Heating and Electricity Consumption Prediction

Giovanni Pelosi

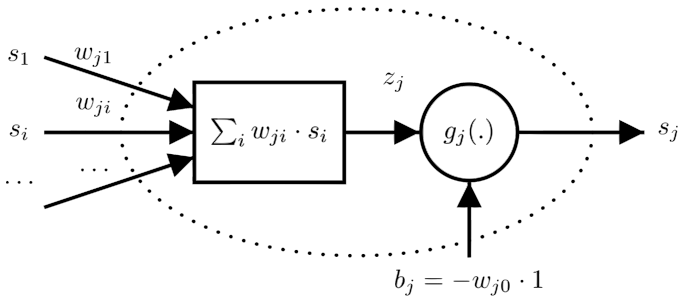

Fig. 1. Artificial Neuron.

***Abstract*— The goal of this project is to predict the heating and electricity consumption of an hospital for the next following 24 hours, using a DNN architecture. In particular, as a theoretical goal, we want to check if a LSTM neural network has a prediction accuracy similar to a single-layer neural network for which we carefully select a set of features.**

I. Introduction

In this paper we present the structure of a Deep Neural Network DNN and study how well it is able to learn from sequential inputs. Our goal is to verify if in certain condition a DNN is able to extract the important features that characterize sequential inputs. To do so we will compare Feed Forward Neural Networks FFNNs with Long Short Term Memory Networks LSTM. As an empirical verification we will compare these two classes of nets in predicting the heating and electricity consumption of an hospital. In section 2, we will introduce the basic structure of Artificial Neural Networks and how they are generally trained. In section 3 we will present the problem of predicting the heating and electricity consumption of an hospital and derive our conclusions.

II. Artificial Neural Networks

A. Artificial Neuron

An artificial neuron [Figure 1] is a component that tries to mimic the working principle of a real brain neuron. In its simplest implementation, it is characterized by four parts: A set of inputs s_i with their weights $w_j i$, an activation value z_j that is the weighted sum of the inputs, an activation threshold (also called bias) $b_j = -w_{j0} \cdot 1$ and an activation function $g_j(\cdot)$ that maps z_j into the output y. There are different kinds of activation functions; normally they are derivable such as sigmoids or linear functions, but they can be also of other forms, such as ReLU, depending on the application.

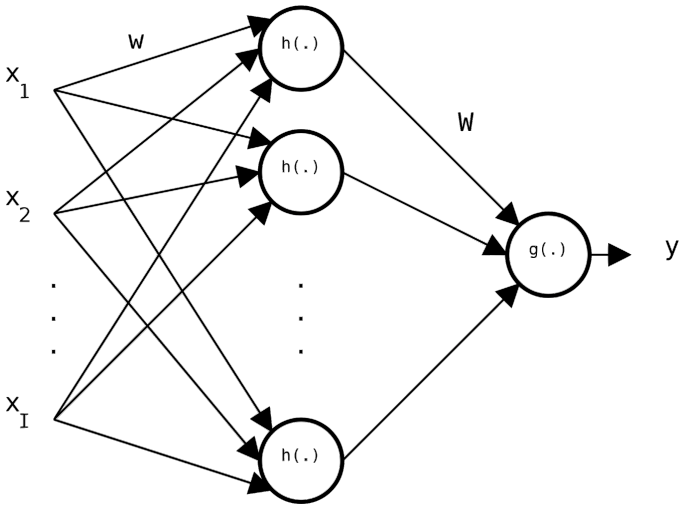

Fig. 2. Artificial Neuron Network. This figure represents a single layer feed forward neural network. It is called single layer because it has only one hidden layer. In particular the net has I inputs, J hidden neurons and one output. The weights are w_{ji} from the input to the hidden layer, and W_j from the hidden layer to the output
. The data flows according to the arrows' direction.

B. Artificial Neural Network

An artificial Neural Network is a network composed by a variable number of artificial neurons. Normally neural networks NN are subdivided in layers. In particular we identify an *input* layer, that takes the input samples and it is composed by a number of neurons equal to the number of input features. A variable number of *hidden* layers, each of those containing a variable number of neurons. And finally an *output* layer that contains a number of neurons equal to the number of outputs that we want (1 or more).

The most simple artificial neural network is a network in which each neuron of a layer is connected to all the network of the next layer, as in Figure 2. Moreover, normally these connections have the same direction, in the sense that the data flows from a neuron of a given layer to all the neurons of the immediately next layer. When this is true for all the connections we obtain the so called feed forward neural network FFNN, in which the input is propagated from the input layer, through the hidden layers, to the output layer.

C. Training an ANN

The general algorithm used to train an ANN is called $back-propagation$ or $gradientdescent$. The goal of back-propagation is to gradually update the weights of the net in order to reduce the error between the output y_n (computed by the net for the n^{th} input) and the target t_n (the "correct" result

that the net should output). For the net in Figure 2 we can write the output as: $y = g(\sum_j^J W_j \cdot h(\sum_i^I w_{ji} \cdot x_i))$ and the error as $E = \sum_n^N (t_n - y_n)^2$. Notice that the error is actually the sum for all the training samples of the squared difference between the targets t_n and the outputs y_n. Now, given the error, the back-propagation algorithm simply updates the weights of the net proportionally to a learning rate η and the partial derivative of the error.

$$w^{new} := w^{old} - \eta \frac{\partial E}{\partial w} \tag{1}$$

Back-propagation is a two-phase algorithm. In the first phase, called *propagation* the sample flows through the net until the output is computed. In the second phase the weight are updated (based on the error). Here we present the training procedure for a generic regression problem, in particular we introduce the updated formula for the last hidden layer. First, let's introduce some variables to simplify the notation:

$$\begin{aligned} a_j &= \sum_i^I w_{ji} \cdot x_i \\ b_j &= h(a_j) \\ A &= \sum_j^J W_j b_j \end{aligned}$$

Now we can derive the update formula.

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= \sum_n^N 2(t - g(A)) \cdot \frac{\partial}{\partial W_j}(t - g(A)) \\ &= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot \frac{\partial}{\partial W_j} A \\ &= \sum_n^N 2(t - g(A)) \cdot (-g'(A)) \cdot b_j \\ W_j^{k+1} &= W_j^k + 2\eta \sum_n^N (t - g(A)) \cdot g'(A) \cdot b_j \end{aligned}$$

To compute the partial derivative for the weights $w_j i$ we simply repeat the previous procedure. One theoretical advantage of back-propagation, beside being very simple, is that it can be used to compute the updates of the weights also for very deep networks. Unfortunately, this is cannot be easily exploited in practice, due to an issue known as vanishing gradient. We will analyze this issue and others in more details, in the following sections.

D. Issues in training an ANN

Three are the main issues that characterized the training of neural networks:

- Convergence
- Local optima
- Generalization and overfitting

a) Convergence: Since the learning rate η directly influence the convergence, selecting the wrong one we might end up with an algorithm that converges very slowly in case of plateaus, or even that does not converges at all. This two situations are quite problematic and many solutions have been introduced to try to reduce this problem. The most simple solution is to set a schedule on the learning rate η in order to make it gradually decrease after a certain number of iterations. This method guarantees convergence but it might still produce very long training sequences. More sophisticated methods are $gradientdescentwithmomentum$, $quasinewtonmethods$ and $conjugategradient$.

b) Local optima: The second big issue of gradient descent is that (unless the function we want to optimize is convex, normally not the case in neural networks) we are not guaranteed to reach a global optimum. Indeed we almost always end up in local optima. There are not solutions to this problem that guarantee us to eventually reach a global optimum. Some ways to somehow reduce the issue is to retrain the net multiple times ($multiplerestart$) or exploit some $randomizedalgorithm$.

c) Generalization and overfitting: Unless the network contains only a very small number of neurons (which is normally not the case), neural networks contain usually a very large number of weights. This fact give to neural networks the possibility to approximate theoretically any function. This might seems a very nice property, and indeed it is, but it must be used carefully. Neural networks can in fact overfit. That means that they can learn so well the training set and commit zero error on it, but perform very poorly on the test set (producing a lack of generalization). To reduce this issue, two are the normally exploited solutions: $earlystopping$ in which we stop the training of the net immediately after the net reaches a desired performance or after a fixed number of iterations and $weightdecay$ in which we penalize large weights.

E. Stochastic Gradient Descent Algorithm

Stochastic gradient descent SGD is a revised version of gradient descent. In particular it is an online algorithm (differently from gradient descent). Indeed in SGD the weights are updated on each iteration while in gradient descent we need to wait the end of the epoch (i.e. after all the training set as been propagated through the net and we have a computed output for each sample). In practice, SGD is implemented as a $mini-batch$ version. That means that instead of updating the weight at each very iteration we average the gradient over a mini-batch (normally 32 or 64 input samples). In this way the learning is generally faster and still online. Unfortunately even SGD is not perfect since the gradients of the mini-batches might have high variance and produce a non very smooth learning curve. Some of the ways to improve SGD are:

- Momentum: in which the algorithm remembers the previous update at each iteration and determines the next update as a linear combination of the gradient and the previous update.
- AdaGrad $AdaptiveGradientAlgorithm$: SGD with per-parameter learning rate [increases the learning rate for more sparse parameters and decreases the learning rate for less sparse ones].

- RMSProp *RootMeanSquarePropagation*: in which the learning rate is adapted for each of the parameters. It divides the learning rate of a weight by a running average of the magnitudes of recent gradients for that weight.

F. Adaptive Moment Estimation

Another very used optimization algorithm is Adaptive Moment Estimation, also known as *ADAM*. It basically combines the AdaGrad and RMSProp algorithms. In particular, it adapts the parameter learning rates based on the average first moment (the mean) as in RMSProp, and on the average of the second moments of the gradients (the uncentered variance). Some nice properties of ADAM are that it is invariant to diagonal re-scaling of the gradients and it is suited for problems that are large in terms of data and/or parameters.

G. Deep Neural Networks

Deep neural networks are artificial neural networks with a large number of hidden layers, typically more than 5. We can group up basically four kinds of networks:

- Feed Forward Neural Networks FFNN
- Recursive Neural Networks RNN
- Long Short Term Memory Neural Networks LSTM NN
- Convolutional Neural Networks CNN

Each of these network is more or less suitable for particular tasks. For instance LSTMs are generally used when it is required to learn sequences while CNNs are generally used for image analysis and classification. [It is not our purpose to enter into the details of how CNN works, but we will briefly introduce how RNNs and LSTMs are structured].

a) Recurrent Neural Network: The goal of RNN [Figure 3] is to try to memorize and subsequently exploit the past inputs. To do so, a new unit b is added to the hidden layer and a new input unit $c(t)$ to represent the value of b at time $(t1)$. b thus can summarize information from earlier values of x arbitrarily distant in time. Figure 3 represents a generic RNN. To train a RNN we basically unfold the neurons that represent the memory (the content network), as shown in Figure 4. Notice the in the unfold procedure we create similar arcs between the layers. The weights of similar arcs will be averaged after the training. Analyzing this structure from a theoretical point of view we might think that is possible to unfold the net for a desired number of times, in order to be able to learn and keep in memory correlations from a very far past and the present. Unfortunately this is not possible in practice due to the issue of vanishing gradient. Vanishing gradient is a known issue that prevent exploiting very deep architecture by sending the gradients of the weights of the first layers of the net to zero. In order to solve this issue, LSTMs were introduced.

b) Long Short Term Memory Neural Networks: LSTM are neural networks composed by LSTM neurons. These LSTM neurons [Figure 5] are more complex than the classic artificial neurons. A LSTM neuron is composed by 5 parts:

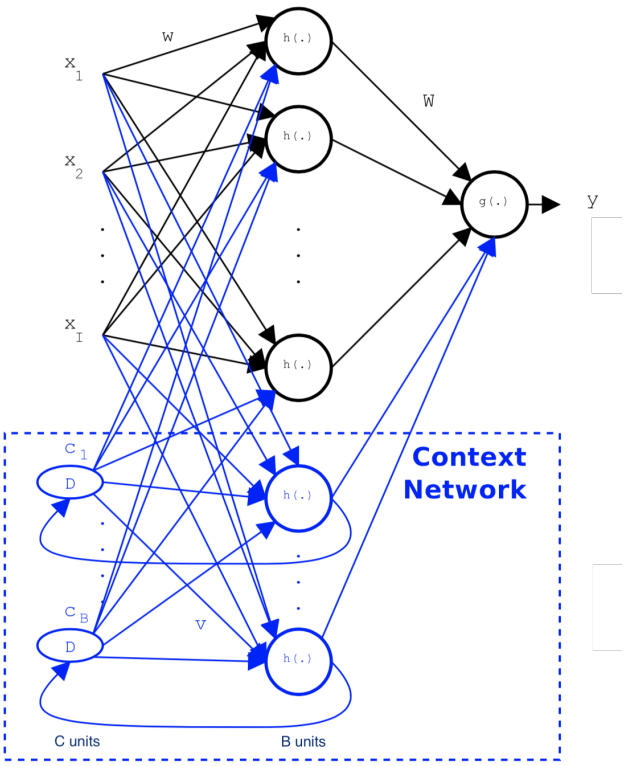

Fig. 3. Recurrent Neural Network. This Structure is known as Elman Network. The content network represents the memory.

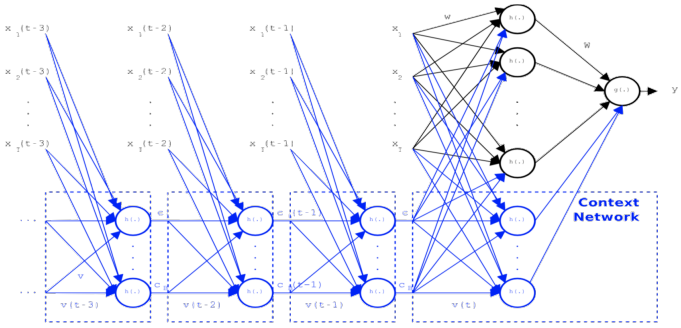

Fig. 4. Training of a RNN by unfolding the context network three times.

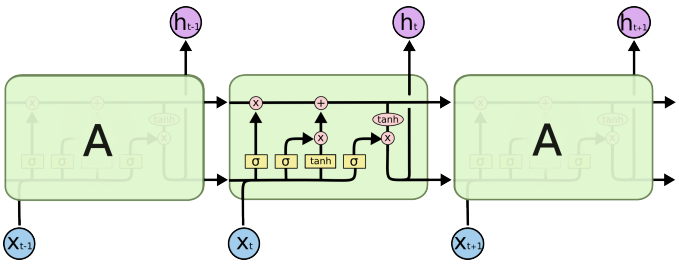

Fig. 5. Long Short Term Memory Neural Network. http://colah.github.io/posts/2015-08-Understanding-LSTMs

- Cell state: it runs straight down the entire chain, with only some minor linear interactions. It retains the memory of the neuron. It is very easy for information to just flow along it unchanged.
- Forget gate: it is used to decide which information to remove from the cell state. This decision is made by a sigmoid layer.
- Input gate: it decides what new information it is going to be stored in the cell state. This gate is formed by two components: the former is a *sigmoid* layer that decides which values of the cell state it will update, while the latter is a *tanh* layer that creates a vector of new candidates values (of the input) that can be added to the cell state. Finally the results of these two layers are combined.
- Memory Gate: First it multiplies the old cell state by the result of the forget gate, removing the things it decides to forget. Then it adds the scaled updates computed by the input layer.
- Output layer: It computes a filtered version of the cell state. First, it runs a *sigmoid* layer which decides what parts of the cell state it is going to output. Then, it puts the cell state through *tanh* (to push the values to be between -1 and 1) and multiplies it by the output of the *sigmoid* gate.

H. Pros and Cons of DNNs

DNNs are very powerful tools that can be used in many applications. One of the biggest advantages of DNNs is that they do not require domain knowledge to be used. In other words they can be seen as a complete automatic tool that takes raw input features and produce meaningful outputs. This anyway gives rise to one of the biggest problem of DNNs, that is our inability to control the learning process and our inability to understand the learned features. DNNs are indeed referenced as black-boxes for which we do not know what is going on inside. Two other major issues of DNNs are that they generally require a lot of data for training and that the initialization might strongly influence the learning rate and the learned solution.

III. Heating and Electricity Prediction Consumption

The goal of this project is to predict the heating and electricity consumption of an hospital of the next following 24 hours, using a DNN architecture. In particular, as a theoretical goal, we want to check if a LSTM neural network has a prediction accuracy similar to a single-layer neural network for which we carefully select a set of features.

A. The Data

a) Training data structure: The data [Figure 6] contains samples about the consumption of electricity and heating of an hospital. The data is divided in three seasons and each season is subdivided in two groups: one containing the heating data and the other the electricity data. Finally for each of these groups there are 28 data sets where each of them contains the data of 28 days (for training) and 1 (the next day) for testing. For each day we have 24 samples (one per hour). All the data has been previously normalized. There are 26 input features:

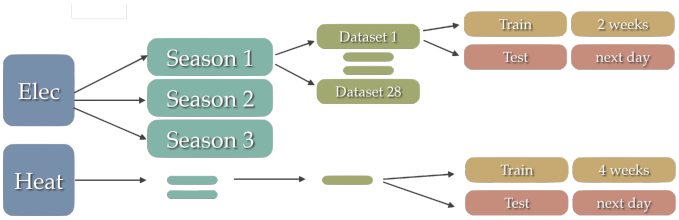

Fig. 6. Data structure.

- 4 weather forecast info for the specific hour (sample)
- 6 binary variables that identify the week day
- 1 binary variable that states if the day is holiday or not
- 12 weather forecast info of the 3 previous (hours) samples
- 3 variables indicating the electric (or heating) consumption of the same hour of the same week day of the previous three weeks

In particular we define a *small* data-set that represents the data with the first 11 features and a *big* data set that represents the data with all the 26 features.

b) Result data structure: Again the data is divided for each season and according to its type (heating or electricity). For each train set we have multiple outputs:

- matrices with the output results.
- figures that compare the real value of the target and the predicted one.

All the data-sets and the results are available at `https://github.com/giovannipelosi/heating-consumptions-prediction`.

B. Theoretical goal

With this project we want to try to answer to two questions:

- To discover if a DNN trained on the *small* data-set is able to predict with a accuracy similar to that of a single hidden layer NN trained on *big* data-set. Is the DNN able to learn the important features that characterize the given problem?
- Compare the results with a LSTM NN trained on the *small* data-set. Is a LSTM NN able to exploit the past to better predict the future?

C. Experiments

In this subsection we briefly introduce some of the most relevant experiments we performed. We can group them up if five big categories:

- First experiment with simple FFNNs: for this experiments we used FFNNs with very few layers and inner nodes in order to have a look at the complexity of the problem we were going to solve.

- Preprocess the data-set by removing seasonal components: starting from the results of the previous experiment we tried to preprocess the data-set in order to make the training process simpler. In particular we removed the (periodic) daily trend but we did not attempt to remove any linear trend. To test the preprocessing experiment we created two $de-season$ data-set, one starting from the *small* original data-set and one starting from the *big* one.
- Modify the structure and the training options of the nets: We performed different experiments modifying both the training options and the network layout. In particular we varied the number of layers, neurons for each layer and the optimization algorithm (choosing between SGD and ADAM). Navigate to `https://github.com/giovannipelosi/heating-consumptions-prediction` for details about the network structures.
- Data multiplication: In this set of experiment we tried to increase the dimension of the training data-set by creating new samples starting from the original ones and adding uniform noise. The idea was to try to overcome the lack of data. Unfortunately this set of experiments failed because prohibitive in terms of computational costs.
- Enlarge the data-set (adding 2 weeks in the training [slide-horizon and incremental]): With this set of experiments we tried to enlarge the data-set in "time" without data multiplication.

  - Heating: We have 6 week of training and one day (the next) for testing in the sliding-horizon data-set with totally 15 data-set (0 to 14).In the *incremental* data-set we have 6 week for training and the next for testing (data-set 0) then we add the next day in the train data-set and we test on the next (not in the training) day. And so on (until data-set 14). In this way, for the last test we have 8 weeks for training and 1 day of testing. The *slidinghorizon* data-set will be used to test if a classical FFNN performs better than with 4 weeks training (original sliding horizon) data-set. While the incremental data-set will be used to train a LSTM NN in order to see if the net improves with more data for training (in particular here we will compare the performance of a FFNN trained on the 26 features data-set, with the performance of a LSTM NN trained on the 11 features data-set. The goal is to check if the memory layers of the LSTM NN can substitute the absence of the 15 features (these features are indeed just past data).
  - Electricity: Similar to the heating data-set but here we have 4 weeks of training and the last 2 weeks for testing (2 weeks less of data, compared to the heating data-set)

ELECTRICITY Top 10 - 2 weeks dataset

	Season 1		Season 2		Season 3	
	kind of net :	rmse	kind of net :	rmse	kind of net :	rmse
1	"all_deseason_40"	"0.059021"	"small_deseason_70"	"0.11623"	"all_deseason_56"	"0.055899"
2	"all_deseason_72"	"0.059078"	"small_deseason_58"	"0.11646"	"all_deseason_80"	"0.055916"
3	"all_deseason_32"	"0.059283"	"small_deseason_82"	"0.11652"	"all_deseason_44"	"0.056047"
4	"all_deseason_44"	"0.059527"	"small_deseason_62"	"0.11675"	"all_deseason_40"	"0.056232"
5	"all_deseason_76"	"0.060081"	"small_deseason_66"	"0.11722"	"all_deseason_36"	"0.056552"
6	"all_deseason_68"	"0.060191"	"small_deseason_74"	"0.11753"	"all_deseason_76"	"0.056567"
7	"all_deseason_16"	"0.060398"	"small_deseason_78"	"0.11814"	"all_deseason_68"	"0.056705"
8	"all_deseason_80"	"0.060421"	"small_57"	"0.1183"	"all_deseason_60"	"0.057028"
9	"all_deseason_8"	"0.061021"	"small_61"	"0.12021"	"all_deseason_64"	"0.057055"
10	"all_deseason_36"	"0.061359"	"small_81"	"0.12337"	"all_deseason_72"	"0.057151"

Fig. 7. Electricity Top 10 on original data-set. Number from 1 to 56 stand for SGD Algorithm ; From 57 to 84 stand for ADAM Algorithm

HEATING Top 10 - 4 weeks dataset

	Season 1		Season 2		Season 3	
	kind of net	rmse	kind of net :	rmse	kind of net :	rmse
1	"all_55"	"0.1058"	"all_75"	"0.071589"	"all_deseason_84"	"0.059433"
2	"all_35"	"0.10647"	"all_43"	"0.07177"	"all_deseason_64"	"0.059747"
3	"all_71"	"0.10706"	"all_55"	"0.071785"	"all_deseason_60"	"0.060165"
4	"all_31"	"0.10754"	"all_3"	"0.072347"	"all_deseason_28"	"0.06032"
5	"all_59"	"0.10768"	"all_47"	"0.072429"	"all_deseason_72"	"0.06074"
6	"all_43"	"0.10825"	"all_35"	"0.072557"	"all_deseason_68"	"0.060958"
7	"all_63"	"0.10903"	"all_31"	"0.072701"	"all_deseason_80"	"0.06116"
8	"all_67"	"0.11012"	"all_67"	"0.073447"	"all_deseason_56"	"0.061252"
9	"all_39"	"0.1102"	"all_79"	"0.073576"	"all_deseason_36"	"0.061338"
10	"all_83"	"0.11024"	"all_51"	"0.073652"	"all_deseason_76"	"0.061376"

Fig. 8. Heating Top 10 on original data-set. Number from 1 to 56 stand for SGD Algorithm ; From 57 to 84 stand for ADAM Algorithm

D. Results on the original data-sets

We now present the best results we obtained. In particular the nets are classified according to their average RMSE (on the 28 data-sets) for each season.

a) Top 10 Electricity on the original data-set [Figure 7]: For all the seasons we can see that the trend-removal is quite effective. Indeed almost all the best performing nets are those that have been trained using the de-season data-set. Further investigation is necessary on the season 2. Apparently a data-set with less features (11 instead of 26) allows the net to perform better. Almost all the nets have been trained with the ADAM algorithm.

b) Top 10 Heating on the original data-set [Figure 8]: For all the seasons the "all-features" data-set allowed better performances. There is a significant difference of performance though, among the seasons; and only on season 3 the de-season preprocess has been helpful (season 3 is indeed a more clean data-set). In season 3 the ADAM algorithm has produced the best results. Almost all the best performing nets have been trained with the ADAM algorithm.

Overall we can conclude that the big feature data-set (26 features) allow to reach better performance(5 over 6 seasons). Different nets perform better on different data-set (with a preference for the ADAM algorithm).

E. Results on the enlarged data-sets [Figure 9 and Figure 10]

In all the cases the nets trained on the new (bigger) data-set perform worse than the same (best) nets trained on the smaller data-sets (sliding of 4 weeks for the heating and of 2 weeks for the electricity). In Particular the performances are quite similar for the best cases (season 1 and 3 of electricity and season 2 and 3 of heating) but they degraded much more on the other seasons.

ELECTRICITY - 4 weeks Increased dataset						
	Season 1		Season 2		Season 3	
	Kind of net	rmse	Kind of net	rmse	Kind of net	rmse
2 weeks Best	*"all_deseason_40"*	*"0.059021"*	*"small_deseason_70"*	*"0.11623"*	*"all_deseason_56"*	*"0.055899"*
1	FFNN_SH_3	"0.07161"	FFNN_I_8	"0.17826"	FFNN_SH_3	"0.054087"
2	FFNN_SH_4	"0.072802"	FFNN_SH_3	"0.18829"	FFNN_SH_2	"0.054209"
3	FFNN_SH_2	"0.073216"	FFNN_SH_2	"0.19272"	FFNN_SH_5	"0.056093"
4	FFNN_SH_5	"0.074144"	FFNN_SH_4	"0.19675"	FFNN_SH_4	"0.059436"
5	FFNN_I_8	"0.075395"	FFNN_SH_1	"0.19687"	FFNN_I_8	"0.061119"
6	FFNN_SH_1	"0.076003"	FFNN_SH_5	"0.20191"	FFNN_SH_1	"0.070659"
7	LSTM_I_6	"0.089365"	LSTM_I_6	"0.22048"	LSTM_I_6	"0.085532"
8	LSTM_I_7	"0.090354"	LSTM_I_7	"0.22324"	LSTM_I_7	"0.10065"

Fig. 9. Electricity on Increased data-set. SH: Sliding Horizon data-set, I: incremental data-set, RMSE: average of the daily error per season

HEATING - 6 weeks Increased dataset						
	Season 1		Season 2		Season 3	
	Kind of net	rmse	Kind of net	rmse	Kind of net	rmse
4 weeks Best	*"all_55"*	*"0.1058"*	*"all_75"*	*"0.071589"*	*"all_deseason_84"*	*"0.059433"*
1	FFNN_SH_1	"0.10814"	FFNN_SH_1	"0.087646"	FFNN_SH_1	"0.069542"
2	FFNN_I_8	"0.11132"	FFNN_SH_2	"0.088027"	FFNN_SH_4	"0.071043"
3	FFNN_SH_4	"0.11171"	FFNN_SH_3	"0.091076"	FFNN_SH_5	"0.073204"
4	FFNN_SH_5	"0.11209"	FFNN_SH_4	"0.091672"	FFNN_I_8	"0.075838"
5	FFNN_SH_3	"0.11351"	FFNN_SH_5	"0.091917"	FFNN_SH_2	"0.077845"
6	FFNN_SH_2	"0.11388"	FFNN_I_8	"0.099661"	FFNN_SH_3	"0.078878"
7	LSTM_I_7	"0.34203"	LSTM_I_6	"0.13229"	LSTM_I_7	"0.20192"
8	LSTM_I_6	"0.36638"	LSTM_I_7	"0.14767"	LSTM_I_6	"0.27224"

Fig. 10. Heating on Increased data-set. SH: Sliding Horizon data-set, I: incremental data-set, RMSE: average of the daily error per season

F. Considerations on the results

The worse performances obtained by the nets trained on the bigger data-set can be explained, by the fact that increasing the time horizon of the training we capture some seasonal components (trends) that cannot be anymore approximated by linear functions (or other very simple function). This trends could be probably learned if we had more training data for that particular season. Since we have few data, a less wide training window" allows us to implicitly approximate the seasonal component (typically sinusoidal) with a very simple function, that is easier to learn. As a proof of concept look at the Figures 11 and 12. These considerations are supported by the fact that a net trained on the incremental data-set does not always perform better (on average) than a net trained on the slide-horizon data-set.

For what concerns LSTM NNs, their performances are generally worse when compared to the simpler FFNNs. The main problem here is the beginning of the day. That means that the first hours of the test data-sets are predicted very badly by the LSTM nets. However the prediction accuracy generally increases a lot as we come close to the end of the testing data-set. On the other hand, the FF nets perform similarly throughout all the test data-set [no particular improvements (on average) for some hours of the testing day]. Figure 13 represents the performance of the nets trained on the bigger data-set. As we can see, at the beginning LSTM NNs perform very poorly.

G. Conclusions

From our experiments we can conclude that to train effectively a deep neural network a lot of data is necessary. If the training data is not clean enough or not enough in quantity,

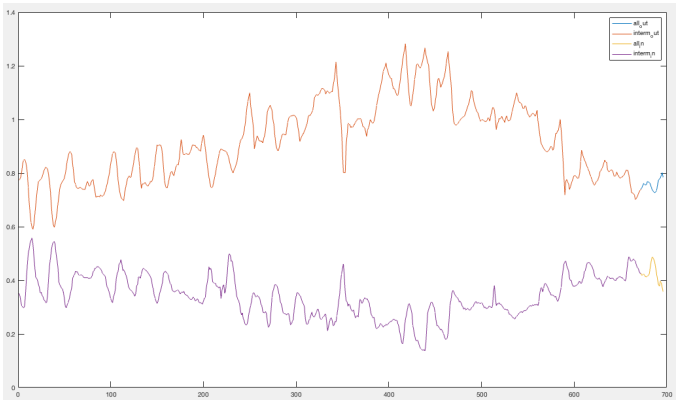

Fig. 11. 4 weeks of training on heating data-set. Simple trend to learn. season 3; training set 2; in orange is the input temperature and in purple the heating consumption

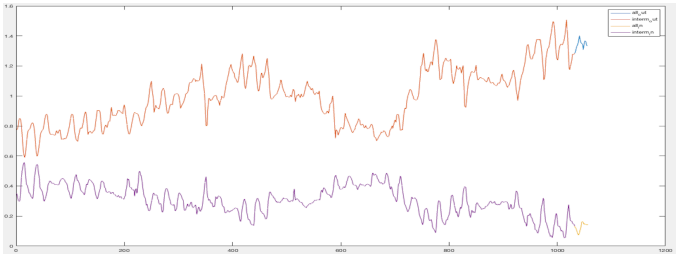

Fig. 12. 6 weeks of training on heating data-set. harder trend to learn. season 3; training set 2; in orange is the input temperature and in purple the heating consumption

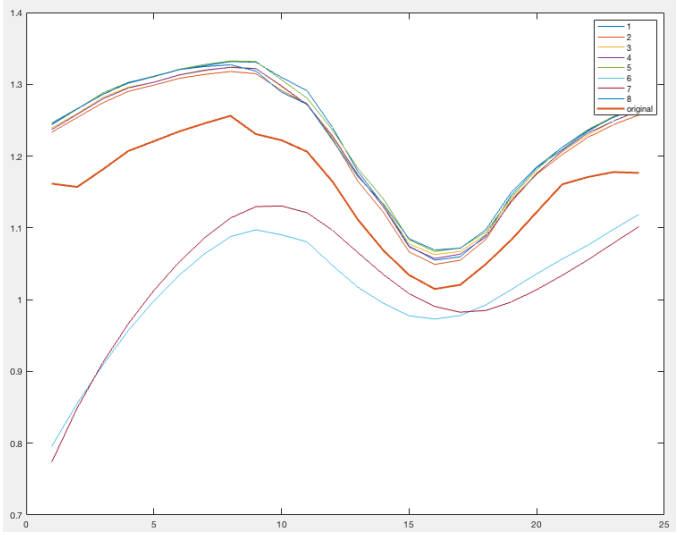

Fig. 13. Example of comparison between the performances of FFNN and LSTN on the heating data-set, season 3, data-set 14. The number from 1 to 5 represents FFNNs trained with sliding-horizon (6 weeks); number 6 and 7 represents LSTM trained on the incremental data-set (8 weeks); Number 8 represent a FFNN trained on the incremental data-set (8 weeks)

simpler models are preferable. It might be a good idea in general to preprocess the data-set by removing seasonal trends and add then back after the prediction occurred. In our experiments, we noticed that nets trained on smaller data-set perform better than on bigger data-sets. This might be due to seasonal components and "noise", that become harder to approximate when the train set is increased (in this case, when we increase the training data-set we are introducing annual (seasonal) components. To reduce this problem we should have data of the same season for many years). In our experiments we also compared fully LSTM networks with FF neural networks. In particular we wanted to check if the LSTM was able to learn the importance of previous (past) data [for instance, remember (and use) the data of the same day (of the test day) of the previous week]. Experiments showed us that FF NNs perform better both with less data and with more data. However, the performance of the LSTM nets increases as the training data increases. Another remarkable fact is that the LSTM perform quite badly during the first hours of the test set, and they prediction accuracy generally increases toward the end of the test day. On the other hand the FF nets exhibit more or less the same prediction accuracy through out all the day. For what concern training time and parameters: there is not a really noticeable difference when we consider different FF nets. Things change when we consider LSTM nets. In particular, in LSTM the training time is much large and the training parameter are much more critical (validation is suggested). It is important to test different networks structures and compare the performance. For instance, one might think that the best nets for a prediction problem (of a sequence) is a LSTM net. However, as we noticed in our experiment, there are many factors that contribute to the overall accuracy and there are many other factors (not directly connected to the accuracy; for instance the training time) that must be taken into consideration. In our experiments a net trained with more input features (where the additional features where just past data (used in order to introduce some sort of sequentially)) and smaller training data-set, performed better than more complex nets such as LSTM networks. And were trained in much less time.

REFERENCES

[1] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[2] https://it.mathworks.com/help/nnet/examples/time-series-forecasting-using-deep-learning.html

[3] https://it.mathworks.com/help/nnet/ug/improve-neural-network-generalization-and-avoid-overfitting.html

[4] https://it.mathworks.com/help/nnet/ug/ist-of-deep-learning-layers.html

[5] https://machinelearningmastery.com/improve-deep-learning-performance/

[6] https://it.mathworks.com/videos/developing-forecast-models-from-time-series-data-in-matlab-part-1-93067.html

[7] https://it.mathworks.com/videos/developing-forecast-models-from-time-series-data-in-matlab-part-2-93066.html

[8] Diederik P. Kingma, Jimmy Lei Ba, ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

[9] Grippo, Manno, Sciandrone, Decomposition Techniques for Multilayer Perceptron Training

[10] LeCun, Bengio, Hinton, Deep learning

[11] Glorot, Bengio, Understanding the difficulty of training deep feedforward neural networks

[12] Bottou, Large-Scale Machine Learning with Stochastic Gradient Descent