the example randomly. This would reduce the performance of our system significantly, as now there is a set of examples that is classified correctly only one sixth of the time. The other option comes at the risk of skewing our results. We can pick the label with the worst performance, as it has the highest chance of having incorrectly classified the example as negative. This may result for a large amount of false positives, and thus reduced precision, for a specific label.

One of the advantages of using multiple NNs is the need for optimisation. Because of the problems of overfitting and the presence of local minima, we can get greater accuracy by optimising the weights of each network separately and then averaging their performance. Plus, we are getting a better statistical assessment of each network's reliability. This distinction also allows for a better strategy when choosing which network's prediction to pick, as their individual performance is now known.

As another advantage, it is easier to distribute the training of multiple networks, simplyfying the problem of resource allocation.

The main disadvantage is that since we are now using our entire data set six times, the time required to train all networks is now greatly increased, even when accounting for the reduced complexity of each individual network.

4.5 Picking the optimal network

As was described in the answer to question 1, our process of picking the optimal network is quite simple. When cross-validating for specific parameter values, we simply ”save” the folds that produced the best performance in terms of minimising the error rate, and then pick that network to try on our testing set. Refer to section **2.2** for more details.