ML Project A.A. 2019-2020

Andrea Graziani (0273395)
andrea.graziani93@outlook.it
Università degli Studi di Roma "Tor Vergata"
Rome, Italy

KEYWORDS

Non-Intrusive Load Monitoring, Time-Series, Walk-Forward (Sliding Window Approach), Hyper-parameters optimization, RNN, LSTM, Early Stopping, Over-Fitting

1 INTRODUCTION

Non-Intrusive Load Monitoring (**NILM**) is the task of estimating the power consumption of each appliance given aggregate consumption.

Formally, given the *aggregate power consumption* $(x_1, \ldots, x_T)$ from N active appliances, our objective is to deduce the power consumption $(y_1^i, \ldots, y_T^i)$ of appliance $i = 1, \ldots, N$, such that at time $t = 1, \ldots, T$.

The aggregate power consumption x_t at time t is given by the sum of the power consumption of all the known appliances plus a noise term ε_t, that is:

$$x_t = \sum_{i=1}^{N} y_t^i + \varepsilon_t \tag{1}$$

2 DATA-SET

According to project specification, our data-set includes all observations, regarding the aggregate power consumption, from *2019-01-01 00:00:00* to *2019-03-14 23:59:59*.

A very important characteristic of our data-set it that it contains **time series** data, where the time of each instance, containing the attribute value regarding power consumption, is given by a *timestamp* attribute; thus our data-set represents a sequence of *discrete-time data* [8]. All data are listed in time order.

The *sampling frequency*, i.e. the number of sampling points per unit time, is equal to 1 Hz.

3 PRE-PROCESSING

In this section, we will describe how we have **pre-processed** our data before to use our neural network.

Our pre-processing activity is made up of following three step:

- *Missing data points management*
- *Standardization*
- *Feature extraction*

3.1 Missing data points management

To check the existing of missing values inside aforementioned range, we have built a function called `manage_missing_time_series_records`, which detected exactly 224692 missing data points.

First of all, to manage missing data, we have filled all the "*holes*" inside our data-set with `NaN` values.

Finally, all `NaN` values was filled via *linear interpolation*, exploiting `pandas.Series.interpolate`[1] function.

3.2 Standardization

Since the range of values of raw data varies widely, we have applied a **feature scaling** method, according to which all attributes values must to lie in a fixed range [8].

Scientific literature reports that feature scaling techniques are able to accelerate deep network training [4][8].

To be more precise, we used a technique called **standardization** according to which, after calculating the statistical mean μ and standard deviation σ of attribute values, we subtracted the mean from each value and divide the result by the standard deviation [8]. Here is the formula for standardization:

$$X' = \frac{X - \mu}{\sigma} \tag{2}$$

From an implementation point of view, standardization task was performed using `RawDataPreprocessor` python class.

To be more precise, we have tried another approach, called *normalization*, which is applied by dividing all values by the maximum value encountered or by subtracting the minimum value and dividing by the range between the maximum and minimum values. However, our trials showed that normalization reduce our neural network performance on test set, therefore this method was been discarded

3.3 Feature extraction

Finally, inspired by Schirmer and Mporas [7]'s report, we have performed a **feature extraction** activity according to which we built some derived, informative and non-redundant values from our raw data in order to facilitate the subsequent learning and generalization steps.

Precisely, for every 240[2] instances we have calculate 3 statistical features, that is *mean* values, *minimum* and *maximum* values, adding them to our dataset. This task was performed by `features_extraction` function.

Our trials showed that adding this activity has improved slightly our neural network performance on test set.

4 THE VALIDATION TECHNIQUE

When working with deep learning, it is vital to have separate *training*, *test*, and *validation* sets. How to identify them?

In order to take into account the time-sensitiveness of our data, we need a **validation technique**, that is a *technique which defines a specific way to split available data in train, validation and test*

[1] pandas.Series.interpolate
[2] This value was been heuristically chosen according to our neural network performance on test set.

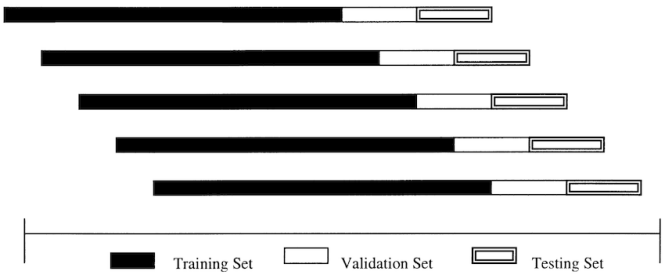

Figure 1: Walk-forward's scheme used to divide each whole data set into several overlapping training–validation-test sets.

sets, capable **to preserve the temporal order of data**, preventing for example that the testing set contains data antecedent to the training set. These techniques are generally called *time-series* and **must** be used when the data is time-sensitive [2].

Therefore, since traditional methods of validation and cross-validation are unusable in our context, during our project development, we have adopted a widely used time-series technique called **Walk-Forward** with **Sliding Window Approach**.

Using this approach, is possible to train our model excluding both any future data including those which are too far in the past too [2].

To be more precise, we have adopted the walk-forward scheme proposed by both Cao and Tay [1] and Kaastra and Boyd [5] according to which we have divided whole-data set into overlapping training–validation–testing sets in order to the testing set follows validation set which, in turn, follows training set, respecting temporal order of data. This scheme is shown in figure 1.

Since, according to project specification, all observations, belonging to *final* test set regarding the temporal range from *2019-03-15 00:00:00* to *2019-03-31 23:59:59*, taking into account the sampling frequency of 1 Hz, are exactly 1468800, corresponding to exactly 17 days of data, according to our scheme each training–validation–testing is made up of as follows:

- 1468800 instances for test set (17 days).
- 1468800 instances for validation set (17 days).
- 2937600 instances for training set (34 days).

In other words, each training–validation–testing sets include a total of 5875200 (68 days) instances corresponding to exactly 64 days of data.

Finally, we have decided that each training–validation–testing set is moved forward through the time series by 950400 instances (11 days); in this way, we are able to include the final test set during the last iteration of walk-forward scheme.

The main disadvantage of that scheme are the frequent **retraining** steps which we have to perform as new data becomes available. Despite the scheme is more *time consuming*, it allow us to adapt more quickly to the change of power consumption over the time[1][5].

From an implementation point of view, aforementioned validation technique is performed using `SlidingWindowWalkForward` class.

5 NEURAL NETWORK ARCHITECTURE

In order to resolve the NILM problem, we have exploited so-called **recurrent neural networks** (**RNN**).

We have adopted this kind of network because time series data, such as our data-set, containing aggregate power consumptions, exhibit a dependence on past data, called the *secular trend*[3]. In other words, it is said that time series data tend to exhibit a significant *autocorrelation*, which means that a data point value at time $t + 1$ are quite likely close to data point value at time t.

RNN networks incorporate this dependence by having a *hidden state*, or *memory*, which makes them prime candidates for tackling learning problems involving sequences of data, such as time series data[3][8].

5.1 A brief digression about RNN

Formally, the value of the hidden state at any point in time is a function of the value of the hidden state at the previous time step and the value of the input at the current time step, that is:

$$h_t = \Phi(h_{t-1}, x_t) \tag{3}$$

where h_t and h_{t-1} are the values of the hidden states at the time steps t and $t-1$ respectively, and x_t is the value of the input at time t [3].

To be precise, to compute the internal state h_t, using an activation functions $act()$, we have to compute the sum of the product of the weight matrix W and the hidden state h_{t-1} at time $t-1$ and the product of the weight matrix U and the input x_t at time t, that is [3][8]:

$$h_t = act(Wh_{t-1} + Ux_t) \tag{4}$$

Generally, $tanh$ is the most used activation function because its second derivative decaying very slowly to zero, which help to combat so-called *vanishing gradient problem*, that is the problem according to which, for several reasons occurring during back-propagation, the gradient can very easily either diminish to zero or explode to infinity [3][8][6].

The output vector y_t at time t is the product of the weight matrix V and the hidden state h_t, using an activation function, generally a *softmax*[3][8]:

$$y_t = act(Vh_t) \tag{5}$$

Finally just like traditional neural networks, training the **RNN** also involves *back-propagation*. The difference in this case is that since the parameters are shared by all time steps, the gradient at each output depends not only on the current time step, but also on the previous ones. This process is called **back-propagation through time** (**BPTT**) [6][3].

5.2 Our implementation

Fortunately `Keras` provides the `SimpleRNN` recurrent layer that incorporates all the logic we have seen so far.

Despite the `SimpleRNN` layer performs well, we have decided to adopt so-called **LSTM** (**Long short term memory**) neural network, a variant of RNN capable of learning long term dependencies

and specifically created to address the vanishing gradient problem [3][8][6].

The exact configuration of our regression neural network is as follows:

- **Conv1D**, convolutional layer with F filters, kernel size K, stride 1, and ReLU activation function.
- **LSTM**, Long short term memory with H units.
- **Dense**, that is a fully connected layer with D units and ReLU activation function.
- **Dense**, that is a fully connected layer with $D/2$ units and ReLU activation function.
- **Dense**, that is a fully connected layer with $D/4$ units and ReLU activation function.
- **Dense**, that is a fully connected layer with 1 units and linear activation function.

In principal, the convolutional layer was not expected, however our trials show that the addition of a convolution layer slightly increases performance on test set.

6 HYPER-PARAMETERS OPTIMIZATION

As known, best performance on a test set is achieved by adjusting the value of several **hyper-parameters** to suit the characteristics of data.

First of all, we remember that, during hyper-parameters optimization, it is very important *not to use performance on the test data to choose best values for hyper-parameters*. This is because peeking at the test data to make choices automatically introduces optimistic bias in the performance score obtained from this same data. Performance on future new data will very likely be worse than the estimate [8].

In our project, hyper-parameters optimization regards following elements:

- The number of filters (F).
- The size of each kernel (K).
- The number of neurons of *Dense* layer (D).
- The number of neurons in the recurrent layer (H)

To do hyper-parameters optimization we perform the hyper-parameters optimization scheme proposed by Witten et al. [8].

According to their scheme, we have trained our neural network model several times with different hyper-parameter values using, clearly, the training set, and each of the resulting neural network models is evaluated using the validation set.

Once the hyper-parameters values, that give best performance on the validation set (minimum value of validation loss), have been determined, a *final neural network model is built with that hyper-parameters values* and trained with*both* aforementioned training and validation sets.

As stated by Witten et al. [8], omitting data belonging to validation set from final training can reduce performance in the test but, at the same time, this choice can be risky due to over-fitting. The Witten et al. [8]'s solution is to stop training after the same number of epochs that led to the best validation set performance.

Only after the final model is fixed are we allowed to use the test data to obtain an estimate of the performance of this model on new, unseen data. Basically, the test data can only be used once, to establish the final *Energy Based F1* performance score.

From implementation point of view, hyper-parameters optimization is performed using `start_hyperparameters_selection_phase` function.

6.1 Early stopping

Early stopping is employed as a form of regularization to avoid over-fitting since it stops the training when the error on the validation set starts to grow.

7 HOW TO RUN

To establish the final *Energy Based F1* performance score on test set regarding the temporal range from *2019-03-15 00:00:00* to *2019-03-31 23:59:59*, is necessary to execute following steps:

- Open `Nilm.ipynb` file using *Google Colab*[3]
- Execute all code inside "*Imports, 'Google Drive' mounting and GPU checking*", "*Project's functions...*" and "*Project's classes...*" sections of the notebook.
  Make sure to mount *Google Drive* on `/content/drive` directory.
- Following files **must** exist on *Google Drive*:
  - `main_train.csv`
  - `fridge_train.csv`
  - `dishwasher_train.csv`
- Following files, regarding test sets, **must** exist on *Google Drive*:
  - `main_test.csv`
  - `fridge_test.csv`
  - `dishwasher_test.csv`
- Load into current session's main directory following files:
  - `dishwasher_neural_network.h5`
  - `fridge_neural_network.h5`
- Run all code inside "*Script for final testing...*". The final *Energy Based F1* performance score will be displayed automatically.

REFERENCES

[1] L.J. Cao and Francis Tay. 2003. Support vector machine with adaptive parameters in financial time series forecasting. *Neural Networks, IEEE Transactions on* 14 (12 2003), 1506 – 1518. https://doi.org/10.1109/TNN.2003.820556

[2] Davide Falessi, Likhita Narayana, Jennifer Thai, and Burak Turhan. 2018. Preserving Order of Data When Validating Defect Prediction Models.

[3] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras.* Packt Publishing.

[4] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 448–456. http://proceedings.mlr.press/v37/ioffe15.html

[5] Iebeling Kaastra and Milton Boyd. 1996. Designing a neural network for forecasting financial and economic time series. *Neurocomputing* 10, 3 (1996), 215 – 236. https://doi.org/10.1016/0925-2312(95)00039-9 Financial Applications, Part II.

[6] Jack Kelly and William Knottenbelt. 2015. Neural NILM: Deep Neural Networks Applied to Energy Disaggregation. https://doi.org/10.1145/2821650.2821672

[7] Pascal Schirmer and Iosif Mporas. 2019. Statistical and Electrical Features Evaluation for Electrical Appliances Energy Disaggregation. *Sustainability* 11 (06 2019), 3222. https://doi.org/10.3390/su11113222

[8] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques* (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[3] colab.research.google.com