Financial Engineering
S.S. 2019 - 2020

Group 7B
Final project NAX

Course supervisor: Prof. Baviera Roberto

17.06.2020

Brambilla Letizia
Hartmann Konrad
Maggioni Fabio

POLITECNICO
MILANO 1863

Abstract

The goal of the present project is to develop a model for middle-term horizon power consumption prediction. First we build a simple GLM model, that represents trend and seasonality of the phenomenon. Then we improve this model considering the shallow Neural network with autoregressive feature presented in Azzone & Baviera (2020) [1]. It takes as input seasonal and weather variables and outputs the parameters of the residuals' distribution, assumed to be Gaussian. This network has been implemented in Python, Keras, with Tensorflow back-end. Some implementation details are discussed, with a focus on stability techniques and sensitivity analysis, which are carried out to let the network achieve reliable and efficient results. More complex versions of the network with multiple layers were also built. Moreover, we implemented a modified version of the network, exploiting the Matlab Deep Learning Toolbox.
We applied the model to the GEFCom2017 dataset, containing data about power consumption in New England U.S.A. We consider a three year time-window training set, and validate the results on the fourth year. We value the goodness of the model through some evaluation techniques, such as Pinball Loss function and Backtested Confidence Intervals.

Contents

1 Data mining and data description

In this project, the GEFCom2017 dataset on New England households' power consumption was studied.
It includes data of the whole New England area and eight different New England zones. It contains households' hourly consumption in MWh, together with averaged wet bulb and dry bulb hourly temperatures in Fahrenheit degrees.
We consider the data for the whole New England area. We select nine calendar years, from January 2008 up to December 2016, and we remove the 29^{th} of February to preserve the seasonality structure in leap years.
Moreover we aggregate data to get daily consumption and average temperatures for every day.
Descriptive statistics about daily power consumption and weather data in New England for the 2008-2016 time window are shown in Table 1.

	Min	Max	Mean	Median	Standard Deviation
Consumption [GWh]	246.38	543.01	345.60	338.91	42.49
Dry bulb [°F]	-1.52	88.04	50.12	51.35	17.72
Wet bulb [°F]	-19.75	72.46	38.48	39.64	18.99

Tab. 1: Descriptive statistics

Cumulated power consumption in New England between January 2008 and December 2010 is shown in Figure 1 (red line). Sundays' demands are marked with blue circles.

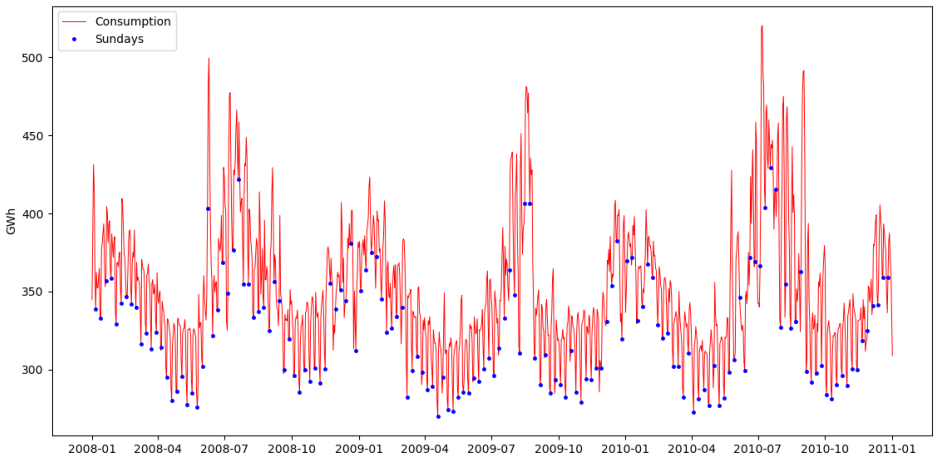

Fig. 1: Cumulated power consumption

We observe a yearly seasonal behaviour with two peaks per year, in winter and summer. We notice that demand in summer is higher than in winter, and can be even double w.r.t. demand in spring and autumn. We also observe local minima corresponding to power consumption on Sundays. Finally it can be seen that volatility is higher during summer than in the rest of the year.

2 GLM model

The first implemented model is a Generalized Linear Model. As explained in the paper, we consider eight regressors:

- the time, measured in days starting from 0 on the first date in the dataset (1^{st} March 2003) and then standardize such that it takes values between 0 and 1
- four goniometric functions with angular frequency ω and 2ω, which represent yearly and semi-annual behaviours; they consider the time measured in days
- three dummy variables taking values 1 respectively on Saturday, Sunday and on holidays.

Using this regression, we want to describe the seasonal and weekly behaviours, aside from a general trend.
We get the following intercept and regression coefficients:

		Estimate	(SE)
Intercept	β_0	0.495***	(0.020)
Trend	β_1	-0.041	(0.044)
sin (ωt)	β_2	-0.010*	(0.004)
cos (ωt)	β_3	-0.028***	(0.004)
sin $(2\omega t)$	β_4	-0.131***	(0.004)
cos $(2\omega t)$	β_5	0.033***	(0.004)
Saturday	β_6	-0.120***	(0.008)
Sunday	β_7	-0.143***	(0.008)
Holiday	β_8	-0.123***	(0.016)

Tab. 2: GLM parameters calibrated on the training dataset 2008-2010 with their Standard Error (SE). With *** we indicate statistical significance of the parameter at 0.1% significance level and with * statistical significance of the parameter at 5% significance level.

Notice that, although the variable *Trend* is not significant on this training set, we decided to keep it in the model because it shows to be significant when the GLM was fitted on following years.
Figure 2 shows realized power consumption (blue line) on the test set 2011, together with prediction of a GLM model trained on 2008-2010 time-window.

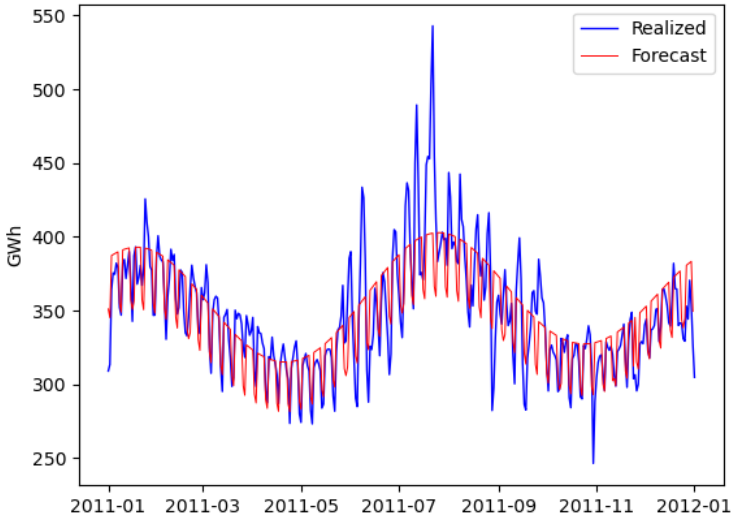

Fig. 2: GLM power consumption prediction

3 NAX model

3.1 Model selection

The GLM model explains $R^2 = 63.7\%$ of the variability of the energy demand; this means that it is meaningful and describes well the seasonal behaviour, although there is still high variability in the residuals, which we want to explain. Such residuals seem to be normally distributed, as shown in figure 3, but they show different variance over time: especially in summer the residuals dispersion shows high variance.

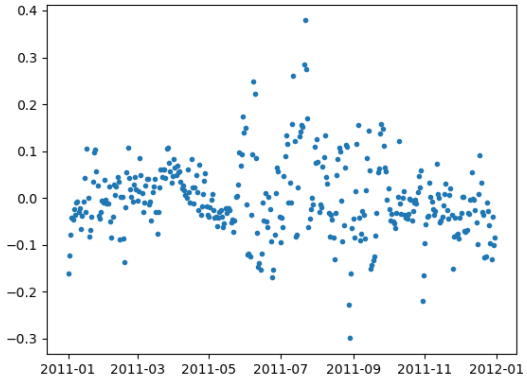

Fig. 3: GLM residuals

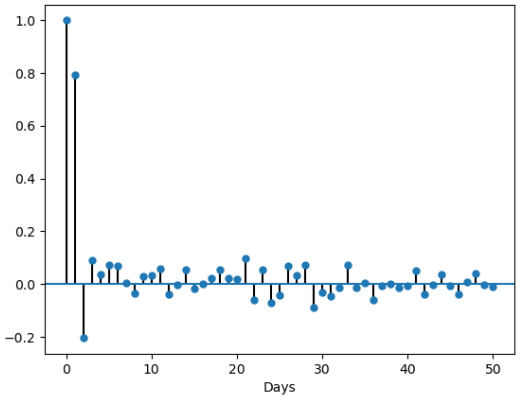

Fig. 4: Partial autocorrelation

In the attempt to fit and predict the behaviour of such residuals and taking into account the partial autocorrelation at one time step, as shown in Figure 4, we implemented the Neural network presented in Azzone & Baviera (2020) [1]. Such network is a Neural Autoregressive eXogeneous network (NAX). We aim at a probabilistic forecast of the residuals of the GLM. Based on Baviera & Messuti (2019) [2], we assume a Gaussian distribution for each residual R_t, and we want the network to estimate the parameters μ_t and σ_t of such distribution. The network is composed by three layers:

- input layer: it takes exogenous inputs X_t, namely calendar and weather conditions at time t, plus the autoregressive term, which is the output P_{t-1} of the network from last iteration, at time $t-1$
- hidden layer: a layer with N neurons and a non-linear activation function $\mathcal{H}$
- output layer: a layer with two neurons, as the output P_t is constituted by a two-elements vector representing the mean and the standard deviation of the supposed Gaussian distribution.

NAX can be represented by the following formula:

$$P_t = l\mathcal{H}(wX_t + fP_{t-1} + w_0) + l_0$$

Figure 5 shows an example of this scheme, with three hidden neurons and a softmax activation function for the hidden layer.

The NAX was implemented in Python using a Sequential model. We built the hidden layer using a SimpleRNN layer, from the Python package Keras, which gives the recurrent part of the Neural network. The output layer was implemented via a Dense layer, constituted by two neurons.

In this network we are not giving as input the output $P_{t-1} = [\mu_{t-1},\ \sigma_{t-1}]$ of the past iteration, instead we are giving as autoregressive input the outcome of the N neurons of the hidden layer. This is not a problem and does not change the physical interpretation of the autoregressive inputs given to the network, as the output layer has a linear activation function, i.e. the obtained μ and σ are just a linear combination of the N values given by the hidden layer, plus a bias (work done by l and l_0). In

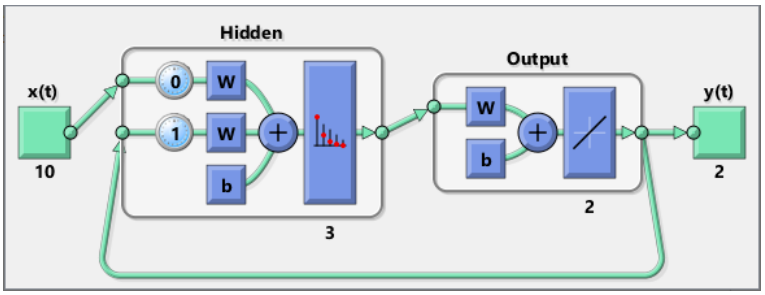

Fig. 5: NAX Network

the presented network, the output of the last iteration P_{t-1}, when considered as input, is multiplied by another weights' matrix f, before being fed to the N hidden neurons. In our construction we are simply compounding this passages from hidden neurons to outcomes μ and σ and from there to the hidden neurons of next iteration, in just one passage, handled by the layer SimpleRNN, which uses a $N \times N$ squared matrix of weights g, to describe the contribution given by last step hidden neurons, to the ones of next iteration. If we define the output of the hidden layer at iteration t as D_t, we can represent the implemented network via the formula:

$$P_t = l\mathcal{H}(wX_t + gD_{t-1} + w_0) + l_0$$

Figure 6 shows a scheme of the model implemented in Python, again with three hidden neurons and a softmax activation function in the hidden layer.

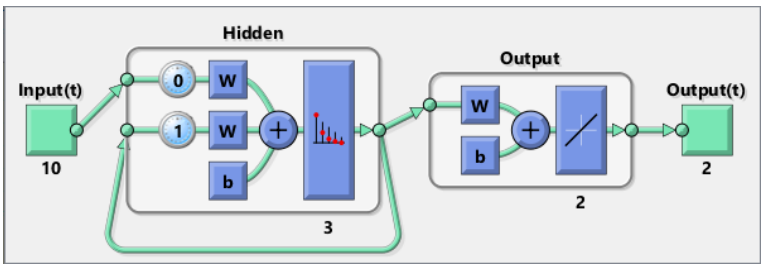

Fig. 6: NAX Network with Sequential model

As we built a Neural network with two outputs, we need a loss function which considers both parameters. The one proposed in the Azzone & Baviera (2020) [1] is the negative loglikelihood (NLL) of the assumed Gaussian distribution. This can be seen as a generalization of the classical Mean Squared Error (MSE), which assumes the variables to be i.i.d.

3.2 Convergence problems

Before choosing the right hyper-parameters we needed to tackle an issue that emerged when implementing the model, regarding the divergence of the solutions. With divergence we mean that, either the network doesn't significantly change the estimation done by GLM, estimating all μ values extremely near to zero, or the results showily take a completely wrong direction w.r.t both the demand and the GLM. In both cases the estimation of sigma generally takes huge or meaningless values.

With most seeds the model diverges and is not able to give a meaningful improvement to the RMSE. The divergence is not related to the hyper-parameters' combination choice, as the same combination converges or diverges depending on the seed. To tackle our problem, we split our solution in two: weights initialization and loss function tweaks.

3.3 Weights initialization

The stability of the network shows to be highly dependent on the initial guess on the weights. Given a good starting point, convergence is ensured. On the other hand, it is impossible to know a priori which are the right weights for a given model. A good initial guess for most Neural network is to use random weights at the start of each iteration training. We implemented uniforms, glorot uniform and Gaussian initialization functions, all random. Our network showed low stability with the said initialization, diverging in a large majority of starting seeds. To tackle this problem, we will choose two different initializations in the hyper-parameters' research and in the calibration of the network. This is due to two factors: in the calibration the shape of the matrix which need to be initialized is fixed, and we will exploit the information obtained during the hyper-parameters' research for a meaningful initialization.

3.4 Loss function

Regarding the loss function, the first problem we have is that it is an even function w.r.t. the standard deviation, so, getting negative σ is as likely as getting positive ones. This is no big issue, as it is just related to the loss function shape, so negative values for σ do not mean any error. The problem can be fixed simply asking for the absolute value of the σ values found by NAX.

Moreover, the loss function gives convergence problem when the variance gets values near to zero. We fix this problem by bounding below the values that can be taken by the variance at a given level *strike*, before feeding them to the loss function with the function `clip` from `keras.backend`. The choice of such *strike* is fundamental and complex as we need to consider an important trade-off. A low value makes the model prone to diverge, while a high *strike* can clip the values taken by the variance, giving a meaningless standard deviation σ in output. Using a model with clipped variance leads to a hybrid loss function, halfway between the Gaussian negative loglikelihood and the MSE function, as it calculates the loss using the same variance for every value of estimated variance below the *strike*. This also means that we are losing any sensitivity on the set of sigma values laying in $[0, \sqrt{strike}]$. We aim to look for a *strike* which doesn't clip possible standard deviation values, but ensures convergence when the weights are properly initialized. We choose for this purpose $strike = 0.0001$. With this value we have a high probability to diverge when weights are randomly initialized, but when we get to a good initialization for the network parameters, the calibration process always achieve stability. In our results all estimated variance values are above the bound. Should some variance take lower values, they should be all set to value *strike*, as we have no sensitivity on values below, and underestimating the variance is worse than overestimating it.

3.5 Hyper-parameters

The research of optimal hyper-parameters consisted in an optimization problem, solved by enumeration, in which we considered five different parameters, namely the number of neurons in the hidden layer, the non-linear activation function of this layer, the initial learning rate of the used optimizer ADAM, the size of the batch and the regularization parameter, a weight which penalizes high absolute values of the network weights. We considered the values presented in Table 3 for each parameter, and then proved all possible combinations, training the network on 2008-2010, and testing its performances on the base of the Root Mean Squared Error (RMSE) between predicted and realised energy demand on year 2011. It could be interesting to underline how, such a performance indicator considers only one of the two values computed by the network. Although both parameters μ and σ are taken into account by the loss function, and μ is surely the most important parameter between the two, it could be meaningful to reconsider such a decision, in favour of a performance indicator which also takes into account the precision of the estimated standard deviation. It is not unlikely to consider a combination as optimal due to low RMSE, to later find out that it has convergence problem in the estimation of σ.

Hyper-parameter	Values
Number of neurons (hidden layer)	3, 4, 5, 6
Activation function	softmax and sigmoid
Initial learning rate	0.1, 0.01, 0.003, 0.001
Batch size	50, no batch
Regularization parameter	0.001, 0.0001, 0

Tab. 3: Set of hyper-parameters

For this phase, we considered different weights initialization implemented in `keras`, to find a stable alternative to the default initialization. The one giving the most stable results, is an initialization to zero of all weights, except for the recurrent weights, which were initialized with default initialization 'orthogonal'. The method gave excellent results on μ, and so on RMSE. However, all values of σ were often estimated around zero, as the neurons corresponding to the variance never activated, i.e. the lower bound on σ brought to keep it stable around zero. To avoid this, we initialize the bias of the output layer with a bias of zero on μ, but of 0.1 on σ. In this way we make sure the neuron corresponding to variance gets activated from the start of the training.

In the figure below, we can see the comparison between the distribution of RMSE obtained by this initialization (Figure 7) versus the default one (Figure 8). Values over 30 GWh have been collapsed to the last bin. We can see a clear division between values which are around the original GLM model (approximately 25 GWh) and values which are optimal iterations for the code (around 10 GWh). We observe that many iterations do not converge.

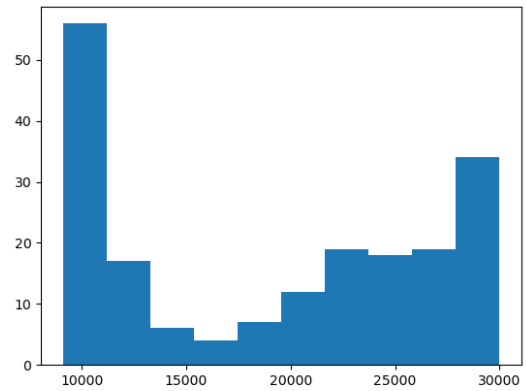

Fig. 7: Our initialization

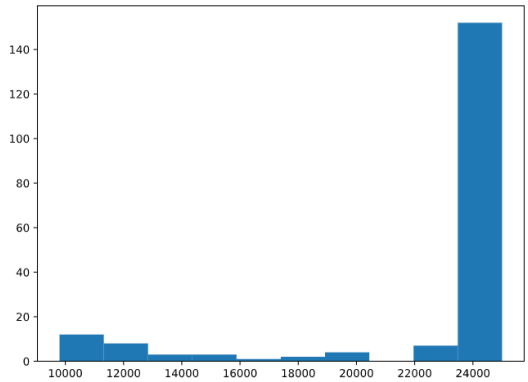

Fig. 8: Default initialization

With this initialization we obtain the following optimal hyper-parameters:

Number of neurons (hidden layer)	3
Activation function	softmax
Initial learning rate	0.003
Batch size	50
Regularization parameter	0.0001

Tab. 4: Selected hyper-parameters

with $RMSE = 9.09\ GWh$. The code took 88 minutes to run in our best performing machine.

It is important to observe that it is difficult to define an optimal combination: the optimal hyper-parameters search algorithm brings to define a set of good combinations, with RMSE around 10, as shown in Figure 7. We establish the optimal combination between these as the one with lowest RMSE,

but this highly depends on the seed. So, we do not claim to have surely chosen the best combination, but we have definitely found a good one.

In Figure 9, we can observe that we have a wide range of heterogeneous combinations which give fine results. This plot shows how, for all considered hyper-parameters, we have a quite uniform distribution between the values taken by each hyper-parameter. This underlines how important it is to conduce the research considering all possible combinations, as it is not the value of a parameter to be optimal, but the right combination. The only parameter which shows a big difference between its values is the batch size. Probably using a batch is more efficient than not using it.

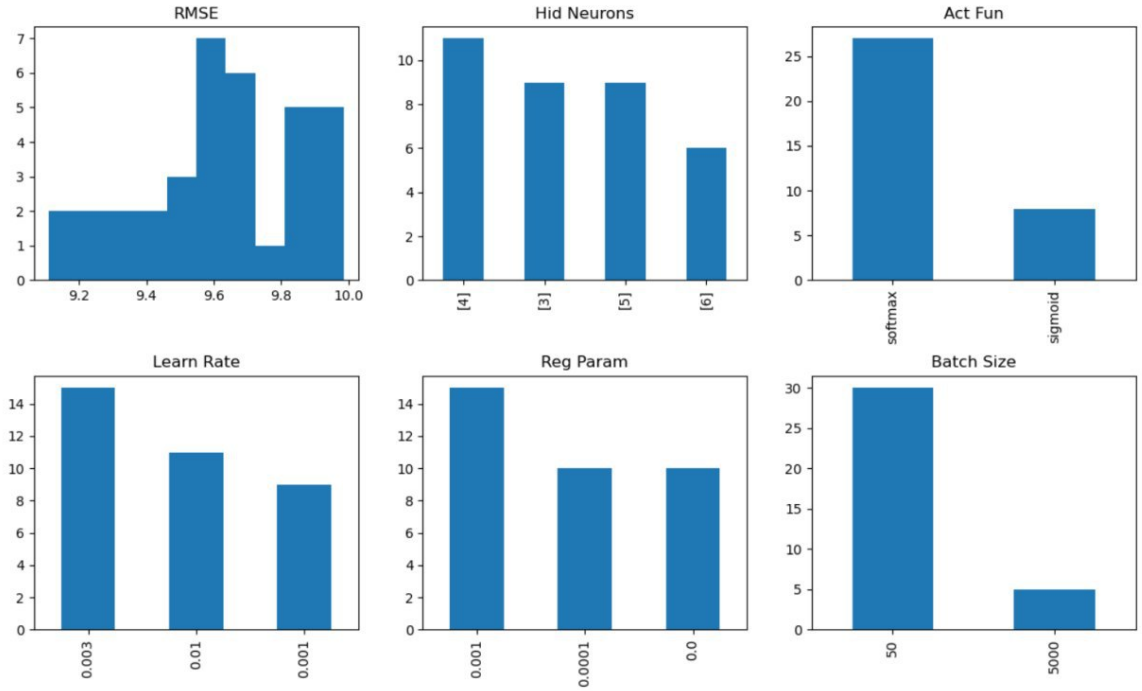

Fig. 9: Barplots

4 Model evaluation

After the choice of optimal hyper-parameters, we calibrated the model on the 2009-2011 time-window. In Figure 10 realized (blue line) and expected (red line) power consumption in GWh between January and December 2012 are shown, together with 95% Confidence Interval. We built the Confidence Intervals taking into account that the natural logarithm of power demand at each time t, after being standardized, is distributed as a Gaussian random variable, with mean $T_t + S_t + \mu_t$ and standard deviation σ_t, where T_t and S_t are respectively the trend and the seasonality terms involved in GLM model, and μ_t and σ_t were found through the Neural network. Therefore, the Confidence Intervals are built exploiting the quantiles of a log-normal distribution.

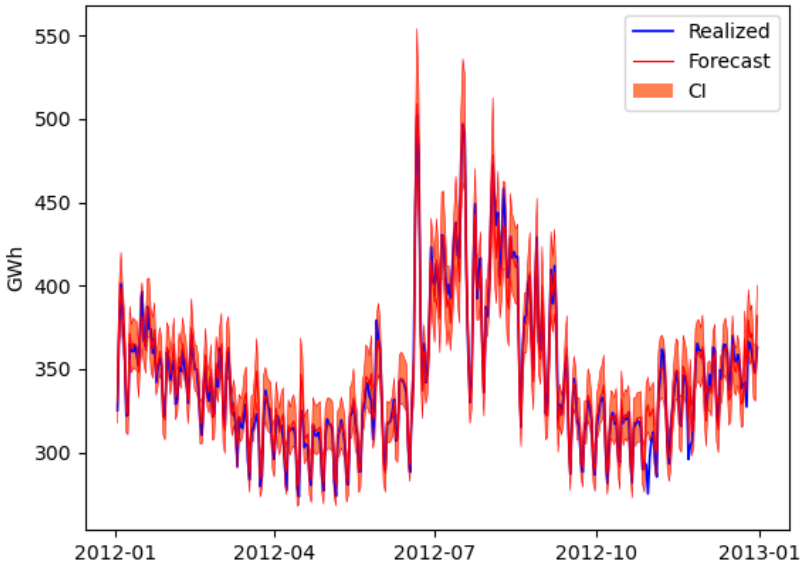

Fig. 10: NAX power consumption middle-term density prediction - test year 2012

Moreover we plot Pinball Loss and the Backtested Confidence Interval for the test set, i.e. year 2012.

Pinball Loss is an error measure for quantile forecast. For every α between 1% and 99%, the α-quantile of the predicted distribution is compared with all the realized values of power consumption: if a realized value falls above the quantile, we consider its distance from the realization multiplied by its probability α; otherwise we consider its distance from the realization multiplied by $1-\alpha$. The pinball loss function penalizes low-probability quantiles more for overestimation than for underestimation, and vice versa in the case of high-probability quantiles. Than the average across all the realization is taken, for each value of α.

Figure 11 shows Pinball Loss for the 1^{st} to the 99^{th} percentiles of GLM, ARX and NAX models. It can be seen that NAX Pinball Loss is lower than the one of GLM and ARX models for every percentile, i.e. NAX model is sharper and more accurate than the other ones. Moreover NAX Pinball Loss is symmetric, that means that density forecasting of power consumption is reproducing with the same accuracy both right and left tails of the actual consumption density. This is important in energy demand prediction because we are interested in both higher consumptions (right tail) and lower consumptions (left tail), as the latter can lead to negative electricity prices.

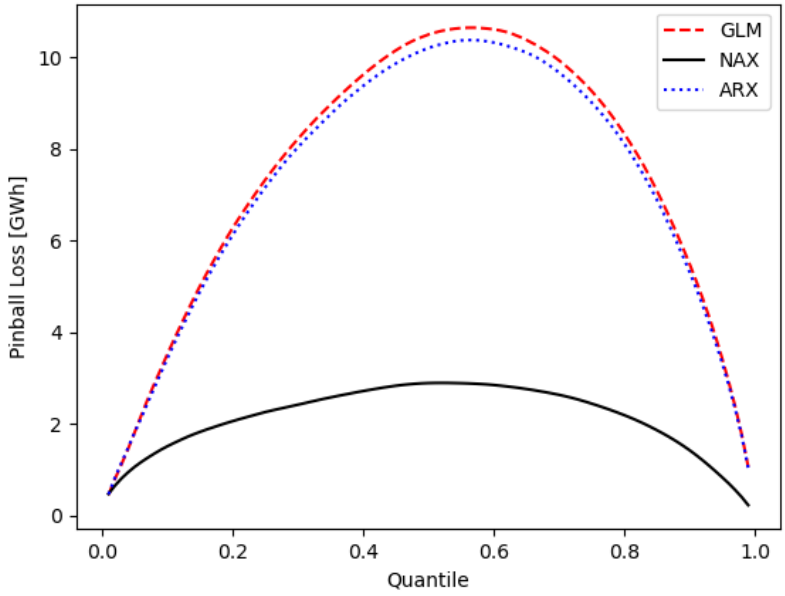

Fig. 11: Pinball Loss function for GLM, ARX and NAX models for every percentile - test year 2012

The reliability of the model is measured backtesting Confidence Intervals at α confidence level, with α from 1% to 10%. For each confidence level, we count the fraction of realizations that falls outside the corresponding CI, and compare it to the expected one, namely $1 - \alpha$.
Figure 12 shows that NAX backtested levels are close to the nominal ones, and compare them to that of simple benchmarks as GLM and ARX models.

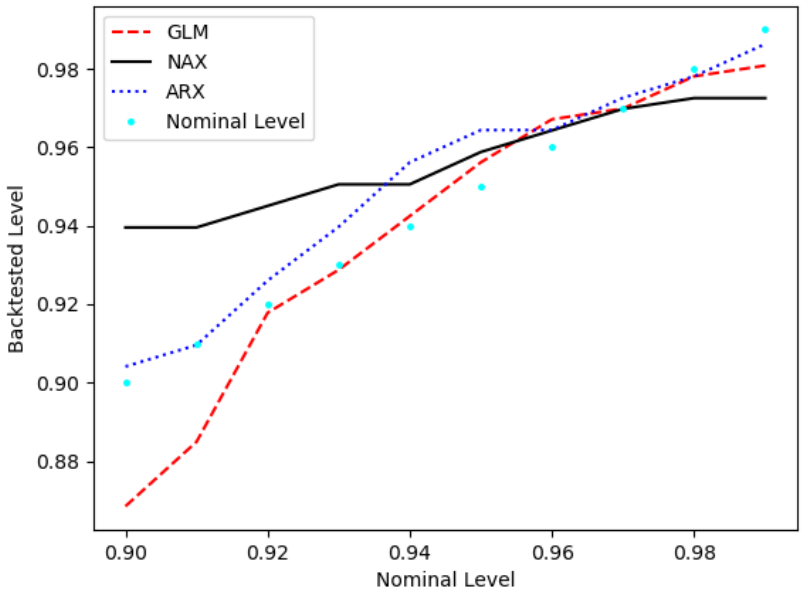

Fig. 12: Backtested CI for GLM, ARX and NAX models - test year 2012

In order to quantitatively verify that Nominal and Backtested Levels are close enough, Unconditional Coverage Test is performed. It checks the zero-hypotheses that the empirical coverage equals the nominal level. We perform also the Conditional Coverage test that checks whether there is a particular correlation within the exceptions, i.e. if the exceptions tend to cluster in particular periods of the year.

Table 5 shows the results of these tests, together with the threshold over which the null hypotheses should be rejected. It can be seen that GLM, ARX and NAX models pass the first test, but fail the second one.

	GLM	ARX	NAX	χ^2 Statistic
Unconditional Coverage	0.30	1.76	0.63	3.84
Conditional Coverage	33.26	27.35	29.99	5.99

Tab. 5: Likelihood Ratios tests at 95% CI

In order to check the robustness of NAX model, we test on the years 2013 to 2016, each time calibrating the model on the previous three years time-window.
Figure 13 and 14 show Pinball Loss function and Backtested Confidence Interval related to test year 2013, while Figure 15 and 16 are the same graphs for the test year 2014.

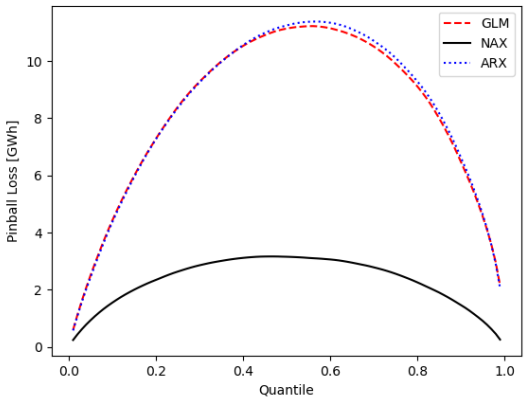

Fig. 13: Pinball Loss function - test year 2013

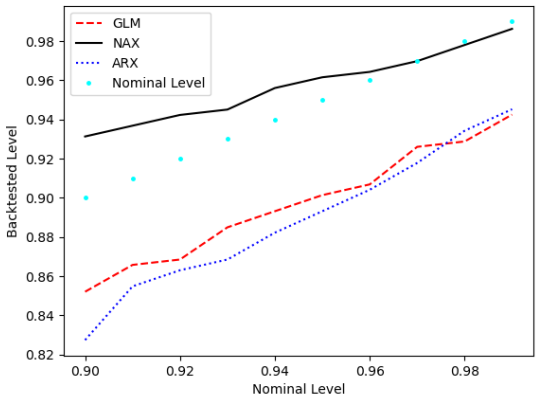

Fig. 14: Backtested CI - test year 2013

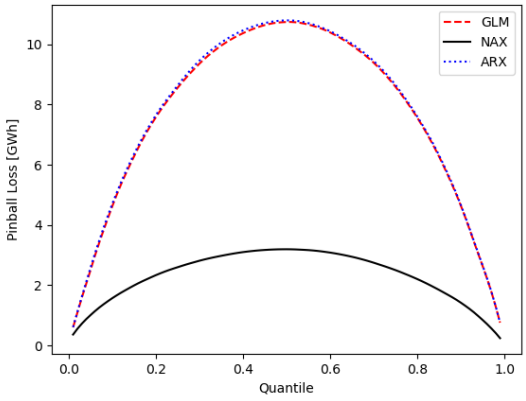

Fig. 15: Pinball Loss function - test year 2014

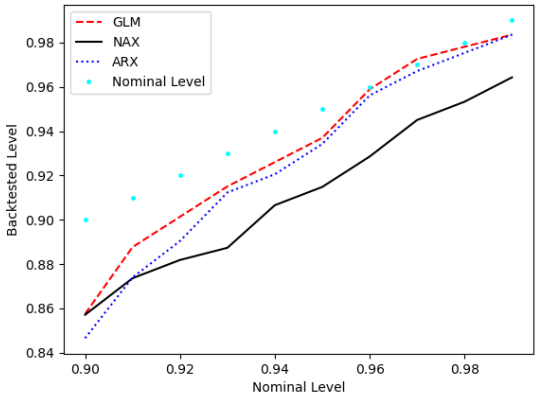

Fig. 16: Backtested CI - test year 2014

It is important to underline that σ_{NAX} is, on average, less than one half σ_{GLM} and σ_{ARX}, so the NAX Confidence Intervals turn out to be tighter than the ones obtained with the other models. This, in principle, could lead to worse Backtested Confidence Intervals plot. However, NAX model provides a more accurate prediction of the residuals' mean value, enabling us to have Backtested Confidence Interval similar to the ones obtained with the two benchmark models. Moreover a smaller standard deviation results in a sharper prediction, so leads to lower values of the Pinball Loss function.

In the end, we make a comparison between MAPE, RMSE and APL of GLM, ARX and NAX on the different test sets (from 2012 to 2016). The results are shown in Table 6: NAX has always the best indicators, in particular its MAPE falls always below the 5% threshold. This confirms both the capability of the model to make accurate point and density forecasting, and its robustness over time.

	MAPE (%)			RMSE [GWh]			APL [GWh]		
	GLM	ARX	**NAX**	GLM	ARX	**NAX**	GLM	ARX	**NAX**
2012	5.81	5.66	**1.68**	26.69	26.13	**8.21**	7.43	7.24	**2.13**
2013	6.38	6.47	**1.79**	31.03	31.02	**8.59**	8.26	8.33	**2.32**
2014	6.06	6.08	**1.85**	26.78	26.97	**8.26**	7.64	7.70	**2.31**
2015	6.44	6.27	**1.97**	28.51	28.25	**8.88**	7.99	7.85	**2.47**
2016	6.30	6.17	**1.81**	28.18	27.76	**7.76**	7.81	7.62	**2.20**

Tab. 6: MAPE, RMSE and APL for GLM, ARX and NAX on different test sets (from 2012 to 2016)

5 Ex-post forecasting technique

An ex-post forecasting technique is a method that uses the information about the prediction variables which extends beyond the time at which the actual forecast is performed, but should not assume knowledge of the data that are to be forecast. Obviously this means that ex-post forecasting are always carried out on past events. In our specific case, we are assuming to know weather and calendar variables, while we use the realized demand only to validate the results.

The ex-post forecasting technique is generally compared with ex-ante forecasting technique, which considers only the information available at the time of the actual forecast. This information will be processed in two steps to forecast the desired output: in a first step a prediction on the driver variables in the future is carried out, then such variables are fed into the model to forecast the response.

The main advantage of ex-post forecast w.r.t ex-ante is that it allows us to evaluate the goodness of the model, neglecting the error in the prediction of the driver variables. In our specific case, we already know wet bulb and dry bulb temperatures on the test set and use them to predict the distribution's parameters. In this way we become aware of the capability of the Neural network to explain the variability of the response, starting from the inputs.

On the other hand, such an assumption is obviously unrealistic if we have to predict values in the future. In our case we have no problem on the calendar variables, but forecasts on future temperatures are reliable only in few days in advance. This model is thought for middle-term predictions, i.e. from few months up to one year in advance. As such, we could think to simulate weather variables and then use this model to predict distribution's parameters, although this prediction will be affected by another error not taken into account by ex-post technique.

6 Facultative

6.1 Extended hyper-parameters grid

We repeated the model training, but this time on a bigger hyper-parameters' grid (Table 7). We added possible values for each parameter, and also modified the code to accept multiple layers. As each new parameter added lots of iterations to the number of models we had to train, we were very careful in choosing a grid which considered meaningful alternatives, but at the same time didn't overload our computers. This particular grid took 18.6 hours to complete in our best-performing machine. The number of combinations is 2810, more than 15 times the original grid.

Hyper-parameter	Values
Number of neurons (hidden layer)	2, 3, 4, 5, 6, 8, 10, [3,3], [3,3,3], [3,10], [10,3]
Activation function	softmax, sigmoid, tanh, softplus
Initial learning rate	0.0001, 0.001, 0.01, 0.1
Batch size	25, 50, 500, no batch
Regularization parameter	0, 0.0001, 0.01, 0.001

Tab. 7: Enlarged grid of hyper-parameters. Values in brackets stand for multi-layers network

To choose new parameters, especially the ones which are numbers, we did an analysis of the best values from the previous grid search. We added values that were similar to the ones in the best solutions to try to do a sensitivity analysis on the hyper-parameters selection. By best solutions we mean the solutions that gave values under 10 GWh, when considering RMSE.

The multiple layers were coded as sequential SimpleRNN layers. Each single hidden layer is recurrent in itself. Each layer takes the output of the previous iteration. As shown previously, the number of neurons does not significantly affects the results (at least between 3 and 5). To construct the new models we choose to use 3 neurons and 10 neurons layers. Figure 17 shows an example of a NAX network with two hidden layers, both with three neurons and a softmax activation function.

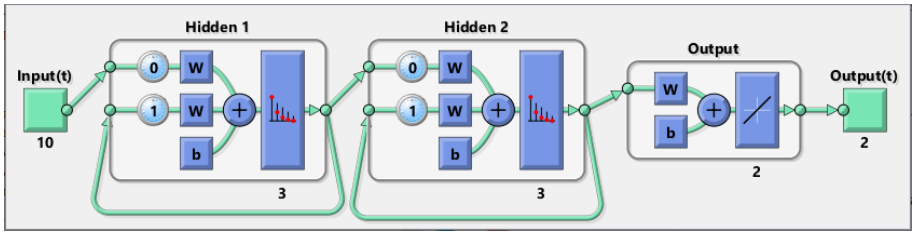

Fig. 17: Example of multi-layer NAX Network

6.1.1 Model selection

The optimal combination is

Number of neurons (hidden layer)	5
Activation function	sigmoid
Initial learning rate	0.01
Batch size	500
Regularization parameter	0.001

Tab. 8: Selected hyper-parameters

with $RMSE = 9.13\ GWh$.

In the Figure 18 we observe the barplots and histogram for the RMSE values under 10 GWh. Any model with an RMSE under 10 GWh is a pretty good model, much better than the original GLM. Furthermore, although a model may better fit this particular test year, it doesn't mean it will be the best fit for the following years. For this reason, it is interesting to analyse the neighbourhood of the optimal solution.

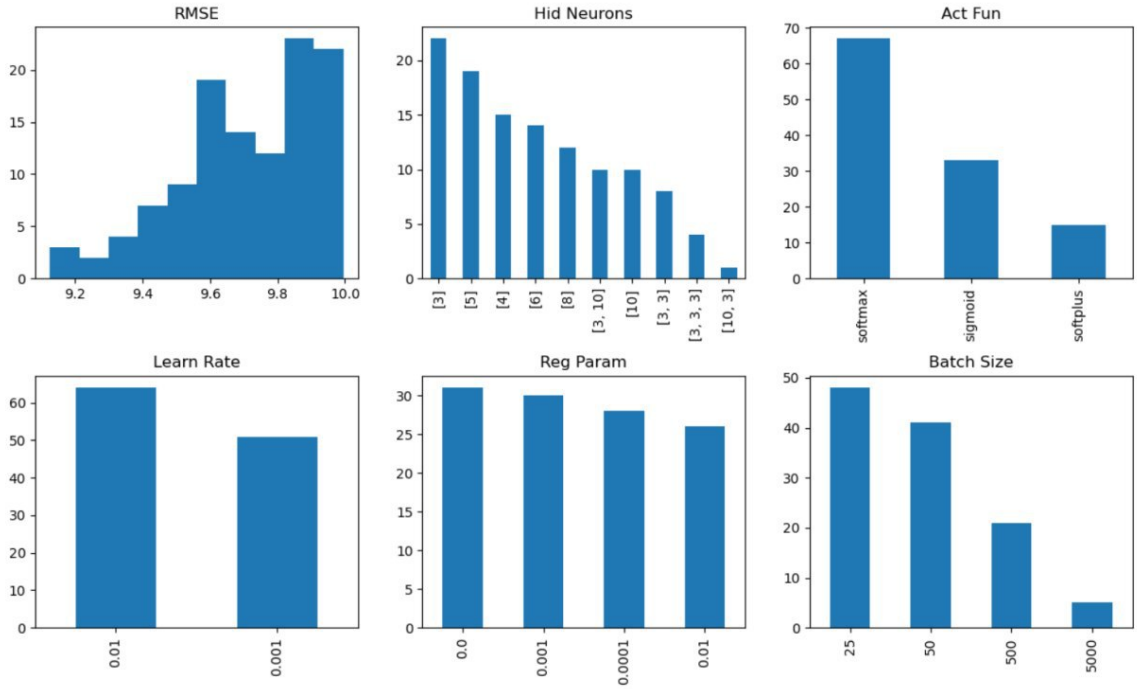

Fig. 18: Barplots

Observing the results, we can see how, each combination of neurons can return good results given a right selection of the other parameters. However, layers with high number of neurons and multi-layer networks don't improve the fit of the GLM as much as other combinations. The only exception being 2 neurons, as none single layer 2 neuron model gave results significantly better than GLM.

The original activation functions outperform the default 'tanh' used in SimpleRNN layers. The 'softplus' converges much less than the original activation functions, but nevertheless is capable of achieving sub 10 GWh RMSE results.

6.1.2 Sensitivity analysis

The parameters for the optimal solution are all inside the original set, except the batch size. Without more iterations we can't be sure of the reason behind this, but we can try to give a naive explanation. Almost every model converges to the wanted value, but the validation loss oscillates a lot, before stopping thanks to the implemented callback. A different starting seed may change how the models read the data, and thus it would change the timestep at which the validation loss will stop. With different starting seeds a completely different optimal combination may be found, but the set of "low" solutions is stable. Having said that, we can see in the different bar plots that the original parameters are the ones that consistently give results under 10 GWh.

We have to highlight that the algorithm to choose the best performing hyper-parameter combination only takes into account the RMSE. Indeed, the Backtesting plots and Likelihood Ratios for this best performing point is worse than for the original. If instead we take the best model, subject to being a single layer 3 neuron network the back testing is much better, as we will see in the next section.

6.1.3 Model evaluation

In this section we compare the optimal combination found on the previous grid, on the whole enlarged grid and on the enlarged grid, subject to being a single three neurons layer. We can see that both RMSE and MAPE are similar between the three proposed solutions. As we can see in table 9, the Likelihood Ratios are much worse for the 5 neurons optimal solution than for the other 3 neurons iterations. As we already highlighted, choosing a point in the hyper-parameters grid only due to its RMSE disregards the variance, and may lead to worse results for the estimation of the confidence interval.

	NAX_{std}	NAX_{big}	NAX_3	χ^2 Statistic
Unconditional Coverage	0.63	14.44	3.12	3.84
Conditional Coverage	29.99	40.71	14.39	5.99

Tab. 9: Likelihood Ratios tests at 95% CI

Table 10 compares the results for the different grids of hyper-parameters.

	MAPE (%)			RMSE [GWh]			APL [GWh]		
	NAX_{std}	NAX_{big}	NAX_3	NAX_{std}	NAX_{big}	NAX_3	NAX_{std}	NAX_{big}	NAX_3
2012	1.68	1.94	1.67	8.21	9.10	8.08	2.13	2.51	2.14
2013	1.79	1.80	2.05	8.59	8.22	9.60	2.32	2.25	2.77
2014	1.85	1.93	2.33	8.26	9.13	9.80	2.31	2.63	2.81
2015	1.97	2.19	2.57	8.88	10.25	11.04	2.47	2.75	3.15
2016	1.81	1.81	1.75	7.76	8.13	7.67	2.20	2.27	2.24

Tab. 10: MAPE, RMSE and APL for NAX on the standard hyper-parameters grid (NAX_{std}), on the enlarged hyper-parameters grid (NAX_{big}) and with three neurons (NAX_3) on different test sets (from 2012 to 2016)

Figures 19, 20 and 21 show a comparison between Confidence Intervals, Pinball Loss functions and Backtested Confidence Intervals obtained with the hyper-parameters resulting from the enlarged grid and from the grid with only 3 neurons, on the test year 2012. We can see the Backtesting for the five neurons network is much worse than for its three neurons counterpart.

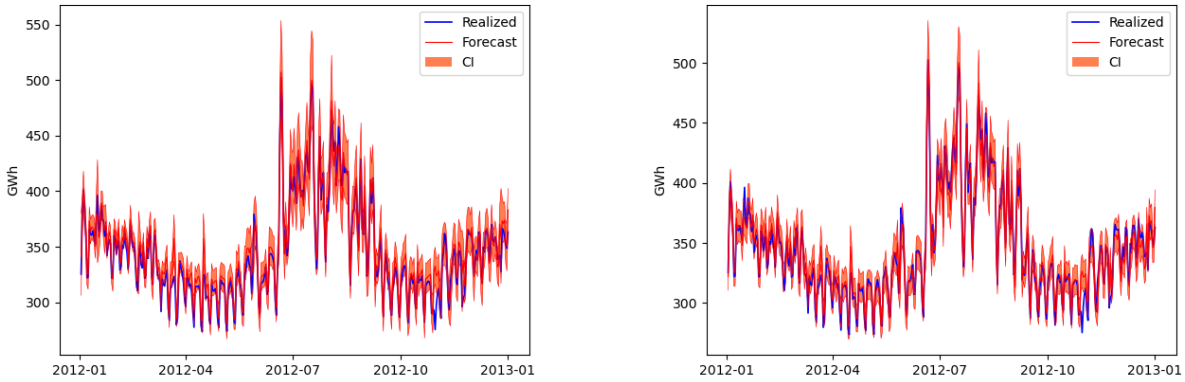

Fig. 19: Confidence Intervals obtained with the hyper-parameters resulting from the enlarged grid, and from the grid with 3 neurons

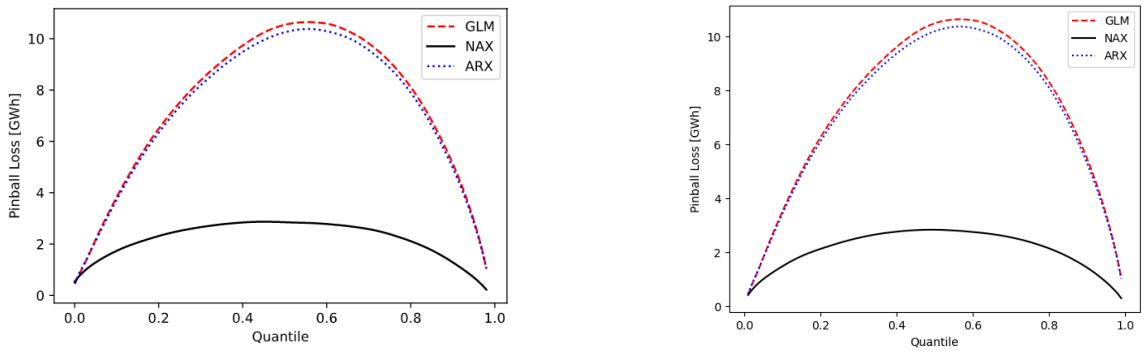

Fig. 20: Pinball Loss functions obtained with the hyper-parameters resulting from the enlarged grid, and from the grid with 3 neurons

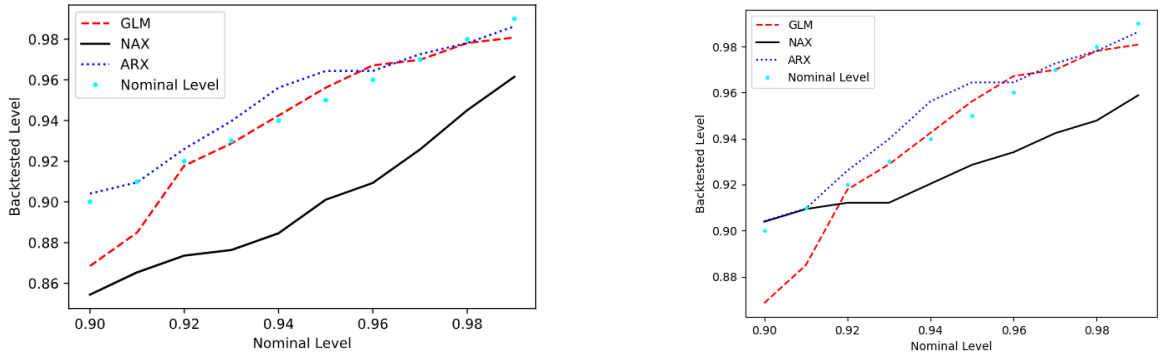

Fig. 21: Backtested Confidence Intervals obtained with the hyper-parameters resulting from the enlarged grid, and from the grid with 3 neurons

6.2 Matlab

We implemented the same model in Matlab, using the Deep Learning Toolbox. There are some differences in the building of the Neural network which we would like to highlight and explain. For the other two models, GLM and ARX, we obtained results coherent with the ones observed in Python.

Using the Toolbox we constructed (with the command `narxnet`) a Neural network that resembles the one from Python, with one main difference: the recurrent part gives, as autoregressive input to the hidden layer, μ and σ outputted from last time instant. This more closely resembles the model proposed in the paper by Azzone & Baviera (2020) [1].

The main difference between Keras and the Deep Learning Toolbox regarding our model is the loss function. In Keras it is rather easy to use a function as custom loss that behaves as planned, having as input two tensors (that can have different dimensions) and output the loss. Instead, in the Deep Learning Toolbox it is much tricky to implement such a function. Eleven different functions have to be written and packed, and they have to be imported from inside the Deep Learning Toolbox directory. We implemented a custom loss function, namely `mll`, which stands for Mean Log Likelihood function. The function can be found inside the Matlab folder, but it should be moved inside the Deep Learning Toolbox directory for it to be used.

Although the custom loss was implemented, in order to use the Toolbox full potential, a stock function must be used. If we choose to run the model with the function `mse`, the Network trains outstandingly fast compared to Python. MSE stands for Mean Squared Error, and is equivalent to the mll under the hypothesis that each output has equal variance. In addition, to completely make use of the Toolbox, the learning method was set to `trainlm`. This training method doesn't allow to change batch size.

Using MSE as loss function, gave us the opportunity to compare the results of the two methods and to further appreciate the negative loglikelihood as loss function. In this case the estimation of σ was done as in GLM and ARX models, using the most classic statistical estimator, on the train set. In this way we obtain a constant σ over all days of the train and test set.

For a more thorough comparison between Tensorflow Keras and the Deep Learning Toolbox the training method "Adam" can be implemented as training options, and batch size can be specified with it. The results for these different scenarios weren't pursued due to time constraint, and limited computing power of our machines.

In the research of optimal hyper-parameters we consider the number of neurons in the hidden layer, its activation function, the learning rate and the regularization parameter, and no batch size. It is remarkable how fast the process of finding an optimal hyper-parameters' combination runs w.r.t. the Python code, even considering that the size of the used grid is half the one used in Python. The optimal value we obtained was:

Number of neurons (hidden layer)	3
Activation function	softmax
Initial learning rate	0.1
Regularization parameter	0.0001

Tab. 11: Selected hyper-parameters

with $RMSE = 9.99\ GWh$.

Besides the smaller hyper-parameters grid and the differences related to the use of MSE we consider the results quite satisfactory. The estimation of μ gives accurate results, obtaining RMSE values definitely lower than GLM and ARX ones. The network is perfectly able to do a precise point forecast. Thanks to this exact prediction on μ, we estimate a value for σ, which is less than half than the ones estimated in GLM and ARX. Obviously, we cannot recover results as precise as the ones observed in Python, but we can still consider this probabilistic forecast as a huge improvement with respect to the other methods considered. The point estimation is extremely accurate and the estimated Gaussian distributions are also quite precise and tight, as shown by the evaluation techniques. The Pinball Loss is way lower than the ones of GLM and ARX. The Backtest gives results which are worse, but still comparable to the ones of GLM and ARX. The worse precision of the Backtests is due to values of σ way smaller than the ones estimated in the other methods. As an example, in the figures below we can observe Confidence Intervals on the test set 2012 (Figure 22) together with Pinball Loss function and Backtested Confidence Interval (Figure 23).

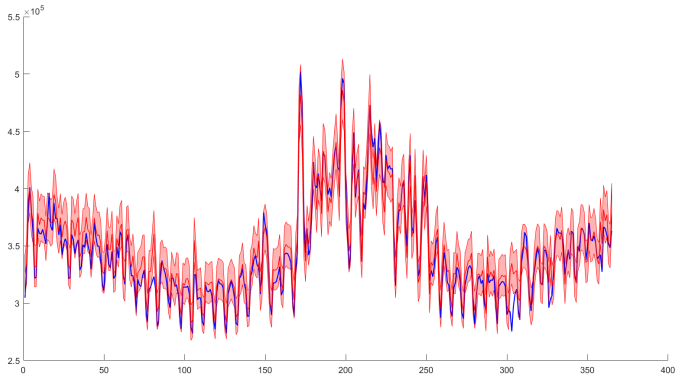

Fig. 22: NAX power consumption middle-term density prediction - test year 2012

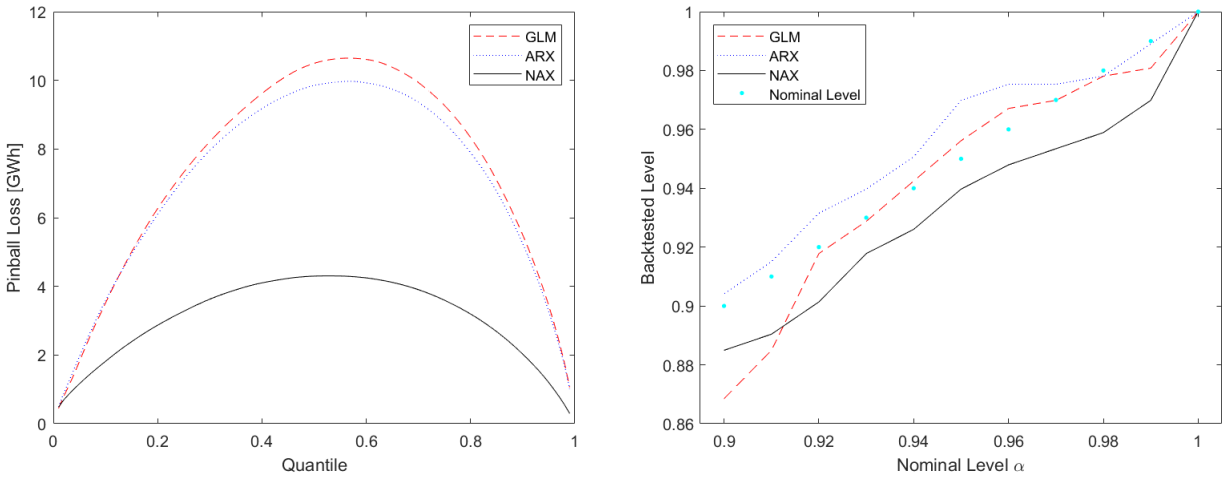

Fig. 23: Pinball Loss function and Backtested CI - test year 2012

Moreover, Table 12 shows a comparison between RMSE of GLM, ARX and NAX on the different test sets (from 2012 to 2016).

	RMSE [GWh]		
	GLM	ARX	**NAX**
2012	26.69	25.96	**11.15**
2013	31.03	31.20	**12.63**
2014	26.78	28.65	**11.64**
2015	28.51	31.22	**13.77**
2016	28.18	31.07	**11.98**

Tab. 12: MAPE, RMSE and APL for GLM, ARX and NAX on different test sets (from 2012 to 2016)

Overall, we can still consider this network as a significant improvement, although not comparable with the one implemented in Python.

7 References

[1] Azzone, M., & Baviera, R., (2020). Neural Network Middle-Term Probabilistic Forecasting of Daily Power Consumption.

[2] Baviera, R., & Messuti, G., (2019). Daily middle-term probabilistic forecasting of power consumption in North-East England. *Mimeo*.

[3] Hatalis, K., Lamadrid, A.J., Scheinberg, K., & Kishore, S., (2017). Smooth Pinball Neural Network for Probabilistic Forecasting of Wind Power. (p. 3)

[4] Hyndman, R., & Fan, S. (2010). Density forecasting for long-term peak electricity demand. Power Systems, IEEE Transactions on.