Artificial Neural Networks and Deep Learning

Classification of time series using RNN

Authors:
La Greca Michele Carlo (10864460)
Giuffrida Andrea (10643540)
Nicolis Nicholas (10867841)

Professors:
Prof. M. Matteucci
Prof. G. Boracchi

December 20, 2022

Contents

1 Introduction

This project is part of the exam of *Artificial Neural Networks and Deep Learning* course at Politecnico di Milano, and it corresponds to the second challenge. The aim of the project is to classify samples in the multivariate time series format. In other words, since this is a classification problem, the objective is to correctly map the information contained in the features calculated over time to their labels.

Recurrent neural networks (RNN) [23] are a type of neural network that are particularly well-suited for processing sequential data, such as time series data, and there are several reasons why using RNNs for time series classification can be beneficial.

RNN are able to capture temporal dependencies, and the network can learn to model the dependencies between past, present, and future data points in the series, even going back for a long time in the past. RNN are flexible in handling variable-length sequences, as well as noise and missing data, since they can use information from previous time steps to help fill in missing values or smooth out noise.

The project will describe the data exploration and preprocessing steps in order to prepare and understand the data, and the implementation of the classification using several approaches: *RNN, LSTM* [21], *Bidirectional LSTM* [15], *1DConv* [13], *Feedforward Neural Networks* [19], *GRU* [20], *LSTM-1DConv*. Tensorflow [14] and Keras [1] will be used in Python for the implementation. Moreover, references to the notebooks will be inserted in order to check the details directly in the code.

2 Data

The dataset consists of 2429 time series, each having one of the 12 possible labels (integer form 0 to 11). Each time series is composed of 36 time-steps, each having 6 features. The data consists on a *Numpy* [12] 3dim array with the time series, and another array with a label for each time series. In Figure 1 it is possible to see the structure of a random time series, where each of the 6 plot refers to a feature, and the x axes of each plot represents the time-step *(Data_ explore)*.

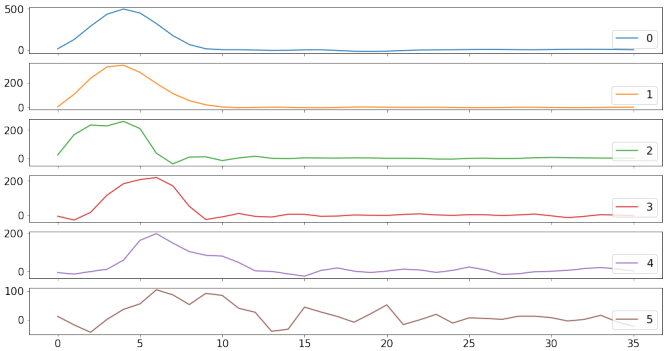

Figure 1: Example of time series

Looking at the distribution of the labels, it is possible to note an high unbalance between them.

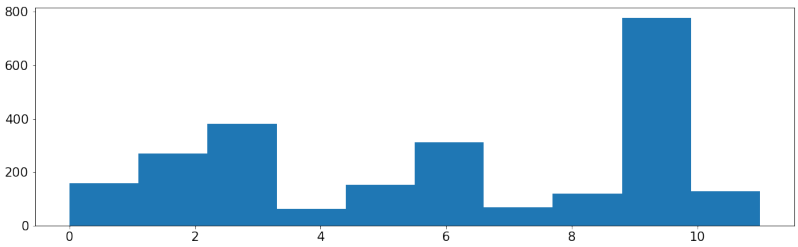

Figure 2: Distribution of the labels

3 Preprocessing

3.1 Splitting

The splitting of the initial dataset has been performed using *train_ test_ split* [11] library, obtaining 4 sets: a training set with 80% of the total data, the correspondent set of targets, a validation set with 20% of the total data and the correspondent set of targets. The test set was not created and splitted because the model will be tested on a hidden and private test set.

3.2 Normalization

Initially, no normalization has been used, keeping the time sequences with the original values. After some experiements, the data has been scaled using different approaches:

- **Standard scaling** [10]: is a popular scaling method that standardizes the data to have a mean of 0 and a standard deviation of 1. It scales the input data to a normal distribution. It can help to prevent different features in the input data from having an outsized impact on the model's performance due to differing scales;
- **MinMax scaling** [8]: it scales the data so that the minimum and maximum values of each feature are mapped to a predetermined range, usually between 0 and 1. This method is useful when it is present a dataset with many outliers or data points that are highly variable;
- **Robust scaling** [9]: is similar to the standard scaling, but it is more robust to outliers in the data. It scales the data to have a median of 0 and a range of values that is more resistant to the influence of outliers than the standard scaling;

In the very first experiments, the normalization has been performed before the splitting, and this is theoretically wrong. Infact, if the data is normalized before splitting it into training and testing sets, information from the test set can leak into the training set, which can cause the model to overfit on the training data and give overly optimistic performance results. This is because normalization typically involves using information from the entire dataset, including the testing data, to calculate the scaling parameters.

In short, after the very first experiment where the normalization has been performed before splitting, the normalization has been performed after the splitting: the scalers has been applied to the training data, adn then the validation and the hidden test sets has been scaled with the parameters of the training data. This ensures that the model is evaluated on truly unseen data, and the evaluation metrics provide an accurate measure of the model's performance on new, unseen data.

3.3 Balancing

In order to solve the unbalance of the classes, two approaches have been used:

- **Class weights**: for each class, the frequency of the time series belonging to the class has been calculated, and the weight of each class accounts the frequency of it inside the training set. It

is possible to see that the classes are highly unbalanced. The class weights will be used in the training procedure. The labels that have a small frequency of units in the data will have higher weights, so that during the training the data belonging to these classes are influencing the same even if they have lower frequency.

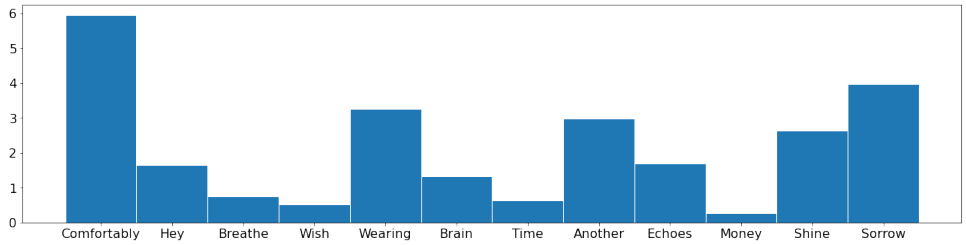

Figure 3: Distribution of the class weights

- **Oversampling and undersampling**: the library *RandomOverSampler* [7] has been used. It over-sample the minority class(es) by picking samples at random with replacement. The bootstrap can be generated in a smoothed manner.

  Then, *SMOTE* [4] library has been used. It works by generating synthetic samples from the minority class to create a more balanced dataset. This is done by randomly selecting an instance from the minority class and finding its k nearest neighbors. Synthetic instances are then created by interpolating between the minority instance and its k nearest neighbors. This process continues until the desired balance between the minority and majority classes is achieved.

  Another technique similar to SMOTE called *ADASYN* [3] has been used. ADASYN has an additional adaptive component that makes it more effective than SMOTE in some situations. The main difference between ADASYN and SMOTE is that ADASYN generates more synthetic samples for instances that are harder to learn, while generating fewer synthetic samples for instances that are easier to learn. This is done by computing a density distribution function that estimates the distribution of the minority class instances, and then generating synthetic instances in the regions where the density is lower. This adaptivity can help to ensure that the synthetic data generated is more representative of the underlying data distribution.

  Moreover, *SMOTEENN* [5] and *SMOTETomek* [6] have been used, combining both undersampling and oversampling.

3.4 Data Augmentation

Data augmentation [18] is a technique used to increase the diversity of the training set by applying random (but realistic) transformations. In this project, oversampling techniques introduced before may be seen as augmentation techniques in order to increase the available data for the training. Moreover, data augmentation has also been performed by creating duplicates of the data with some noise inside generated form a Gaussian distribution.

4 Models

4.1 RNN

RNNs are a type of neural network designed for sequential data, such as time series. They use a recurrent connection that allows information to persist over time and be passed from one time step to the next. RNNs can be used for classification of time series data by processing the data in a sequence and generating a prediction at each time step.

4.2 LSTM

LSTMs are a type of RNN that are designed to address the vanishing gradient problem, which can occur when training traditional RNNs. LSTMs have an internal memory cell that can selectively store or erase information, allowing them to remember important features over longer periods of time. They are often used for time series classification tasks where long-term dependencies are important.

4.3 Bidirectional LSTM

A bidirectional LSTM is a type of LSTM that processes the input sequence in both forward and backward directions. This allows the network to capture information from both past and future time steps, making it useful for tasks that require a deep understanding of the sequence.

4.4 1DConv

A Convolutional Neural Network (CNN) for image classification typically consists of a series of layers that transform the input image into a probability distribution over a set of target classes. The layers typically used in a CNN for image classification are: *Input Layer, Convolutional Layer, Activation Layer, Pooling Layer, Fully Connected Layer, Batch Norm Layer, Dropout layer.* These layers are usually stacked together to form a deep neural network, with the last layer producing the final output of the network.

1DConv is a type of neural network that uses convolutional layers to extract features from the time series. Convolutional layers are designed to detect local patterns in the data and can be used to capture important features such as peaks or spikes in the time series. 1DConv networks are often used for time series classification tasks where the shape of the signal is important.

4.5 Feedforward Neural Network

Feedforward neural networks are traditional neural networks that consist of a series of layers where each layer is connected to the next. They can be used for time series classification tasks by processing the entire sequence at once and generating a prediction at the end.

4.6 GRU

GRUs are a type of RNN that are similar to LSTMs but with fewer parameters. They use gating mechanisms to selectively update and reset the internal memory state, allowing them to remember important information over longer periods of time. GRUs are often used for time series classification tasks where both long and short-term dependencies are important.

4.7 LSTM-1DConv

A combination of LSTM and 1DConv networks can be used for time series classification tasks where both long-term dependencies and local patterns are important. The 1DConv network can be used to extract important features from the time series, which can then be fed into an LSTM network to capture long-term dependencies. This approach has been shown to be effective in a variety of time series classification tasks.

5 Training

Below, the design choices and the training pipeline of each networks will be described, together with the names of the python notebooks where it is possible to check the implementation details.

5.1 RNN

The RNN is the first network implemented. First, some experiements without the data normalization has been performed *(RNN_ noOverSam_ noNorm)*. To begin we first used a recurrent layer, having initialized first a MinimalCell object with 128 units. In the first model we didn't have any fully convolutional part for the classification or any dropout layer. We managed to reach slightly more than 30% accuracy. Adding a classifier helped a bit bringing the validation accuracy to almost 40%. We then set the patience at 50 epochs, to see if the model was able to overfit. We were able to get the network to overfit by using a dense layer of 512 neurons, but the validation accuracy was still at 40%. Then we increased the neurons in the recurrent layer, added 2 dropout, changed the number of neurons of the classifier and even standardized the data *(RNN_ noOverSam_ Norm)*, but we never went over the validation accuracy of 40%.

5.2 LSTM

Initially we used the original unbalanced data without any augmentation or oversampling/undersampling. Moreover, the data was standard scaled before the splitting in train and validation sets. We used a simple network with two LSTM layers and a dense layer, each with 128 neurons and RELU. We got a model that was overfitting the training, with 55% of accuracy on the validation. Increasing the number of neurons led to an increase only on the training accuracy, thus the model overfitted more. We regularized with a dropout of 0.5 rate after the LSTMs and after the dense, reducing the training accuracy and increasing a bit the validation one. With two LSTMs and a dense, each with 256 neurons, and the dropout we reached 80% on the training and 58% on the validation. In general, keeping only a dropout layer, specifically after the LSTMs, was performing better than an additional dropout before the output layer. The performances remained on that level adding neurons on the layers and adding a regularization term. As a result, we experimented that LSTM perform better when we have 512 on LSTM layers and dense, and dropout and regularization L1L2 to avoid overfitting.

5.3 Bidirectional LSTM

The bidirectional LSTMs increased the performances on the validation set, since by definition they are based on evaluating the memory of the network from both directions, bringing more information for the training phase. A network with two BiLSTM having 256 neurons and a dropout each with 0.4 rate, and a dense with 256 neurons led to a training accuracy of 86% and validation accuracy of 60%. In general BiLSTM has similar performances with respect to LSTM, but it has on average 2% more on the validation set, and it overfit more since the model becomes more complex due to the new information introduced by the BiLSTM layer *(bidir_ LSTM)*.

5.4 1DConv

We noticed that the scaling before splitting was wrong in theory, and we scaled after the splitting. We used two blocks of 128 neurons 1DConv layers and a max pooling, followed by a GAP and a 128 dense. We had 75% on training and 65% on validation. Increasing the convolutional blocks and the neurons led to overfitting (around 85% on training) and to an increase of the validation accuracy (67%). Dropout after convolutional blocks increased to 68% the validation accuracy.

We used blocks of two 256 neurons 1DConv layers followed by a max pooling. With 4 of these blocks, a dropout of rate 0.2, a 256 neurons 1DConv with a max pooling, a GAP and a 256 neurons dense we reached 70% on the validation, overfitting the training with 98% accuracy. We had 70.4% on the test accuracy *(conv_ v10)*. Adding a dropout after the dense kept the training accuracy at 93%.

Using a weighted loss function, using weights based on the frequencies of the samples belonging to the different classes didn't increase the performances, even with a less complex model.

We decided to oversample, generating new samples by randomly sampling with replacement the current available samples to have all the classes with the same number of samples. With a network with two convolutional blocks, each with two 256 neurons 1DConv layers and a max pooling, a GAP and a dense with 256 neurons, led to 73% *(conv_ v15)* on the validation accuracy, and looking at the confusion matrix, no class had 0 predicted samples. We then used a network with 4 convolutional blocks as the one before, but after two blocks we added a dropout with rate 0.4, another one before the GAP, and another one before the output layer.

We also regularized with L1L2 (e-4, e-3), reaching 74% on the validation and 70% on the test *(conv_ v17)*. Doing PCA [22] on the features, increasing the neurons or using the Leaky RELU didn't help.

We decided to use transformers [2]: a Multihead Attention layer that takes as input the first input layer (the original image) and the layer resulting from all the convolutions. In this way we combine the original sequence and the sequence after the convolutions, resulting in a richer representation. We also used 18 heads, that parallelly process different parts of the sequences. We reached 75% of validation and 72,4% on test *(conv_ v21)*, that represent our best result in this project.

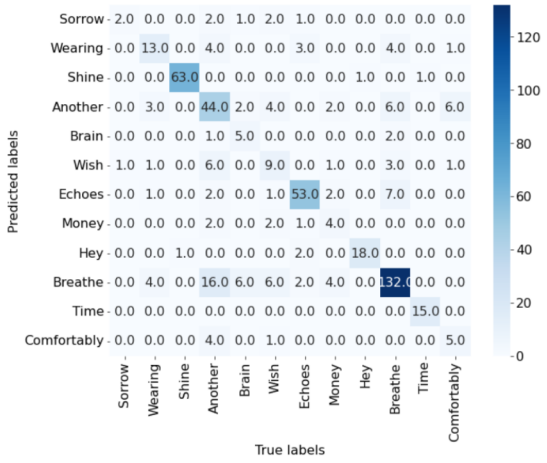

Figure 4: Confusion Matrix of the best 1DConv model so far

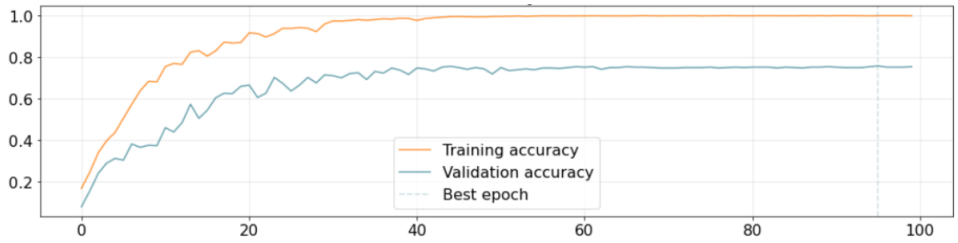

Figure 5: Accuracy of the best 1DConv model so far

We restarted with 1DConv, using 4 convolutions blocks, (each made of 3 convolution layers of 256 filters of size 3x3) with their relative pooling layer and a GAP, and we were able to reach 66% accuracy in validation, without any kind of normalization or oversampling. Then we used the same model with standardized data *(Conv1D_noOverSam_Norm)*: over the third dimension (69%), second dimension (68%) and first dimension (48%). We also tried normalizing between 0 and 1 over the third dimension, but this model gave an accuracy of 37%, so we decided to continue with the standard normalization over the third dimension.

We tried to implement some kind of oversampling *(Conv1D_OverSam_Norm)*. We started with a naive RandomOverSampler and the same model as before with 5x5 filters. We managed to reach a 68% accuracy. Then, since it seemed that the problem doesn't require so many convolutional layers, we decided to reduce them to 2 blocks of convolution, each of 3 layers and 256 filters of 3x3, and got very similar results. We also tried to do class weighting but the performance dropped to 65% accuracy.

Then we decided to try different kind of Oversamplers: Smote, Adasyn, Adasyn with dropout and Adasyn with class weighting. For all of them there were no significant changes, except that with Adasyn we were able to get more balanced f1 scores over the classes and that the class weighting had 60% in validation accuracy.

We also tried to combine UnderSampling and Oversampling with the SMOOTEENN class (50% of validation accuracy) and SMOTETomek class (69% of validation accuracy) and SMOTETomek with dropout (64% of validation accuracy)

We tried also to put a LSTM layer, also hierarchical and bidirectional, of 128 and 256 units, but in this case we didn't gain anything on performance, obtaining a validation accuracy of around 65% for these kind of models.

We tried several different configuration: modifying the number of convolutional blocks and the number of convolutional layers in the block, adding some skip connection *(Conv1D_with_ResNet)*, trying to mimic the ResNet mode), with and without Batch Normalization layer, with and without Oversampling, with different type of normalization, with dropout layers and with some LSTM layers at the beginning and at the end of the model.

At the end we managed to reach 72% validation accuracy with a model with 2 convolutional blocks of 2 layers with skip connections, no oversampling, standardized data and no LSTM, BatchNormalization or dropout layers.

We implemented a model that tries to do data augmentation creating a window that randomly sample a window of the time series *(Conv1D_WindowOversampling_Norm)*.

We also implemented bagging [16] *(Conv1D_Bagging)*, through creating a class of model, data interpolation, through the *interp1d* command, and adding gaussian noise, but there was no significant improvement in the performances *(Conv1D_Interpolation_and_Gaussian_noise)*. We also tried to

make boosting [17] *(Conv1D_Boosting)*, creating a subclass from the class *KerasClassifier*, which wraps a keras model into a scikit-learn one, in order to use the *AdaBoost* classifier, but both gave very bad results, around 40% accuracy. We also tried bagging with the previous best model which gave 70% accuracy *(Conv1D_transformer_bagging)*.

5.5 Feedforward Neural Network

The first one was a really simple FNN composed of just 2 dense layers (Total parameters: 9,932) and a gap one to reshape the output in order to fit the 12 classification classes. The accuracy on the validation set was around 44%. Increasing the number of dense layers at 6 and neurons for each layer (Total parameters: 4,922,060) we reached up to 65%. Further increase of neurons and layers didn't bring an accuracy increase *(FeedForward)*.

5.6 GRU

The model we used here was based on a previous version of a 1DConv, but we decided to attach a GRU layer in the end of it, as substitute of the GAP layer. We placed here this layer because of its input shape and output shape and with the idea of adding some sort of memory to the network. In the end the accuracy on validation was 69%. This mean no gain and neither loss in performances *(GRU)*.

5.7 LSTM-1DConv

the last model is the combination of LSTM and 1DConv, called ConvLSTM1D. We reshaped the input data in order to fit the input shape of the ConvLSTM1D layer, and we performed several tests. First with 1 layer of ConvLSTM1D, then with 2, 3 and 4 setting the *return_sequences=True*. We noticed that every layer we added caused the accuracy to decrease. Thus, in the end we keept the simplest between the models with just 1 layer of ConvLSTM1D, a 1DConv and the usual GAP layer + dense for final classification (total parameters: 3,712,780). The Accuracy reached on validation and test set was 70%. Once we implemented the interpolation of input data, we used it also here, but every test performed badly. We also tried to add Gaussian Noise layer to the network in order to see if could have been a nice theoretical alternative to the dropout, since it create a little of randomness into the internal layer *(lstm1DConv)*.

6 Results

The RNN model achieved a relatively low validation accuracy of 40%. This is probably due to the fact that RNN works well when the length of the time series is not high. Infact, RNN suffer of the vanishing gradient when the steps of the series are more than 15 on average.

The LSTM model achieved a higher validation accuracy of 58%, indicating that it was able to capture some of the temporal patterns in your data. Infact, it works well with long time series, due to the structure of the LSTM made of gates. The BidirLSTM model achieved a higher validation accuracy of 60%, indicating that bidirectional processing helped to capture more complex temporal patterns in the data.

The 1DConv model achieved the highest validation accuracy of 75%, indicating that it was able to effectively capture important features in the time series data.

The GRU model, built on top of the 1DConv, achieved a validation accuracy of 69%, indicating that it was able to capture some temporal patterns, but not as well as the 1DConv

LSTM+1DConv achieved a validation accuracy of 70%, indicating that the combination of these two types of networks can be effective for time series classification.

FNN achieved a relatively low validation accuracy of 44%, indicating that it may not be well-suited for time series classification problems.

7 Conclusions

Overall, the 1DConv model achieved the best results in terms of validation accuracy, followed by BidirLSTM, LSTM+1DConv, and GRU. This suggests that convolutional neural networks and bidirectional processing can be effective for capturing temporal patterns in time series data, and that combining different types of neural networks can be useful for improving classification accuracy.

Moreover, these results taught us that our models, even if theoretically very different from each other, couldn't overcome the 75% accuracy with the inputs we were using. That's another reason why we tried to manipulate input data to extract "more information" from them, using for example oversampling, interpolation and bagging.

References

[1] Keras. Keras. https://keras.io/.

[2] Keras. Transformers. https://keras.io/examples/timeseries/timeseries_transformer_classification/.

[3] Imbalance learn. Adasyn. https://imbalanced-learn.org/dev/references/generated/imblearn.over_sampling.ADASYN.html.

[4] Imbalance learn. Smote. https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html.

[5] Imbalance learn. Smoteenn. https://imbalanced-learn.org/stable/references/generated/imblearn.combine.SMOTEENN.html.

[6] Imbalance learn. Smotetomek. https://imbalanced-learn.org/stable/references/generated/imblearn.combine.SMOTETomek.html.

[7] Inbalanced learn. Randomoversampler. https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.RandomOverSampler.html.

[8] Scikit learn. Minmaxscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[9] Scikit learn. Robustscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html.

[10] Scikit learn. Standardscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[11] Scikit learn. train_test_split. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[12] Numpy. Numpy. https://numpy.org/.

[13] Pytorch. Conv1d. https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html.

[14] Tensorflow. Tensorflow. https://www.tensorflow.org/.

[15] Yugesh Verma. Complete guide to bidirectional lstm (with python codes). https://analyticsindiamag.com/complete-guide-to-bidirectional-lstm-with-python-codes/.

[16] Wikipedia. Bagging. https://it.wikipedia.org/wiki/Bagging.

[17] Wikipedia. Boosting. https://en.wikipedia.org/wiki/Boosting_(machine_learning).

[18] Wikipedia. Data augmentation. https://en.wikipedia.org/wiki/Data_augmentation.

[19] Wikipedia. Feedforward neural network. https://en.wikipedia.org/wiki/Feedforward_neural_network.

[20] Wikipedia. Gated recurrent unit. https://en.wikipedia.org/wiki/Gated_recurrent_unit.

[21] Wikipedia. Long-short term memory. https://en.wikipedia.org/wiki/Long_short-term_memory.

[22] Wikipedia. Pca. https://en.wikipedia.org/wiki/Principal_component_analysis.

[23] Wikipedia. Rnn. https://en.wikipedia.org/wiki/Recurrent_neural_network.