Half-title page, prepared by publisher

Publishers' page

Full title page, prepared by publisher

Dedication Page

(optional)

Contents

Chapter 1

Bayesian approach to energy load forecast with neural networks

1.1 Introduction

Short and long-term forecasts within various types of financial markets have become increasingly important during recent years. The latter is even more evident within the highly competitive scenario of electricity markets. Lately forecasting of possible future loads turned out to be fundamental to build efficient energy management strategies as well as to avoid energy wastage. Such type of challenges are difficult to tackle both from a theoretical and applied points of view. In fact, latter tasks require sophisticated methods to manage multidimensional time series related to stochastic phenomena often highly interconnected. Between the most promising results are those obtained exploiting machine learning approaches in general and those associated with *deep neural networks* (DNN) architectures, in particular. DNNs have shown their power in handling complex temporal data. Nevertheless, mostly because of the large number of parameters often needed to describe realistic models, these neural networks are often hard to regularize to out of sample data, moreover they suffer from lack of interpretability. In what follows, we propose a novel approach to energy load time series forecasting based on combination of neural networks (NNs) and probabilistic programming. Merging these two approaches we can bring uncertainty both in predictions and representations, powerful regularization with priors and show a highly efficient way to build complex neural network architectures for forecasting.

Probabilistic programming is a tool that allows for highly customizable as well as efficient random models, and it is used to get insights from given data. Moreover, such an approach permits to build NNs whose learning performances often beat those reached by mean of classical, possibly mul-

tivariate, techniques. The main idea is to follow the Bayesian approach, which implies to specify priors to inform and constrain machine learning models, then getting uncertainty estimation in form of a posterior distribution. A second step is based on Markov Chain Monte Carlo (MCMC) sampling algorithms used to draw samples, from above mentioned posteriors, to estimate these models. That is why, more recently, variational inference algorithms have been developed, obtaining solutions that are almost the same robust as MCMC, but faster. In fact, the latters, instead of drawing samples from the posterior, are addressed to fit a distribution (e.g. Gaussian, Poisson, etc.) to the posterior, turning a sampling problem into an optimization one. Recent innovations in variational inference allow probabilistic programming to scale model complexity as well as data size, which allows us to apply this type of techniques to multi-layer DNNs. Having conducted a large set of *experiments* using real data and by mean of different machine learning algorithms (e.g. using Random forest regressors, multi-layer NNs, etc.) we have decided to focus our attention on the use of NNs as the main learning algorithms. Hence, we start describing full process of data preparation and feature selection. Then, after statistical analysis of our time series, we discuss stationarity as well as other typical properties characterizing considered data, also choosing correct data normalization. Later on, we will discuss how to define the *right* forecasting model. From a more operative point of view, we first select validation subsets of given data, also performing related hyper-parameters optimization, which allows us to obtain the optimal number of layers and kernel sizes for the particular NNs architecture we develop. The next step regards the model training and validation of results with visualization. Subsequently, we apply what is known as *Automatic Differentiation Variational Inference* (ADVI) to regularize the model even better. In this spirit, the obtained priors will act as regularizers, aiming at minimizing the ANNs weights. We also show such a procedure is mathematically equivalent to consider an L^2 loss term that penalizes large weights into the objective function, with the difference that, acting as written before, we learn this regularization directly from the data. Last but not least, we explicitly show how to obtain model uncertainty, while forecasting, and we compare basic and Bayesian ANNs performances, focusing the analysis on strengths and weakness of both models.

1.2 How to train neural networks: different approaches

Before enter into a detailed description of possible NNs architectures, let us start with the definition of a computationally modelled neuron. Essentially it is a network of interconnected functional elements each of which has several inputs, but only one output. We can formalize it as follows

$$y(x_1, ..., x_n) = f(w_1x_1 + w_2x_2 + ... + w_nx_n) \ , \tag{1.1}$$

where w_i are parameters, f is the activation function, that is usually assumed to be non-linear, e.g. a sigmoid, or as rectified linear unit (ReLU). In this setting a relevant role is played by the $single-layerperceptron$ architecture, characterized by n inputs and one output, and formally defined as follows

$$y(x_1, ..., x_n) = f(w_1x_1 + ... + w_nx_n) \ . \tag{1.2}$$

To gain flexibility in learning the decision hyperplane, we add a bias term w_0 to the product sum: $y(x_1, ..., x_n) = f(w_0 + w_1x_1 + ... + w_nx_n)$, which implies the implementation of a simple algorithm mainly inspired by biological neural system. It is worth to mention that the latter can learn some functions, but still has limited capabilities. A *multilayer perceptron* is then defined by connecting the output of a *basic perceptron* to the input of another one.

The hidden layer maps inputs to the feature space and to the classification space. The latter allows to simplify the output layer procedure scheme. In fact, each hidden unit computes a separation of the input space, so that their combination can carve a polytope in the input space allowing output units to distinguish polytope membership. Basically, stacking the layers in neural network, we achieve a superposition of functions $f_1, \dots, f_n$, where n is the number of layers, namely

$$y(x_1, ..., x_n) = f_n(...f_1(w_0 + w_1x_1 + ... + w_nx_n)) \ . \tag{1.3}$$

1.2.1 *Backpropagation*

The backpropagation approach is a common method to *train* ANNs. It is usually used in combination with some numerical optimization techniques, e.g. the gradient descent method, in order to reduce computational efforts. The basic algorithm repeats a two phase cycle: the propagation cycle and the weights updating. After having a new input into a neural network, it is propagated (e.g. by multiplying each input by weights, then a non-linearity

is applied before passing the result to the next layer) forward through the network. The latter scheme is repeated *layer by layer*, until it reaches the output one. The output of the network is then compared to the desired output, using a loss function, that is measuring the result of our problem. The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output. Backpropagation uses these error values to calculate the gradient of the loss function with respect to the weights in the network. In the second phase, this gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function. Technically, backpropagation is an algorithm to calculate derivatives in the computational graph, optimization process can be discussed as a separate part of the method. The network is trained for several iterations, usually, having the whole training dataset as input for several times - its called an epoch. After the training the neurons in the intermediate layers organize themselves in such, so every neuron is supposed to learn different features of the input space. If it does not happen and we got several neurons learnt same feature - it leads to the overfitting of a model. After training, when an arbitrary input pattern is present which contains noise or is incomplete, neurons in the hidden layer of the network will respond with an active output if the new input contains a pattern that resembles a feature that the individual neurons have learned to recognize during their training.

1.2.2 *Genetic algorithms*

Genetic algorithm is a variant of the well known *global random search algorithm*. It is very often used for a rather large set of optimization problems, and as far as training a neural network is an optimization process (choosing the weights in order to minimize the loss function), we can apply the *genetic approach* to our problem as well. The main advantage of genetic algorithms is that they do not use gradients for optimization. This implies a dramatic reduction of the probability of been confined in local minima. Nevertheless, it is worth to mention that, when dealing with concrete situation about training NNs, it is rare the case in which we are interested in finding an exact global minima. In fact, even if training NNs, see, e.g., [Montana and Lawrence (1989)], is a procedure that can be reduced to a particular type of optimization problem, with own metric, the real task has also to take into account overfitting. The latter means that in concrete

cases, we have to find local minima that do not overfit neither the training scheme, nor the test set. Basically, genetic algorithms perform operations on what are called *chromosomes*, which could be defined as set of features. Continuing the biological analogy, a *population* consists of many chromosomes. In this setting becomes essential to establish a proper measurement of fitness. In fact the optimization scheme is essentially based on the latter to set the right optimization direction. It is worth to mention that all genetic algorithms are characterized by a standard set of basic operations, including:

- random initialization of the preliminary population;
- in-loop evaluation of every chromosome by measuring its fitness;
- comparison with the minimal desired fitness;
- selection of the fittest subset of chromosomes;
- *crossing-over* operation, namely a features exchange from the selected subset of chromosome;
- *mutations* introduction, which are random changes applied to randomly chosen features of the chromosomes;
- return to the 2nd point.

Because of their general structure, genetic algorithms can be used in almost any optimization problem, spanning from ANNs training, to the definition of best performing structure. During training, every chromosome for genetic algorithm comprises from all the connection weights from ANN. When searching for best structure of ANNs in the network, chromosomes possesses information mostly about number of layers and neurons in each layer.

1.2.3 *Bayesian approach*

To adopt a Bayesian treatment of NNs we assume a prior distribution $p(W)$ over the weights. It is common to assume that the weights are a-priori independent, i.e. $p(W)$ factorizes into a product of elements each of which refers to an individual weight. Typical choices for the prior distribution are zero-mean Gaussian, or Laplace, distributions. Nevertheless, also scale-mixture priors have shown to be effective. Moreover, the output of an NN can be interpreted as a likelihood function $p(D|W)$. The prior together with the likelihood induce a posterior distribution $p(W|D)$ over the weights. However, $p(W|D)$ is typically intractable. The aim of variational inference is therefore to approximate $p(W|D)$ with a simpler variational distribution

$q(W|\nu)$ and then perform inference based on $q(W|\nu)$, where ν denotes the set of variational parameters that are optimized. Then $q(W|\nu)$ is as close as possible to the posterior $p(W|D)$, provided the adoption of an appropriate metric. In variational inference, see, e.g., [Fox, C. and Roberts, S. (2012)], it is common to minimize $KL(q(W|\nu)||p(W|D))$. We adopt the common mean-field approximation with an approximate posterior $q(W|\nu)$ that factorizes into a product of factors $q(w|\nu w)$ for each weight $w \in W$. We select the variational distribution $q(w|\nu w)$ to be Gaussian, with mean and variance parameters $\nu m = (\mu w, \sigma m)$. This doubles the total amount of parameters to optimize, while adding useful uncertainty estimates to the weights.

1.3 Automatic Differentiation Variational inference

The main purpose of variational inference is to turn the task of computing a posterior into an optimization problem. We set a parametrized family of distributions $q(W) \in Q$, aiming to find a distribution from this family that minimizes the Kullback-Leibler (KL) divergence to the exact posterior. Traditionally, the concrete implementation of a variational inference algorithm (VIA), implies a lot of difficulties with optimization algorithm setting as, e.g., specifying a correct variational family, selecting and computing corresponding loss function, calculating derivatives, and running an iterative gradient-based optimization method. The Automatic Differentiation VIA (ADVI) aims to solve these issues automatically. The user has just to specify the model, expressed as a program, and ADVI, see, e.g., [Kucukelbir (2016)], automatically generates a corresponding variational algorithm. The idea is to first automatically transform the inference problem into a common space and then to solve the associated variational optimization. Solving the problem in this common space allows to resolve variational inference for a large set of models. In more detail, ADVI can be split into the following steps.

- ADVI transforms the model into a model that has unconstrained real-valued latent variables. In particular, it transforms $p(x, W)$ to $p(x, \zeta)$, where the mapping from W to ζ is built into the joint distribution. After this we do not have constraints on variable W anymore; ADVI then sets the corresponding variational problem on the transformed variables, that is, to minimize $KLq(\zeta)||p(\zeta|x)$ as described before. With this transformation, all latent variables are defined on the same space.

ADVI can now use a single variational family for all models;
- ADVI recasts the gradient of the variational objective function as an expectation over q. This involves the gradient of the log joint distribution with respect to the latent variables $\nabla W logp(x, W)$. Having the gradient as expectation allows to use Monte-Carlo methods;
- ADVI re-parametrizes the gradient in terms of a standard Gaussian. To do this, it uses another transformation, this time within the variational family. This second transformation enables ADVI to efficiently compute Monte Carlo approximations;
- ADVI uses noisy gradients to optimize the variational distribution with adaptively tuned step-size.

1.4 Data preparation

On the figure 1.1 there is a sample time series from the dataset. It is easy to see that it looks periodic, and, which is the most important feature, stationary. Hence it is easy to analyse its data as well as to provide related forecasts. We check this hypothesis with Augmented Duckey-Fuller test first (basically testing from mean reverting, which can be considered as equal test). This process refers to a time series that displays a tendency to revert to its historical mean value.

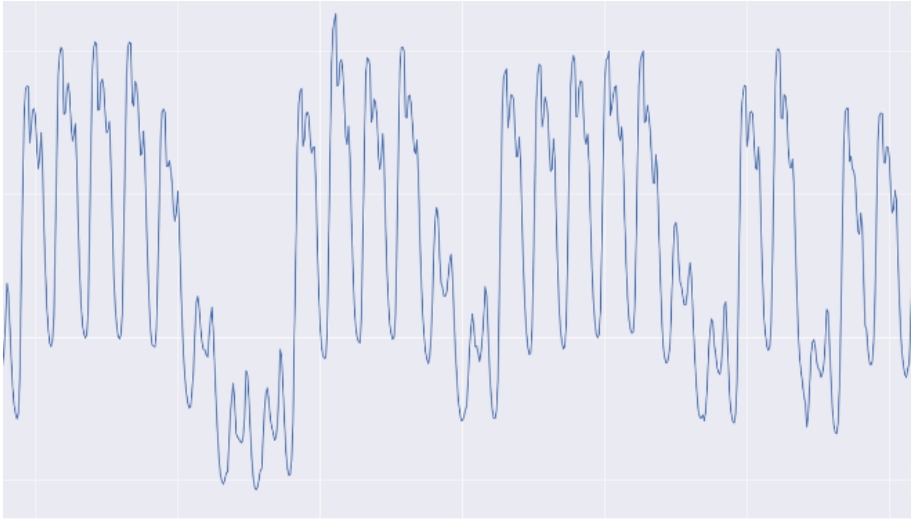

Fig. 1.1 Energy load hourly time series sample

Mathematically, such a (continuous) time series is associated to an Ornstein-Uhlenbeck process. This is in contrast with the Brownian motion case (a symmetric random walk, provided suitable rescaling, from the discrete point of view), since the latter memoryless, being a martingale. We are obtaining rejection of non-stationarity hypothesis with 5% of confidence on the whole dataset, moreover we also check it calculating Hurst exponent. The goal of the Hurst exponent is to provide a scalar value that will help to identify whether a series is mean reverting, is close to a random walk behaviour, or has a trend component, of course up to a given level of statistical significance. For the sake of completnesse, let us recall that a general time series can then be characterized in the following manner:

- $H \leq 0.5$ - the time series is mean reverting ;
- $H = 0.5$ - the time series can be seen as geometric Brownian motion realization ;
- $H \geq 0.5$ - the time series has a trend component.

Analyzing our dataset we find a Hurt Exponent that equals 0.130744832753, hence considered data form a stationary time series. Having stationary time series is very useful, because it is possible to normalize their data using statistical parameters from dataset, as, e.g., mean, standard deviation, minimal or maximal value, etc. The data normalization procedures lead to better performance of machine learning models. For normalizing our data we chose the $z-score$ normalization, subtracting the mean of time window and then dividing by it is standard deviation. Our dataset is then splitted in the *training component* and in the *test component*, as follows: we take 80% of first historical values as train set and check performance on the last 20%, avoiding mixing values from the past and the future. Every training example consists of a pair (X_i, y_i), where X_i is a time window of length 24, for a single day of measurements, while y_i is a value in the next hour. Our training set consists of 31604 examples, while the test set is composed by 12291 examples.

1.5 Neural network architecture

In what follows, we are going to consider a basic model model which will be exploited to implement a $2-layer\ perceptron$ NN. In such a scheme, the first layer consists of 64 neurons whose task is to learn a representation of an 24-dimensional input, while the second layer consists of 16 neurons

to *squeeze* the representation. Latter step is crucial because it allows us to reduce the set of features which, in principle, are responsible for the dataset characteristics, aiming at picking up just the more representative between them. We would like to underline that, from a more classical point of view in multivariate statistics, such technique is similar to the ones commonly grouped under the *Principal Component Analysis* (PCA) label. As non-linear function, we have chosen a rectified linear unit (ReLU) component which is considered as a benchmark in most novel NNs, particularly because it has proven to learn faster, while maintaining a high robustness versus overfitting effects. As last step, we exploit a single neuron, without non-linear term, to obtain real value of prediction. In order to apply Bayesian approach to our NN-model, we also have to set prior distributions on the model's weights. Therefore, accordingly with the *state of the art* literature, we consider our weights to be Gaussian distributed, with *varying*, according to analysed data, parameters.

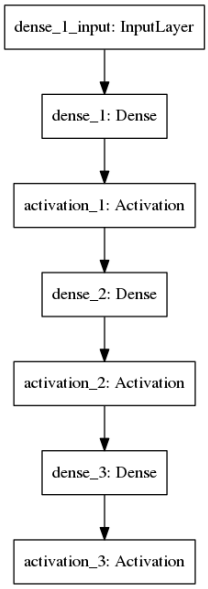

Fig. 1.2 Energy load hourly time series sample

In particular, we start setting the mean of the Gaussian distributions

to zero for all the *transision matrices*, namely from from the input to the first layer, from this to the second layer, as well as from the second layer to the output. Instead, three different *half Gaussian* distributions have been considered to describe standard deviations. For the sake of completeness, let us recall that a half Gaussian is chosen to avoid negative values occurring with $\mathbb{P}(X \leq \mu) = 1/2$, when considering $X \sim N(\mu, \sigma)$.

Such parameters values essentially play the role of model regularizers, being learned directly from the data. Analogously, we consider model output as normally distributed with unknown parameters. The latter will be a distribution from where predictions will be sampled.

By implementing the backpropagation approach, we train our model on 100 epochs, setting a step value of 0.002, and following [D. Kingma, J. Ba (2015)]. The goal is to minimize the mean squared error characterizing the *dataset train phase* and, therefore, the test.

Conerning the variational inference scheme, we first implement 5000 steps of ADVI, where the variational posterior distribution is assumed to be spherical Gaussian, without correlation of parameters, and then we fit it to the true posterior distribution.

The means and standard deviations of the variational posterior are referred to as variational parameters. In such a a framework the chosen optimizer is the Adagrad algorithm. After obtaining variational posterior we can draw samples from it to generate the trace of them. The latter has been provided taking a length of 5000 for both the confidence and the inspection phases of associated parameters. As last step, we generate data then exploited to provide suitable predictions on the hold-out set in our case, by implementing a posterior predictive check with 100 samples. The extrapolated mean of these samples is considered as a prediction for the next hour and, analogously, related standard deviation is taken as the uncertainty level of the model. In particular, higher uncertainty values mean that our model *is not sure* about sampled data nature. Namely, from a statistical point of view, the confidence interval has to be taken rather large, so large that the statistical inference has not a concrete sense. The resulting message is then very clear: *do not rely on the produced forecast*. We would like to underline that such type of *negative* suggestions are, contrary to what can be thought, very useful for real investors, pushing them to take more prudential positions in the market. On the other hand, an agent particularly inclined to risk, can use latter results to consciously bet, aspiring to obtain high revenues while accepting high level of loosing his investment. See also [Di Persio, L. and Honchar, O. (2016)] for a concrete case study

where different NNs have been implemented and compared in results, as an effective models to stock price predictions.

The backpropagation training has been implemented in the *Keras' framework*, see [Fox, C. and Roberts, S. (2015)], since it allows for fast prototypization, with the additional advantage of having as built-in features, all the essential functions we need. Concerning the Bayesian approach, we have defined the network by both $Theano$ and $NumPy$ primitives, while, for ADVI as well as fo the PyMC3 sampling, we refer to [A. Patil, D. Huard, C. J. Fonnesbeck (2016)]. In the following section results obtained by mean of the backpropagation approach are compared to those derived by using the variational inference one, according with the metrics: mean absolute error on the test set, robustness to overfitting, smoothness of parameter learning process, hence MSE loss for backpropagation and evidence lower bound for ADVI. Moreover, within the Bayesian framework, we also reported criticism as well as a detailed analysis about how the proposed model can estimate uncertainty of its prediction.

1.6 Results

1.6.1 *Bayesian approach*

After 5000 iterations of ADVI method, we can see the following picture of evidence lower bound (ELBO) on the figure 1.3, which clearly shows that it converges to the value $1.0529e + 07$. It is interesting to consider on posteriors of the weights and regularization parameters on the figure. First matrix of weights lies in the range $[-6, 6]$, while the second ones is in the range $[-4, 4]$. Therefore we have a smaller range for the second matrix's weights, a result which depends on standard deviations that are also treated as random variables. Moreover, regularizer in the first weight matrix has a value of 1.5, while in the second it equals 1.3, and in the third is around 0.4. Such differences suggest that it makes sense to change the amount of regularization to be applied at each layer of the network.

The figure is a sample plot of real points from the test set (black line) and predictions on it (blue line). The mean absolute error (MAE) is 3234. Moreover we have also reported a plot of standard deviations, as a indicator of uncertainty estimation.

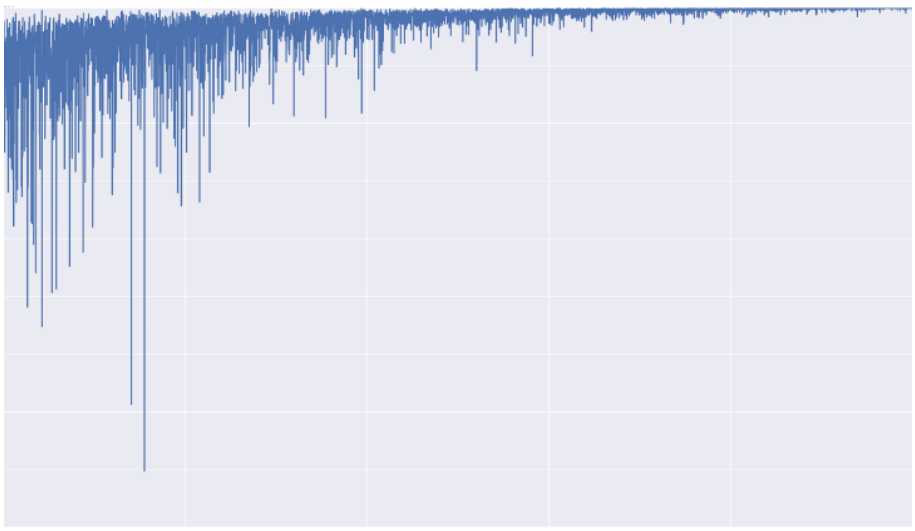

Fig. 1.3 Energy load hourly time series sample

1.6.2 *Backpropagation*

After running 100 epochs of backpropagation, we obtain a $MSE = 0.0133$ on the normalized data. On the figure we can see the plot of a decreasing, regular, loss function, which, besides, does not show any overfitting effects. We have also provided a sample plot of real points from the test set (black line) and predictions on it (blue line), here the MAE equals 313.69, while a graphic comparison of results main metrics is the following:

MSE/ELBO	MAE	Time for train (sec)	Time for prediction (sec)
0.0433	1010.75	100	0.2
1.0529e+07	3234.55	8000	2500

Comparison of main metrics of results

1.7 Discussion

As we can see from the last section, regular deterministic NN trained with backpropagation shows much better (up to ten times) results in accuracy of forecasting due, mainly because of the MAE. To check the impact of

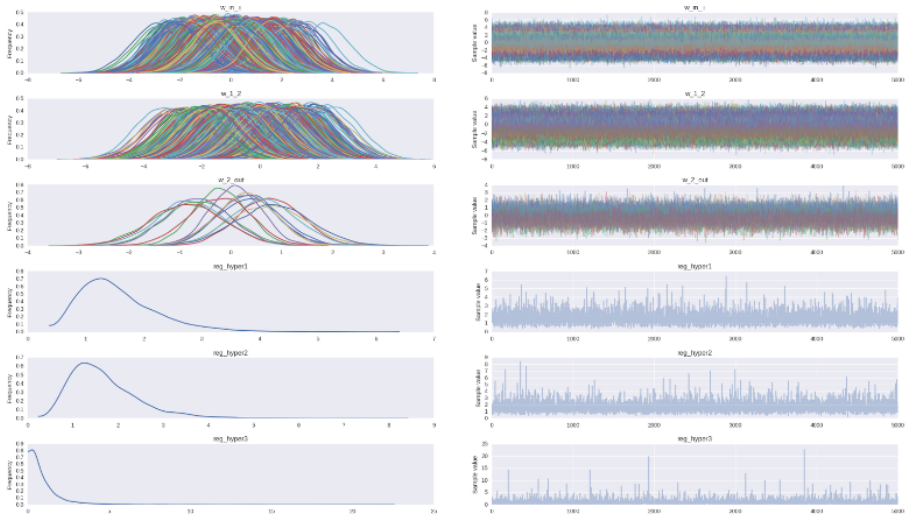

Fig. 1.4 Details of variational inference

Bayesian regularization on results we have applied popular regularization technique called Dropout, see, e.g., [Y. Gal, Z. Ghahramani (2016)] to the NN previously trained with backpropagation. While applying this regularization, we have randomly set some weights to zero, therefore training a rich ensemble of different architectures, while avoiding neurons co-adaptation. It is worth to mention that we have shown as the Dropout technique acts as a Bayesian regularization. The latter allows us to inspect better infer its dynamics as well as provided results. Let us recall that we have applied the Dropout scheme with *drop rate* $= 0.5$ between the first two layers. We have reported the obtained outputs in the following figures, where the plot of loss function clearly shows small overfitting effect. This suggests to stop earlier the training phase, namely before the loss starts to grow again. On the figure we can see that the forecasting results with MAE reach a lower level if compared with the one without Dropout, when $MAE = 978.3$.

Provided results allow for more accurate predictions, also suggesting that it is better to rely on the model trained with backpropagation with deterministic weights. Moreover, such results also show evident improvements when regularization, within the Bayesian approach, is taken into accounto. Nevertheless, it is important to recall that latter approach needs much more deep work on setting the priors, since the initial choice of considering Gaussian distributions is not an invariant one, being used just as

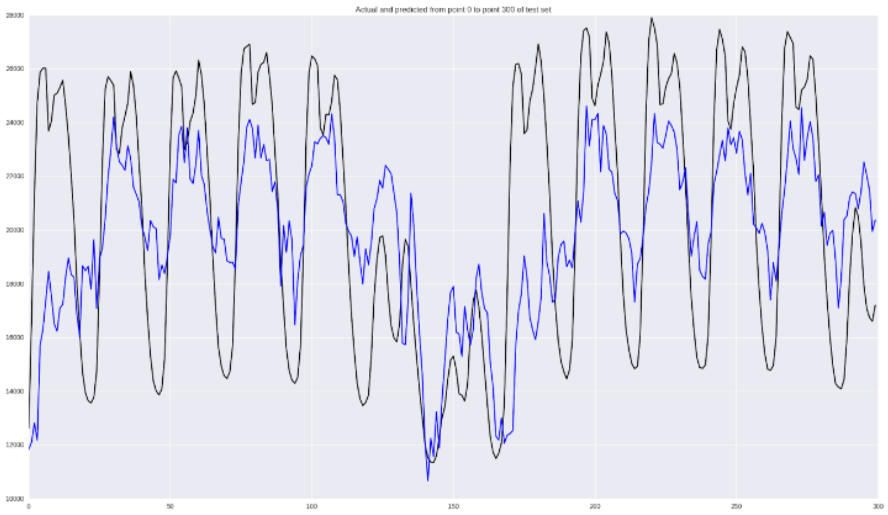

Fig. 1.5 Bayesian approach forecasting results

trigger point. In particular, during the Bayesian scheme life, other distributions have to be considered to improve forecasting, resulting in a huge computational time growth.

1.8 Conclusion

The present contribution would serve as a concrete analysis of energy load time series forecasting by mean of different NNs architectures and approaches. We have provided a self-consistent treatment of the whole *pipeline* which is followed as a standard by both practitioners and data analysts involved in the subject, hence starting from real time series of interest (data set preparation in our treatment) to forecasts details (training NNs' models and obtaining associated outputs). We have considered predictor feed-forward NNs, and we have shown its strength in time series forecasting with high accuracy. Moreover, we amplify have also improved our model by setting priors on the weights, setting robust generalization and clearly identify the role played by model uncertainty. The type of conducted analysis allows for huge advantages in practical use. In fact, we do not only prevent overfitting to training set, but also estimate reliability of our forecast in any moment. It is essential to note that latter feature cannot be reached with deterministic approach to machine learning. Meanwhile, we

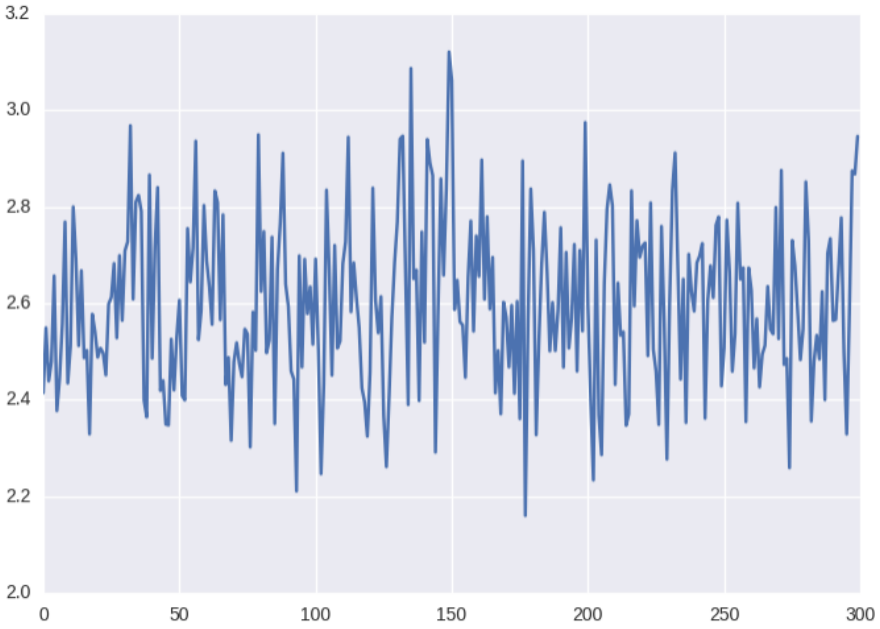

Fig. 1.6 Uncertainty estimation example

show that deterministic approach leads to better numerical results, but the generative model trained with variational inference, provided modifications we proposed, can show uncertainty in our prediction, so we suggest to use both models to get accurate predictions and understand the level of convenience about them. Ways to increase accuracy of our model have been also proposed and tested.

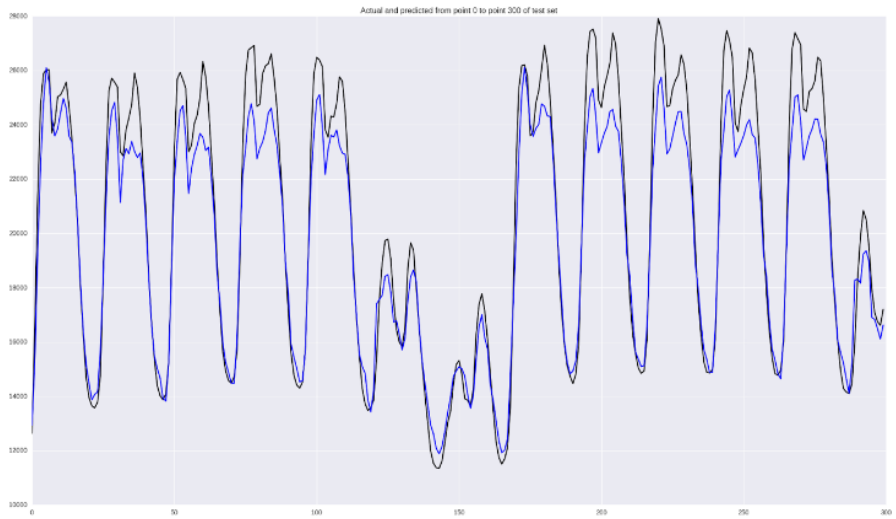

Fig. 1.7 Determenistic NN forecasting results

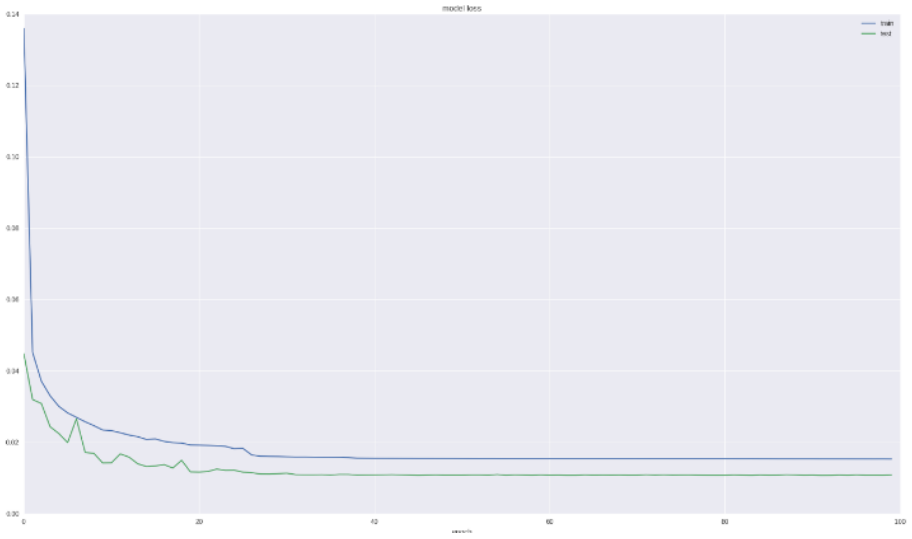

Fig. 1.8 Uncertainty estimation example

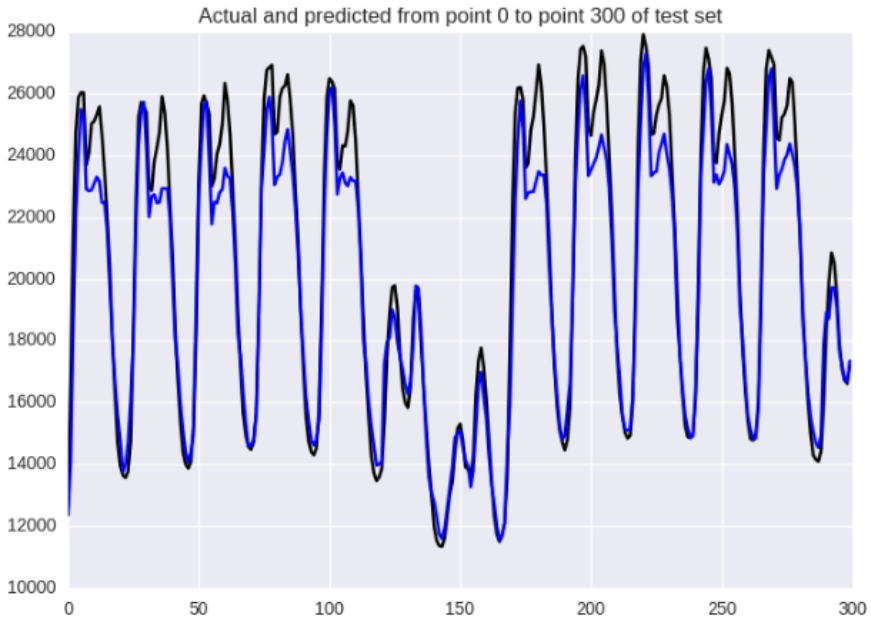

Fig. 1.9 Dropout model forecasting results

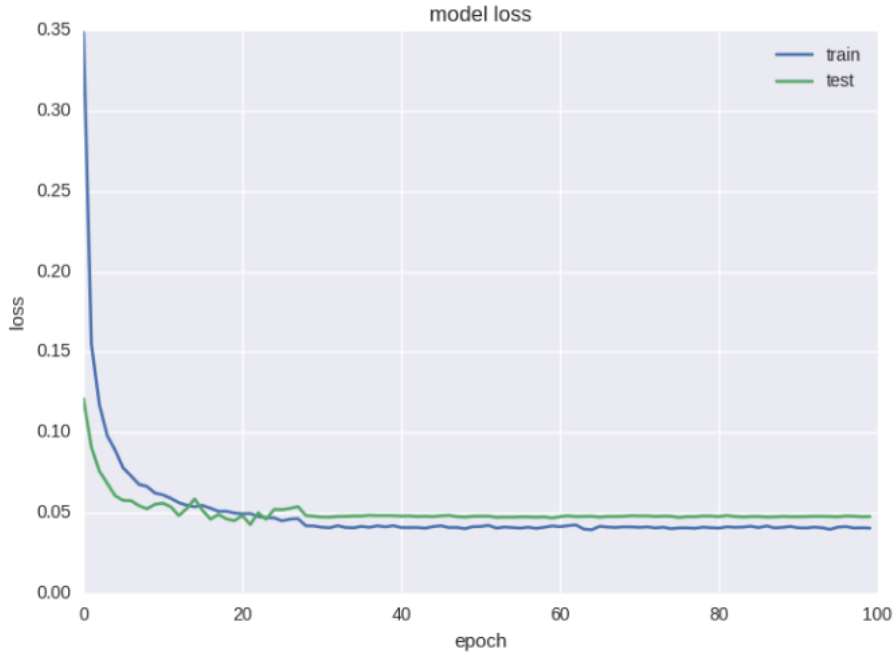

Fig. 1.10 Uncertainty estimation example

Bibliography

Di Persio, L. and Honchar, O. (2016). Artificial neural networks architectures for stock price prediction: Comparisons and applications *International Journal of Circuits, Systems and Signal Processing*, 10, pp. 403-413.

A.Kucukelbir, D. Tran et al. (2016). Automatic Differentiation Variational Inference, *Journal of Machine Learning Research X*.

David J. Montana and Lawrence Davis. (1989). Training Feedforward Neural Networks Using Genetic Algorithms, *BBN Systems and Technologies Corp.*.

Fox, C. and Roberts, S. (2012). A Tutorial on Variational Bayes, *Artificial Intelligence Review*.

Diederik Kingma, Jimmy Ba (2015). Adam: A Method for Stochastic Optimization, *arXiv:1412.6980*.

Chollet, François and others. (2015). Keras, *Github,* `https://github.com/fchollet/keras`.

Anand Patil, David Huard, Christopher J. Fonnesbeck (2016). PyMC: Bayesian Stochastic Modelling in Python, *Journal of Statistical Software*.

Yarin Gal, Zoubin Ghahramani (2016). Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning, *University of Cambridge*.

Index