Driving Behaviour Analysis

Statistical Methods For Machine Learning

Academic Year
2022-2023

Giulia Testa

giulia.testa3@studenti.unimi.it

Contents

Declaration

1 Introduction

The objective of the project was to create a fully functioning classifier, that, given the values - expressed through triples of coordinates (x, y, z), of the accelerometer and gyroscope sensors mounted on everyone's smartphone - categorises the behaviour of a subject, using a Swift mobile application, while driving around. It adapts the requirement of the "Neural Networks" project (SMML project link) for the 2021-2022 course of Statistical Methods for Machine Learning and answer the need of a classifier for the project of the course Sviluppo di Applicazioni per Dispositivi Mobili.

1.1 Classification

The classification algorithms are machine learning systems with the objective of learn functions that predict to which class or label a data point belongs to. These techniques are taught using a *training dataset*, made of examples of data points already correctly labelled. The dataset is fed to the model, and it learns from it.
Once the model is trained, it needs to be validated - before using it, it is necessary to check whether it has learned to classify correctly and if its prediction is reliable. In order to make the model validation, another dataset is used, called test set. If the model presents good results from the validation, it can be used.

1.2 Neural Network

Neural Networks are a class of models in the machine learning literature that mimic the organization and functioning of the human mind: it is made of artificial neurons working together to solve a problem. They are a network of interconnected nodes, a directed graph, where each node performs computation of the outputs of the previous node. They are quite useful for classification problems.

1.3 Convolutional Neural Network

A convolutional neural network is a subtype of the neural network specialised in finding recurring patterns inside datasets. They contain typically at least these three layers: a *convolutional layer*, a *pooling layer* and a *totally connected layer*.
The first one represents the core function of the model: it performs the convolution over the input data - in practise, the idea is to move a small filter, the *kernel*, across the input to extract features.

1.4 Feed Forward Neural Network

A Feed Forward Neural Network is one of the simplest neural networks, in which the information is processed in one direction (from input to output and never vice versa). The one designed for this project has an input layer, a *hidden layer* and an output layer.

1.5 Loss function

The loss function is used in machine learning to measure the quality of a model's prediction. It quantifies the difference between the predicted class and the actual label of a data point. In this project, the *sparse categorical crossentropy* has been used. It is a variation of the categorical crossentropy designed for problems with integer labels, like this one, that are not one-hot encoded. The crossentropy heavily penalises outputs that are extremely inaccurate, with very little penalty for correct classifications. Minimising it leads to good classifiers.

1.6 RMSprop optimiser

In machine learning, the goal of an optimiser is to minimise the loss function, changing the parameters of the model. In this project, the *RMSprop optimiser* has been chosen. RMSprop (*Root Mean Square Propagation*) optimiser is a gradient-based algorithm that adapts the learning rate to prevent the model from divergence.

1.7 K-fold crossvalidation

The k-fold cross validation is a validation method used to find out how well a machine learning model can predict the outcome of unseen data. It assesses the robustness and the performances of a model. It consists of equally subdividing the dataset into *k folds* and of training the model evaluated k times. At each iteration, one fold is used as test set and the other ones are used for the training.

1.8 Sliding window

The sliding window method is an iterating method over a dataset used in the machine learning context that creates overlapping windows of a fixed size, that will be processed individually, reducing considerably the overall complexity of the model.

2 Data Preprocessing

2.1 Data gathering

The dataset used to define the classifier is made from a combination between two different dataset. This comes for the necessities regarding the mobile application I have developed for the driving behaviour analysis: the classifier must recognise a normal and acceptable driving behaviour and a risky and aggressive one as well.
The main dataset was downloaded from Kaggle Driving Behaviour Analysis, that is a "dataset for modelling risky driver behaviours based on accelerometer and gyroscope data". The dataset contains the data associated with four different classes: sudden acceleration, sudden right turn, sudden left turn and sudden break.
The second dataset comes from Kaggle Driving Behaviour. This dataset has data labelled into three classes - slow, normal and aggressive. From this dataset, after having filtered out the data points labelled with aggressive driving behaviour, the remaining data points have been appended to the other dataset with a new label, class 5, to represent the normal driving behaviour.

2.2 Feature extraction

After the preparation of the final dataset to be used for training and testing, it also need some pre-processing. Since the classifier will be embedded in the mobile application, it needs to provide a prediction almost in real-time, without having the whole dataset as input, but some small chunks of it.
In this scenario, the sliding window technique comes to a rescue. It has been applied over the entire dataset and window_number windows were created.

```
def sliding_window(data):
    labels = _get_scaled_labels(data)
    data = _get_data(data)
    windows_number = len(data) - WINDOW_SIZE + 1

    windowed_data = np.zeros((windows_number, WINDOW_SIZE, data.shape[1]), np.float32)
    windowed_labels = np.zeros(windows_number, np.int8)
    for i in range(windows_number):
        index_range = range(i, i + WINDOW_SIZE)
        windowed_data[i] = data[index_range]

        windowed_labels[i] = Counter(labels[index_range]).most_common(1)[0][0]

    return windowed_data, windowed_labels
```

The Kaggle Driving Behaviour Analysis dataset suggests 14 seconds as best window size and that's what has been used in the project: since the frequency is 2 samples per second, WINDOW_SIZE is 2 * 14 = 28.
To label the extracted feature the *majority rule* has been used. The majority rule consists of choosing the label that recurs more frequently. This is done using the Counter built-in collections package class and the method most_common that return a list of the n most common elements and their counts from the most common to the least.
It should be noticed that the labels were in the range of $[1, 5]$, but the model starts counting from 0 and that's why there was the need to scale them down, using the method _get_scaled_labels , that simple subtracts one to all the labels.

2.3 Data splitting

Given the extracted features, the dataset must be split to obtain the *training set* and the *test set.* This has been achieved using the train_test_split function, provided by the *sklearn* package.

```
def data_split(data, label):
    return train_test_split(data, label, train_size=0.8, test_size=0.2)
```

The train_test_split method split arrays or matrices into random train and test subsets. In this case, the ratio between training and test is the most common one, 80 : 20.

3 Model definition and training

Once all the pre-processing is done, the model is ready to be defined and trained. Let's discuss how the Convolutional Neural Network and the Feed Forward Neural Network have been defined first.

3.1 Convolutional Neural Network's definition

The following snippet of code is the method that defines the convolutional neural network:

```
def define_cnn():
    model = Sequential()
    model.add(Conv1D(32, kernel_size=3, activation="relu", input_shape=(WINDOW_SIZE, 6)))
    model.add(BatchNormalization())
    model.add(MaxPooling1D(pool_size=2))
    model.add(Conv1D(64, kernel_size=3, activation="relu"))
    model.add(BatchNormalization())
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(128, activation="relu"))
    model.add(BatchNormalization())
    model.add(Dense(NUM_CLASSES, activation="softmax"))

    optimizer = tf.optimizers.RMSprop(learning_rate=0.0001)

    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model
```

The model definition starts with defining the model Sequential, meaning that it creates models layer-by-layer by stacking them, but it does not allow you to create models that share layers or have multiple inputs or outputs. Then, multiple layers have been added to the CNN model:

- Conv1D: the convolutional layer of the model
- BatchNormalization: it helps in stabilising the training process and potentially improve the model generalisation capabilities.
- MaxPooling1D: it reduces the spatial dimensions, maintaining the relevant information.
- Flatten: it flattens the outputs from a multidimensional data structure to a one-dimension array.
- Dense: it applies the *activation function* over the outputs. The activation function scores pixel values according to some measure of importance and the *ReLU activation* says that negative values are not important and so it sets them to zero; while the *SoftMax activation*, creates the probability distribution of the classes and it used in the output layer.

Then, the model has been compiled, with

- *optimizer*: the *RMSprop* with a learning rate 0.0001
- *loss function*: the sparse categorical crossentropy.
- *metrics*: the accuracy

3.2 Feed Forward Neural Network's definition

The following snippet of code is the method that defines the feed forward neural network:

```
def define_ff():
    model = Sequential()
    model.add(Flatten(input_shape=(WINDOW_SIZE, 6)))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(NUM_CLASSES, activation='softmax'))

    optimizer = tf.optimizers.RMSprop(learning_rate=0.0001)

    model.compile(
        optimizer=optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model
```

It is much simpler than the convolutional neural network. It consists of

- an input layer, that only prepares the input data for the following layers.
- a hidden layer, with 64 neurons and *ReLU activation function.*
- an output layer, with NUM_CLASSES, which, in this case is 5, and *SoftMax activation function.*

3.3 K-fold cross validation and training

To train the model, a 5-fold cross validation is implemented exploiting the split (data, labels) method from the class KFold in the *sklearn* library.
The KFold must be instanced before, with the number of folds and the shuffle parameter set to True to perform data shuffling before splitting into folds. The split method returns the train and test sets as arrays of indices that needs to be extracted from the data points and the labels to obtain the folds.
Once the folds were created, in each iteration, the model was trained using the model.fit function, with the following parameters, along with the data points and labels:

- epochs: an epoch is an iteration over the entire data points and labels provided.
- batch_size: the batch size is the number of samples per gradient update.
- validation_data: is the data on which to evaluate the loss and the accuracy at the end of each epoch.

```
def k_fold_cross_validation(data, labels, cnn=True):
    histories = []
    k_fold = KFold(n_splits=K, shuffle=True)

    model = define_cnn() if cnn else define_ff()

    for train, test in k_fold.split(data, labels):
        data_train = data[train]
        data_test = data[test]
        labels_train = labels[train]
        labels_test = labels[test]

        history = model.fit(
            data_train,
            labels_train,
            epochs=EPOCHS,
            batch_size=BATCH_SIZE,
            verbose=2,
            validation_data=(data_test, labels_test),
        )
        histories.append(history)

    return model, histories
```

3.4 Model validation

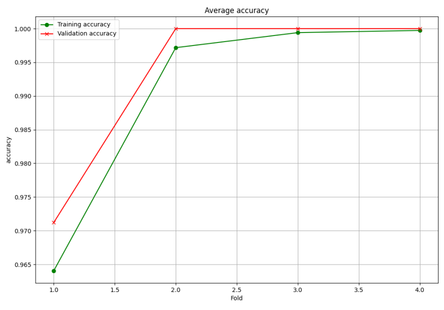
Once the training phase is completed, the model is validated through model.evaluate that computes the loss function and accuracy of the model.

```
def validate(model, data, labels):
    loss, accuracy = model.evaluate(data, labels)
    print(f"Test Loss: {loss}")
    print(f"Test Accuracy: {accuracy}")
```

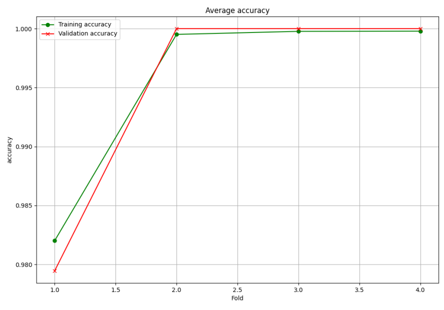
4 Conclusions

4.1 Accuracy plot

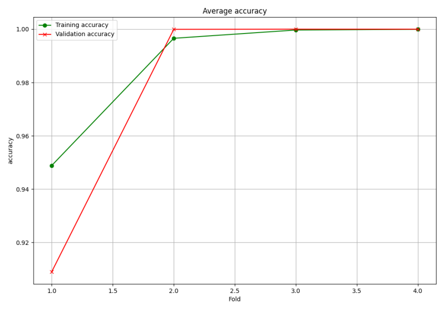
For both the convolutional neural network and the feed forward neural network, through the *matplotlib* Python package, the accuracy plots were drawn.
In general, the training accuracies for the models with higher batch sizes start with lower accuracy levels - see Figure (g) and (h). Nevertheless, both tend to converge at the second fold. As a matter of fact, all of them seems to converge at the second fold, without any sign of overfitting or underfitting. Maintaining 40 as the number of epochs, the increase in the batch size does not produce a substantial improvement in the accuracy performance of the model. 32, 64, 128 and 256 are all valid and legit choices for the model: one can choose one of them depending on the available resources. In the scenario where this model will be embedded into a mobile phone application, a model that requires less memory usage might be more valuable, meaning a model with smaller sizes. The only visible difference between plots with different batch size is that the accuracy tends, with smaller sizes, to start at the first fold with higher accuracy values, while plots with bigger sizes have smaller initial values.
However, it should be noted that Figure (a), (b), (c) and (e) show the convolutional model with worse convergence for the accuracy: it is slower than the others, but it's a small delay, that can be overlooked.



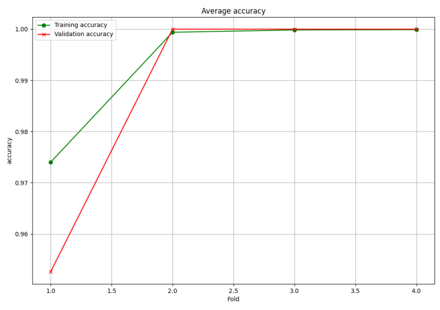
(a) CNN with 20 epochs and batch size 32



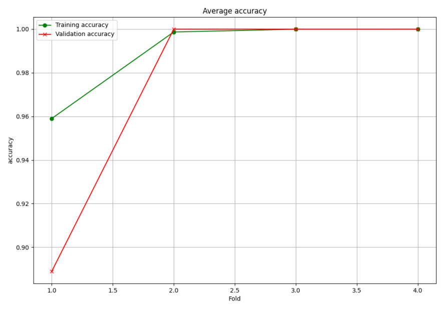
(b) CNN with 40 epochs and batch size 32



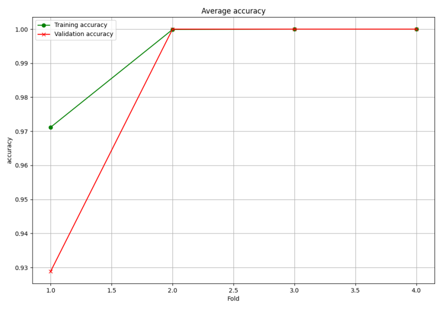
(c) CNN with 20 epochs and batch size 64



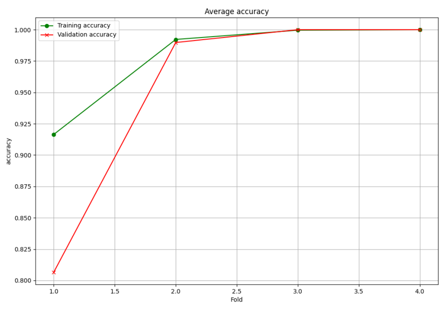
(d) CNN with 40 epochs and batch size 64



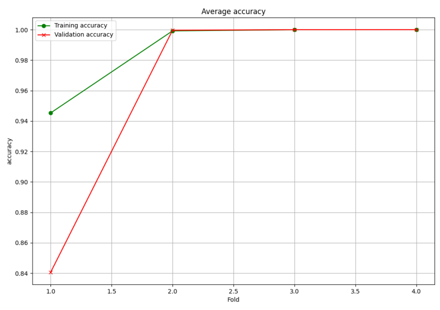
(e) CNN with 20 epochs and batch size 128



(f) CNN with 40 epochs and batch size 128



(g) CNN with 20 epochs and batch size 256

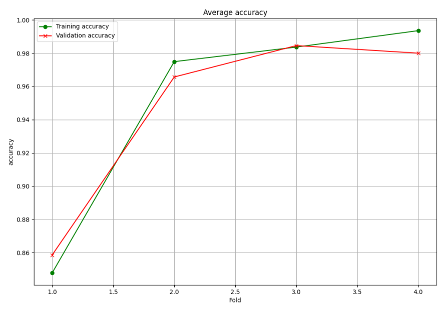


(h) CNN with 40 epochs and batch size 256

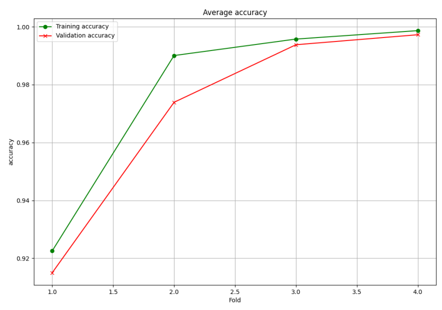
Figure 1: Accuracy Plots for different CNN configurations

Regarding the plots with 20 as number of epochs, the average training accuracy plot is - especially in the first folds - lower than the average validation plot. This is a symptom of underfitting.

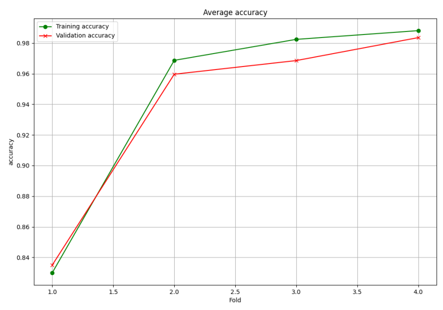
In the next page, the accuracy plots for the feed forward neural network are reported. As expected, the feed forward neural network, just by looking at these plots, has lower accuracy values. They all start from lower initial accuracy values, but in the end, they converge with a relatively good speed. The only exception, here, is the Figure (a) where the validation accuracy tends to decrease in the last fold: this be attributed to an overfitting.
It should be noted how the training accuracy and the validation accuracy have similar shape, but the red one is shifted down on lower accuracy level. Once again, this is a clear sign of overfitting, and it is evident especially in the plots with batch size 64.
A peculiar behaviour of these accuracy plots, that can be seen in Figure (a), (c) and (f), can be seen at fold number 3: all the validation accuracies seem to have a small fluctuation. This can be due to an overfitting on the third fold, if the splitting of the dataset was not optimal.



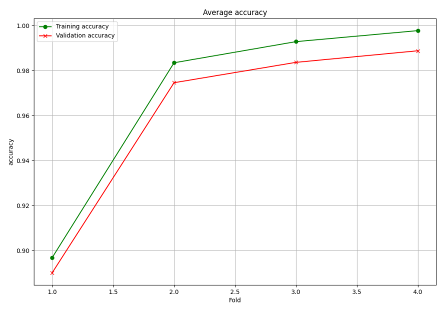
(a) FFNN with 20 epochs and batch size 32



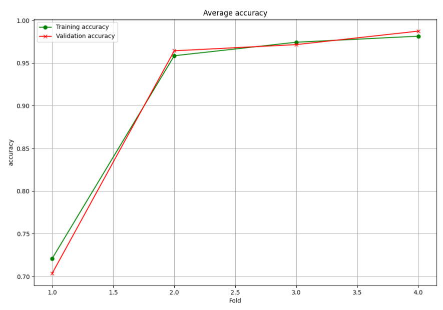
(b) FFNN with 40 epochs and batch size 32



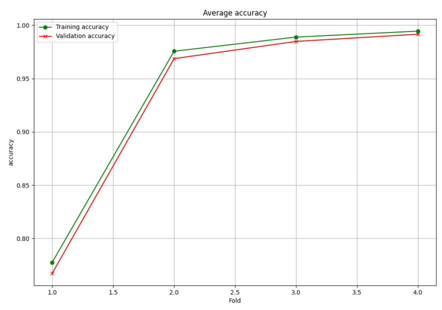
(c) FFNN with 20 epochs and batch size 64



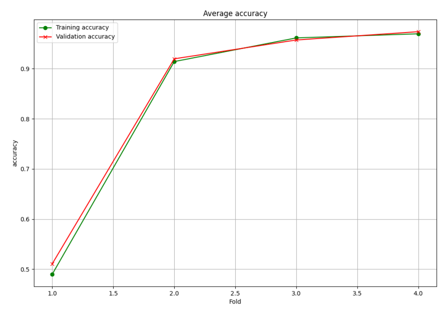
(d) FFNN with 40 epochs and batch size 64



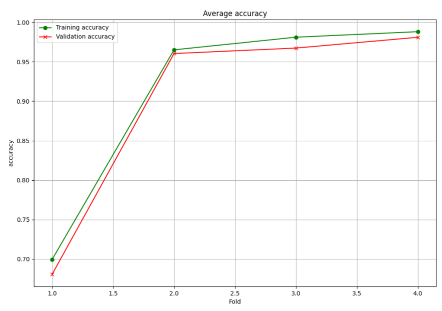
(e) FFNN with 20 epochs and batch size 128



(f) FFNN with 40 epochs and batch size 128



(g) FFNN with 20 epochs and batch size 256



(h) FFNN with 40 epochs and batch size 256

Figure 2: Accuracy Plots for Different FFNN Configurations

4.2 Training results

To choose the best combination of epochs and batch size, multiple tries were necessary. Here, two summarized tables – one per model – of the accuracy and loss values for each combination of parameters I have chosen, that are the same ones as the previous paragraph.

Table 1: Accuracy and Loss values for CNN

Accuracy	**Loss**	**n. of epochs**	**batch sizes**
0.9989	0.0057	20	32
0.9989	0.0033	20	64
0.9979	0.0049	20	128
0.9989	0.0029	20	256
0.9989	0.0162	40	32
0.9968	0.0142	40	64
0.9979	0.0052	40	128
1.0000	2.3243e-04	40	256

These are the results for the convolutional model. It should be noted that in general it is performing outstandingly in all scenarios, but, in terms of higher value for the accuracy, and consequently lower loss value, the “best” choice is the last one, with 40 as number of epochs and 256 as batch size.

Table 2: Accuracy and Loss values for FFNN

Accuracy	**Loss**	**n. of epochs**	**batch sizes**
0.9609	0.1127	20	32
0.9620	0.1171	20	64
0.9514	0.1174	20	128
0.9556	0.1324	20	256
0.9588	0.1520	40	32
0.9620	0.1177	40	64
0.9599	0.1002	40	128
0.9652	0.1002	40	256

These are the feed forward network’s results. In this case, the model is also performing well, even better that what I was expecting, and once again the combination of number of epochs and batch size that seems to work best is the one with 40 epochs and 256 as batch size.

4.3 Model’ comparison and considerations

As expected, the convolutional neural network works better with respect to the feed forward model. In their best-case scenario, the average accuracy is 99% against “only” 96%. The CNN seems to capture well the complex patterns contained in the dataset; while the feed forward may not be as effective in recognizing all the nuances inside the spatial data it was given.

Since the beginning of the developing process of the neural network the goal was using it for classifying real-time accelerometer and gyroscope values. That’s why the convolutional network was chosen, given their ability to distinguish skilfully spatial data. It served the purpose – given the remarkable results obtained. However, it is a complex model, and it requires a lot of computational

resources and quite some training time. It served my purpose – given the remarkable results I have obtained. However, it is a complex model and it requires a lot of computational resources and quite some training time.
In this scenario where a mobile phone was used, a much simpler model may be more suitable. If the resources are limited, the feed forward neural network can be a valid replacement for the convolutional one, at the expense of some accuracy. A feed forward network can be trained faster, and it can be less computationally intense.

In conclusion, these neural networks are sufficiently good in classify the data and the choice to which one to use can be made considering environmental conditions, outside the accuracy and loss they present. Choosing the feed forward networks can be also a "greener" choice: as a matter of facts the Green AI approach is becoming more relevant every day. The idea behind it is to consider in addition to the classical performance metrics, other metrics - such as, for example, the amount of electricity usage and carbon emissions, but also the elapsed real time – in the choice of a model and especially its architecture and number of hyperparameters. Following these parameters, a complex network such as the convolutional one developed for this project might be too consuming. However, in situations where high accuracy is needed a more complex architecture could be necessary.

5 References

5.1 Python's Libraries

Multiple Python libraries were been used in the implementation of this project, such as:

- *TensorFlow*: tensorflow (`https://www.tensorflow.org/`)
- *Keras*: keras (`https://keras.io/api/`)
- *scikit-learn*: scikit −learn (`https://scikit-learn.org/`)
- pandas: pandas (`https://pandas.pydata.org/`)
- numpy: numpy (`https://numpy.org/`)

5.2 Other references

- Yuksel, Asim Sinan; Atmaca, Şerafettin (2020), *"Driving Behavior Dataset"*, Mendeley Data, V2, doi
- Paul-Stefan Popescu, Cojocaru Ion. (2022). *"Driving Behavior"*, Kaggle, doi