MINI PROJECT:

Using Neural Network for Supervised Learning

Team member(s): Tomas Tellier (21020297x)

Semester: Fall 2021 (semester 1)

Course: EE4014A

Last edited: 26.11.2021

Department of Electrical Engineering
The Hong Kong Polytechnic University

Table of Contents

Introduction

The Mini Project is part of the course *EE4014A Intelligent Systems Applications in Electrical Engineering* at The Hong Kong Polytechnic University. The course aims to introduce the fundamentals and basic applications of intelligent systems to electrical engineering students, such as artificial neural networks (ANN), fuzzy systems, evolutionary computing and more.

The essence of the project is to apply the concepts taught in the course to develop a simple but functioning artificial neural network for supervised learning and use the developed model on a data set of choice for training of the network. Furthermore, the results of the training are to be analyzed to increase the performance of the model, as well as to show understanding of the concepts in artificial neural networks and supervised learning.

This paper introduces the data chosen to train a developed artificial neural network for classification of samples and gives a detailed description of the script of the model. Additionally, the results of the training are presented and discussed in order to interpret and increase the performance of the neural network.

Data

The data set used in this project, named “Wine data set”, was obtained from the archives of *The UCI Machine Learning Repository*, and contains 178 labeled samples, each with 13 attributes. The samples represent wines derived from three different cultivars in the same region of Italy. All attributes are continuous. Their attributes represent the following constituents:

1) Alcohol	5) Magnesium	9) Proanthocyanins	13) Proline
2) Malic acid	6) Total phenols	10) Color intensity	
3) Ash	7) Flavonoids	11) Hue	
4) Alkalinity of ash	8) Non-flavonoid phenols	12) OD/OD315 of diluted wines	

Table 1: attributes of data set

Since the wines are derived from *three* different cultivars, the data set has *three* different classes. For simplicity, the cultivars, or classes, are not named, but numbered (1-3). Thus, the goal will be to train the neural network to recognize which cultivar the wine is derived from based on its cultivars. In other words, this is a multiclass classification problem.

The weakness of the chosen data set lies in the number of samples it contains. There is no “golden rule” to find the correct number of samples needed to optimally train an ANN model, as it depends on several factors: the type of model, its complexity, the data set’s quality and more. Despite this, several good suggestions and rules of thumb can be found in research papers on the internet. One research paper found on the internet suggests that the number of samples used to train an ANN model should be at least ten times the number of weights in the network (Alwosheel et.al., 2018). If this rule was to be used in this project, the requirement would be to have at least 200 samples, given that the network has seven hidden nodes and one output node. Thus, the number of samples in the data set does not match the requirement, especially if a part of the data set is to be used for testing.

Nevertheless, this is just a suggested rule of thumb, and not an actual requirement. A simple ANN can require relatively few samples to be trained optimally – the ideal method to find an answer to this question is to try.

Script: Methodology and Design

This chapter gives a detailed description of the developed script of the ANN and argues the details of how the project is implemented.

For programming of the ANN model, *TensorFlow* and *Keras* was used in Python. TensorFlow is an open-source library for dataflow programming and is used for machine learning applications. Keras is an open-source neural network library that is capable to run on top of Tensorflow and focuses on being user friendly.

The script is based on the example given in class: *supervised_MNN_car_rescaled*, and some inspiration for other functions of Keras has been found on the internet. The full script can be found as an attachment to the project report.

Import of Data

For the import of data, the function *genfromtxt* from the *Numpy library* was used. This function loads the data into a matrix and makes the data easy to handle. Inspired by the code example given in class.

```
data = np.genfromtxt('wine_data.txt', delimiter=',')
```

Figure 1: import of data

Splitting and Rescaling of Data

In order to feed the data to the neural network, the data set must first be split up in two groups: the attributes (inputs), and the classes (expected outputs).

For increased performance, the input data should be scaled to a more standardized range. Testing showed that scaling decreases the noise in the model's accuracy and gives an overall better performance after several training epochs.

The next step in the code uses a Keras function to convert the expected output, or the class labels, into the *one-hot* format (standard = ([0,1,2]) → one-hot = ([1, 0, 0], [0, 1, 0], [0, 0, 1])). This allows to use three output nodes instead of one, and better performing activation functions for the output nodes. Testing showed that this increased the model's accuracy on the

testing data drastically just after few epochs, compared to when using only one output node, a linear activation function and the class labels on standard format.

Part of this code was inspired by the example given in class, while the rest came from internet research. (Li, 2019)

```
y_expected = data[:,0]
x_input = data[:,1:15]

rescaling_x = True
if rescaling_x:
    for i in range(0,13):
        x_input[:,i] = x_input[:,i]/max(x_input[:,i])

for j in range(0, len(y_expected)):
    y_expected[j] -= 1

y_expected_onehot = keras.utils.to_categorical(y_expected)
```

Figure 2: splitting and rescaling/reformatting of data

Network Structure

The next few code lines in the script define the structure of the neural network.

In figure 3, the first line shown tells the program to build the network layer by layer, so that it becomes a *feedforward* neural network. The second line of code defines the network's initial weights. Again, there is no textbook answer for what these values should be, but depending on the activation functions used in the model, the initial weights should be of relatively small values, to avoid slow learning and poor performance. A rule of thumb can be to use random values in the range given by the following mathematical expression:

$$\pm \frac{1}{\sqrt{number\ of\ connections\ into\ a\ node}}$$

For simplicity, it was chosen to follow this rule by using the number of *input* nodes.

The next code lines define dimensions of the neural network. Deciding the size of the input layer is straightforward, as this number is defined by the number of attributes from the data set. For simplicity, it was decided to implement only one hidden layer in the model. As the input data is not too complex, there is no need to "over-engineer" the model.

Deciding size of the hidden layer, however, may require some testing. A decent number to start with is the mean of the model's number of input and output nodes. Experimentation shows that this number of hidden nodes gave good results, compared to adjusting the number up or down.

As previously mentioned, the number of output nodes was chosen to be 3. This is due to the one-hot format of the class labels. Experiments with only one output node were conducted, and switching to three nodes and the one-hot format showed great increase in the accuracy of the model's predictions.

Finally, the type of activation functions must be chosen. As the attributes and sample labels are all scaled to the range [0,1], it is logical to use an activation function that outputs a value in the same range, instead of using a linear or similar function. Several activation functions were tested for both the hidden and the output layer. The combination that gave the best results was using the *sigmoid* activation function in the hidden layer, and the *softmax* activation function in the output layer.

The *Dense* function creates a densely connected layer, meaning that all outputs from the previous layer are fed into all nodes. This is the most common form of neural network layer.

This part of the code was inspired by the previously mentioned example given in class.

```
model = Sequential()
keras.initializers.RandomUniform(minval=-1/math.sqrt(13), maxval=-1/math.sqrt(13))
model.add(Dense(units=8, input_dim=13, activation='sigmoid'))
model.add(Dense(units=3, activation='softmax'))
```

Figure 3: network structure

Compilation

The *compilation* is the final step in creating the ANN model. This configures the model for training. The function contains a few important parameters that affects the training and thus the performance of the neural network.

Firstly, the type of error function must be defined. The error function defines how the error is calculated and used to update the weights during training. As this course only teaches about the *mean squared error* function, it was decided not to experiment with other error functions. However, some experimentation was conducted with the choice of optimizer. In the code

example given in class, which again was used as inspiration for this part of the script as well, *Stochastic Gradient Descent* (SGD) was used. Internet research suggested to try the *Adam* method, which is supposed to be computationally efficient and has little memory requirement. The Adam method is a tweak of the SGD method. Further testing showed that using this method resulted in more accurate predictions.

```
model.compile(loss='mean_squared_error', optimizer=keras.optimizers.Adam(learning_rate=0.05), metrics=['mean_squared_error'])
```

Figure 4: compilation

Training

The last step of the code is the training of the model. The inputs and expected outputs are defined, as well as a few other essential parameters.

Validation split splits the data set into two groups: a training group and a testing/validation group. This allows the model to be tested after each epoch, so that its accuracy can be properly analyzed with unknown data. Both the course and research on the internet recommends using around 20-25% of the samples for testing. This recommendation was followed for the project.

The next two parameters are crucial for the accuracy of the model, as well as for its training time. These parameters determine the number of epochs in the training and the size of each batch. Choosing optimal values for these parameters can be quite challenging, as the choice depends on what the developer/user values the highest: speed or accuracy?

A balance was attempted to obtain between the two aforementioned variables. With experimentation it was found that 300 epochs and batch sizes of 30 samples gave good prediction accuracy, with a runtime of around 4.5 seconds.

```
training_history = model.fit(x=x_input, y=y_expected_onehot, validation_split=0.2, epochs=300, batch_size=30, verbose=1)
```

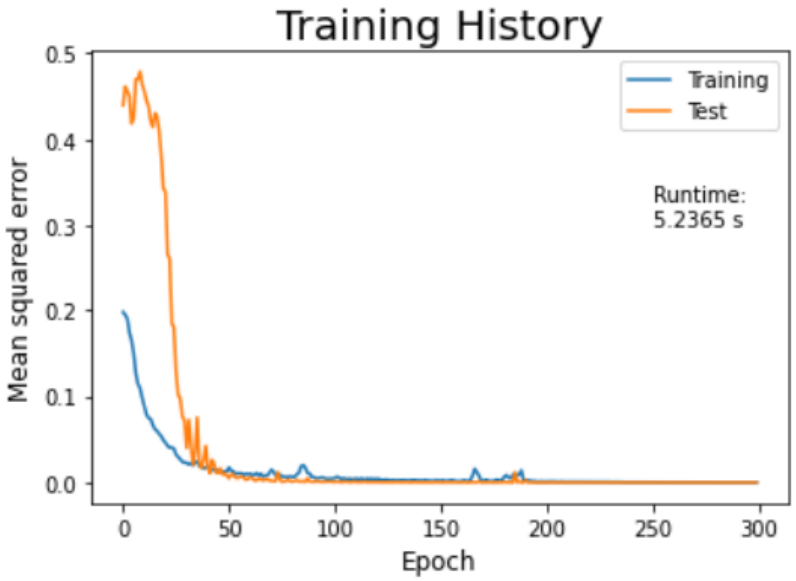
Figure 5: training

Discussion and Results

This chapter presents and discusses the results of the training of the ANN model.

Results

Figure 6 shows the results of the training both graphically and numerically. One can clearly see that the training has led to good results, with a mean squared error of approximately 0.00005 on the *test samples* after 300 epochs. As the error is that low, one could consider lowering the number of epochs to around 120, to reduce to runtime of the model, as this would also give good prediction accuracy. However, some fluctuations in the error of both the training and testing can be observed between 150 and 200 epochs, which might make it safer to stay above 250 epochs.




```
Epoch 300/300
 - 0s - loss: 3.4549e-04 - mean_squared_error: 3.4549e-04 - val_loss: 4.8204e-05 -
val_mean_squared_error: 4.8204e-05
```

Figure 6: results

Under- and Overfitting

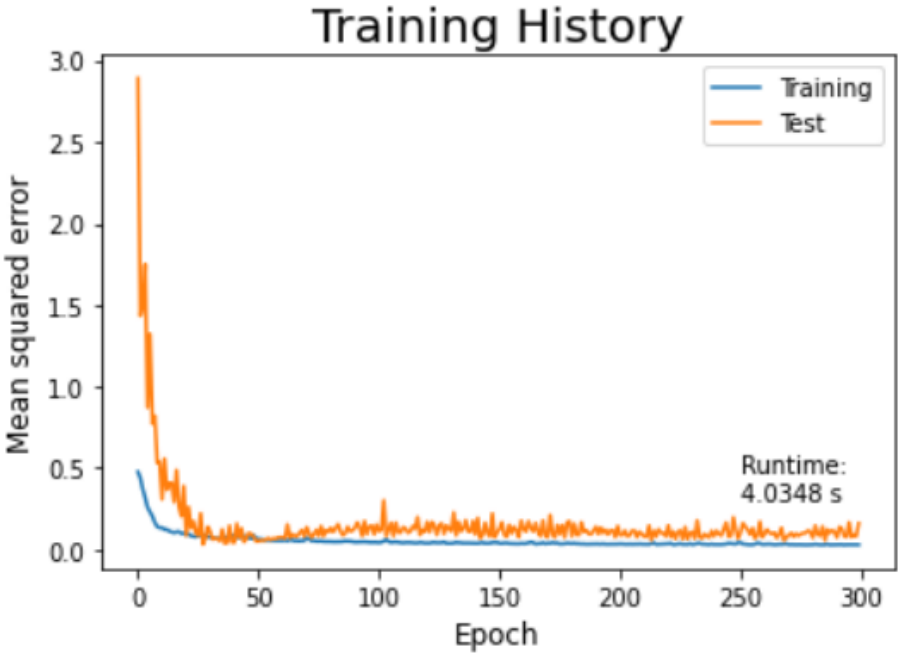
The detection of under- and overfitting is crucial in the analyze of the results and performance of an ANN model. It is desirable to avoid both cases, as they both would give a poorly performing model.

It is safe to say that this model is not underfitted, as it has managed to model the training data well and predicts test data accurately. It seems like overfitting has also been avoided, as the predictions of the model are accurate also on the testing data. However, detecting the

difference between an optimal and an overfitted model might be easier if the data set contains far more samples, and both the input data and the model are more complex. As both the model and the data used for this project are very simple, avoiding overfitting might be a bit too easy.

Format of Expected Output Data

The format of the sample labels had a significant impact on the model's prediction accuracy. As previously mentioned, it was decided to format labels to the *one-hot* format before feeding the data to the neural network, for increased performance. To better describe the importance of the change of the format, figure 7 shows the results of the training *without* formatting the sample labels. Compared to figure 6, the training results of the model with formatted labels gave significantly more precise and stable predictions.




```
Epoch 300/300
 - 0s - loss: 0.0303 - mean_squared_error: 0.0303 - val_loss: 0.1597 -
val_mean_squared_error: 0.1597
```

Figure 7: results with sample labels in original format

Conclusion

The developed neural network model for this project has given good results for the chosen data set. Experimentation with functions and tuning of parameters has given more understanding of how neural networks work, as well as their limitations.

The complexity of this project was quite limited by the data set. Since the set contains relatively few samples, and the data was quite easy to handle, this may have made the project not challenging enough.

For improvement of my work, I could have done more systematic analytical experimentation with the parameters of the model. Even though my tuning led to good results, even better results could probably be achieved by better testing. Another improvement could be to take a closer look at the Keras library and find other functions to make the model more efficient.

Self-evaluation

I specifically chose to do this project alone to force myself to learn more about coding, analyze of data and, of course, neural networks. This turned out to be a good idea, as doing it alone was very manageable. It is also nice to work on a project independently, as most school projects I have worked with have been in groups.

The project has given me more understanding of every component of a neural network, as well as what it takes to program a model. However, I see now that I could have chosen a more challenging approach to the problem, for instance by choosing a more complex data set or writing more of the code from scratch. Still, I would rate my performance as “average plus”, for having obtained good results and having analyzed the data to increase the quality of my work.

Bibliography

Forina, M. et.al. (1991) : *Wine Data Set* (01.07.1991)
https://archive.ics.uci.edu/ml/datasets/Wine [23.11.2021]

Alwosheel, Ahmad et.al. (2018): *Is your dataset big enough? Sample size requirements when using artificial neural networks for discrete choice analysis* (p. 167-182)
https://ideas.repec.org/a/eee/eejocm/v28y2018icp167-182.html [24.11.2021]

Li, Lorraine (2019): *Introduction to Multilayer Neural Networks with TensorFlow's Keras API* (12.6.2019)
https://towardsdatascience.com/introduction-to-multilayer-neural-networks-with-tensorflows-keras-api-abf4f813959 [24.11.2021]

Ye, Hongbo (2021): *EE4014A Intelligent Systems Applications in Electrical Engineering* (class presentations)

Attachments

Attachment 1: OneDrive folder including script, data set and video:
https://connectpolyu-my.sharepoint.com/:f:/g/personal/21020297x_connect_polyu_hk/EshQ9g2iJQJPr8tdgF_7WcUBug7kTLKzrH4Dxk_QIZxX8A?e=QGQz52