INFO - H410
TECHNIQUES OF ARTIFICIAL INTELLIGENCE

Duynslaeger Sacha (000574340),
Mola Bradon Emmanuel (000557160),
Vertu Ndinga Oba Olivier (000575921)

- PROMOTOR -
Professor Hugues BERSINI

24 mai 2023

Table des matières

1 Introduction and Context

This project is willing to be a part of the project "Voisin d'Énergie" created at the ULB. The ambition of "Voisin d'Énergie" is to constitute a network of energy communities within the Brussels region. The goal is that these communities would be as resilient as possible and do not depend of the global energy network. This would be done by sharing the energy produced by some of the communities's homes (for example with photovoltaic pannels).
By being part of this project, our objective is to develop an A.I model that would be able to predict the communities homes energy consumption in order to help the planning of the energy sharing.

Such prediction can be made by using a neural networks, and more specifically by using a LSTM (Long Short Term Memory) model. Currently, there exist several well implemented and documented libraries that could do the job in only a few lines of code (for example Keras). But since our main goal is to learn, we decided to implement our own model from scratch and to compare it with the Keras model.

2 Description of the model

Our model is a LSTM recursive neural network using the MSE as error measurement. We implemented our model in a way that allows the user to choose between the Adam Optimizer and the stochastic gradient descent as optimizer, with the Adam Optimizer being set as the default one.

The understanding from scratch of times series forecasting with LSTM networks is a complex topic that requires a good knowledge of the theory behind the LSTM cell and about neural networks in general. In order to deal with the time we had at our disposition, we decided to focus on the building block of the LSTM networks : the LSTM cell. The model we implemented therefore consists of an input layer, a single LSTM layer (which can be seen as the recursion of a single LSTM cell) and finally an output layer.

2.1 LSTM overview

Since our objective is to predict energy consumption, Recursive Neural Networks (RNN) models are well suited because they can deal with sequential data. The problem with classical RNNs is that they suffer from the gradient vanishing problem[1], that we will not explain in detail in this paper for the sake of conciseness. A very good solution that can process long sequential data and that do not suffer from the vanishing gradient problem are the use of the LSTM (Long Short Term Memory) cells, a specific type of RNNs.

Their architecture is based on two memories, a short-term one that is used to detect complex patterns in recent sequential data and a long-term one that keeps track of the old seen data.

As all RNNs models, the LSTM model works iteratively. Each iteration consists of a new input analyzed by the model. At each iteration, the model computes a forward propagation to update the long and short term memory, a backward propagation to computes the gradient and then uses it to update the parameters of the cell by using an optimization algorithm. In our case, chose to work with the Adam Optimizer because it gave us the best results.

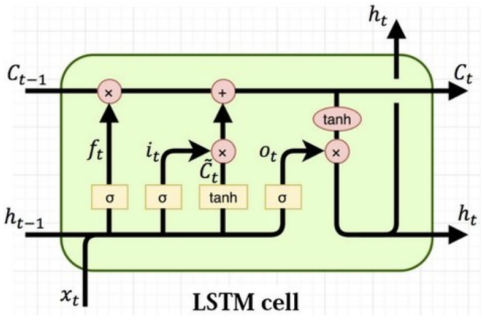


FIGURE 1 – Source : https://www.researchgate.net/figure/Structure-of-the-LSTM-cell-and-equations-that-describe-the-gates-of-an-LSTM-cell_fig5_329362532

2.2 Terminology and notations

When talking about the parameters of a LSTM cell, several terms might refer to the same thing. The hidden state is another term to talk about the short-term memory, and the cell state is another term to talk about the long-term memory. This is why the letter "h" is used when referring to the short-term memory and the letter "C" for the long-term memory.
Within this paper, we might use the two versions of each interchangeably, but we will use the "short-term memory" and "long-term memory" appellation most of the time as they appear more intuitive.

Before entering into the details, we will take some time to explain the notations of the parameters of the cell. The indices of time represent the iteration step. Index t refers to the current iteration, while $t-1$ refers to the last iteration.
For the weights and biases, the index "f" means the weight or bias is associated with the forget gate (the gates will be explained in the next section), "i" for the input gate, "$\tilde{C}$" for the candidate, and "o" for the output gate. As second index of the weights, "x" refers to the input, and "h" to the short-term memory.
For example, w_{fx} is the weight applied to the input value within the forget gate.
The symbol σ used in the equations refers to the sigmoid function.

2.3 Explanation and implementation of the model

Before the first iteration, every weights of the cell are randomly initialized with the Xavier Initialization [4], and every biases and element of the gradient are initialized to 0.

2.3.1 Forward Propagation

The objective of the Forward Propagation is, given the short-term memory h_{t-1} and the long-term memory C_{t-1} computed in iteration $t-1$, and given an input x_t received at iteration t, to update the long-term and short-term memory. To understand how it is done, we can cut the cell into 3 pieces that we will call the forget gate, input gate and output gate, and analyze what they do. The following figure shows in depth what is done within a forward propagation :

The role of the forget get is to compute the percentage of the long-term memory we want to remember. This quantity is called f_t and is computed as following :

$$f_t = \sigma(x_t \cdot w_{fx} + h_{t-1} \cdot w_{fh} + b_f) \tag{1}$$

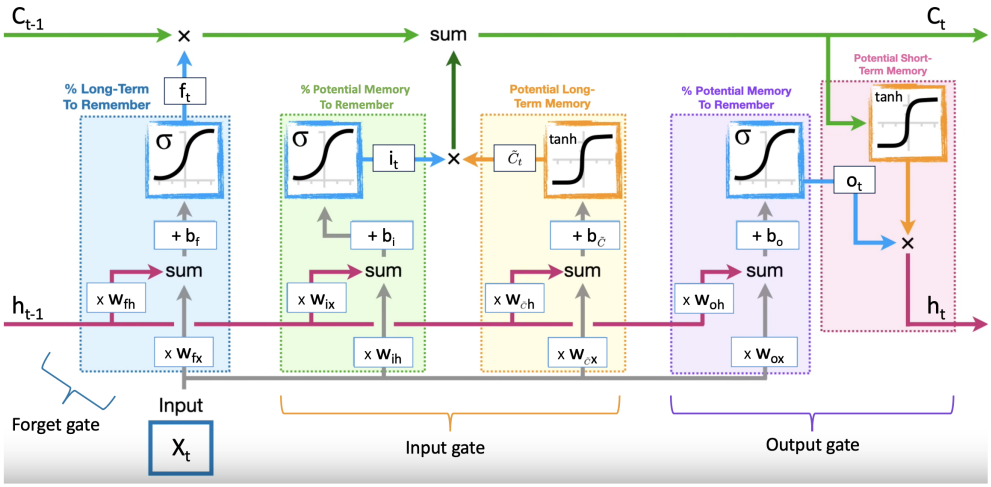


FIGURE 2 – Forward propagation of a LSTM cell. Modified version of the figure found in the LSTM Tutorial from StatQuest[2]

The role of the input gate is to update the long-term memory. It does so by computing the new potential long-term memory to remember, denoted $\tilde{C}_t$, and the percentage of that new potential long-term memory, denoted i_t. These quantity are computed as follow :

$$i_t = \sigma(x_t \cdot w_{ix} + h_{t-1} \cdot w_{ih} + b_i) \tag{2}$$

$$\tilde{C}_t = \sigma(x_t \cdot w_{\tilde{C}x} + h_{t-1} \cdot w_{\tilde{C}h} + b_{\tilde{C}}) \tag{3}$$

Once the forget and input gate have done their job, we can update long-term memory, denoted C_t, such as :

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{4}$$

Then, the output gate computes the new potential short-term memory and the percentage of that potential short-term memory to remember (the percentage is denoted o_t) and updates the short-term memory such as :

$$o_t = \sigma(x_t \cdot w_{ox} + h_{t-1} \cdot w_{oh} + b_o) \tag{5}$$

$$h_t = tanh(C_t) \cdot o_t \tag{6}$$

Once all of these calculations have been done, the forward propagation is finished and the updated short-term memory h_t and long-term memory C_t are returned, ready to be used for the next iteration.

Implementation :

Every step concerning the forward propagation is done within the `forward()` function. The code was written in order to be as similar to the equations as possible.
Note : for the manipulation of weights, we decided to work with vectors for the sake of better code readability. For example, the line `np.dot(self.W_gates["input"], concat)` performs the sum : $\begin{bmatrix} w_{ix} & w_{ih} \end{bmatrix} \cdot \begin{bmatrix} x_t & h_{t-1} \end{bmatrix}^T = x_t \cdot w_{ix} + h_{t-1} \cdot w_{ih}$

2.3.2 Backward Propagation

The objective of the backward propagation is to calculate the gradient of the error, composed of the derivative of the error with respect to each bias and each weight of the LSTM cell. Since the error do not depend directly on the biases and weights, the chain rule must be applied in order to get the results. All the details of the calculations are shown in the Annexes 5.1. The computed gradient is :

$$\nabla = \left[\frac{\partial E}{\partial w_{fh}} \quad \frac{\partial E}{\partial w_{fx}} \quad \frac{\partial E}{\partial b_f} \quad \frac{\partial E}{\partial w_{ih}} \quad \frac{\partial E}{\partial w_{ix}} \quad \frac{\partial E}{\partial b_i} \quad \frac{\partial E}{\partial w_{\tilde{c}h}} \quad \frac{\partial E}{\partial w_{\tilde{c}x}} \quad \frac{\partial E}{\partial b_{\tilde{c}}} \quad \frac{\partial E}{\partial w_{oh}} \quad \frac{\partial E}{\partial w_{ox}} \quad \frac{\partial E}{\partial b_o} \right]^T$$

With the error E being the MSE :

$$E = (h_t - y_{expected})^2$$

Once the gradient has been computed, it is used by the optimizer to update the weights and biases of the cell.

Implementation :

Every step concerning the backward propagation is done within the `backward()` function.
As before, the code was written in order to be as similar to the equations as possible, and the weights manipulation are made with vectors, as in `forward()`.

2.3.3 Updating the parameters

The update of the parameters is done through Adam optimizer. The optimizer works as follow[3] :

First we need to define 4 parameters : α, β_1, β_2 and ϵ. The default value we chose for them are alpha=0.001, beta1=0.09, beta2=0.999 and epsilon=1e-8, because these are the values found in the literature.
Then, for each parameter θ to be optimized (for each element of the gradient computed in the backpropagation, section 2.3.2), we compute :

$$\begin{aligned} g_t &= \nabla(\theta_{t-1}) \\ m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ \hat{m}_t &= \frac{m_t}{(1 - \beta_1^t)} \\ \hat{v}_t &= \frac{v_t}{(1 - \beta_2^t)} \end{aligned}$$

And finally, we update the parameter θ :

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Implementation :

Every step concerning the update of parameters is done within the `LSTM.update()` function, that calls the `AdamOptimizer.update()` function.

2.4 Data pre-processing

The pre-processing data is responsible for preparing the energy consumption data obtained from the "Voisin d'Énergie" database for training the model. The data consists of files that contain power consumption readings recorded at intervals of 8 seconds or every 15 minutes over the past months. To ensure more accurate results, we chose to work with the data collected every 8 seconds.

Before initiating the training phase, we performed several preprocessing steps on the data. Firstly, we removed the days where a significant amount of data was missing to ensure data integrity and reliability. This step helps in maintaining the quality of the dataset.

Implementation :

Every step concerning the data pre-processing is done within the `preprocess_data()` function.

2.5 Training the model

Before training the model, we need to specify two hyperparameters : the sequence length and the prediction size. The sequence length refers to the number of power consumption measurements we take for predicting a number of power consumption values equal to the prediction size.
If the sequence length is 5 and the prediction size 1, the model will feed upon the last 5 values to predict the next one.

The code used for the training is shown on the Annexes, section 5.2. Each step is explained in the comments.

3 Results and discussion

There are several ways to do the training. Here we use the forward function to train our model. You can find more details about this in annexe 5.2.

The results of the LSTM model training are encouraging. The training and validation loss curves showed a significant decrease over time. At the beginning of training, the training loss was about 0.001, while the validation loss was about 6.5 e-05 and 4.61 e-05. After 10 epochs, the learning loss dropped to about 0.0002 and 0.0006, with a validation loss of about 3.65 e-06 and 2.83 e-06. At epoch 20, the training loss continued to decrease to approximately 0.00019 and 0.0006, with a validation loss of approximately 3.28 e-06 and 3.12 e-06. These lower loss values indicate an improvement in the model's ability to fit the data and accurately predict the target values.

The total training time for the LSTM model was 21.96 seconds, showing a relatively fast computational efficiency to achieve these results. After training, predictions were made on the test set. The predictions were compared to the actual values, showing that the model was able to capture trends and patterns in the data. The plots of the predictions on the test set showed a close correlation between the model predictions and the actual target values, suggesting good performance of the model in predicting the temporal data.

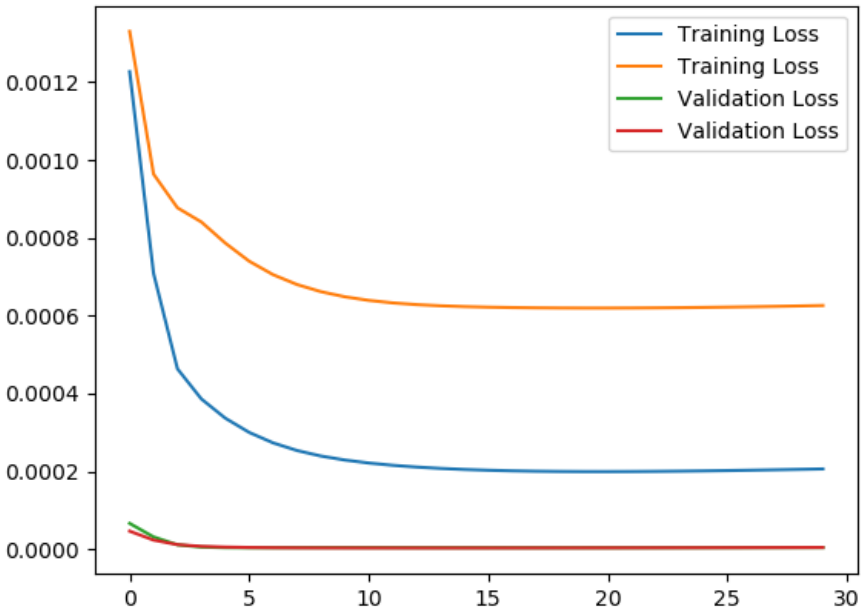


FIGURE 3 – Plot of the predictions against the actual values with CDB001, $seq\_len = 2$, $pred\_size = 2$[2]

We have two training curves from two validation curves because we are predicting two values each time and we decided to leave the graphs separate to observe the trend of both.

The results of evaluating the predictions against the actual values for the LSTM model are exceptional, as there is very little difference between the predictions and the target values. When we plot the predictions against the actual values on the graph, we can observe a close correlation and almost an overlap of the two curves. This indicates that the model was able to accurately capture the patterns and trends in the data. The closeness of the predictions to the target values suggests that the LSTM model has acquired a high capacity for generalization and accurate prediction. These outstanding results highlight the effectiveness and performance of the LSTM model in modeling and predicting temporal data.

The vertical axis represents the values predicted by 16 seconds given that we predict 2 values for each sequence of 2. And the horizontal axis corresponds to the normalized current consumption.

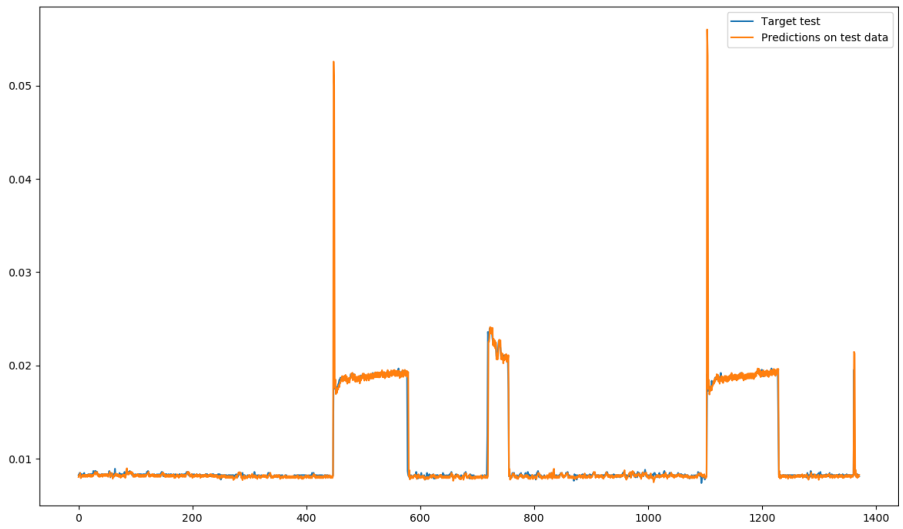
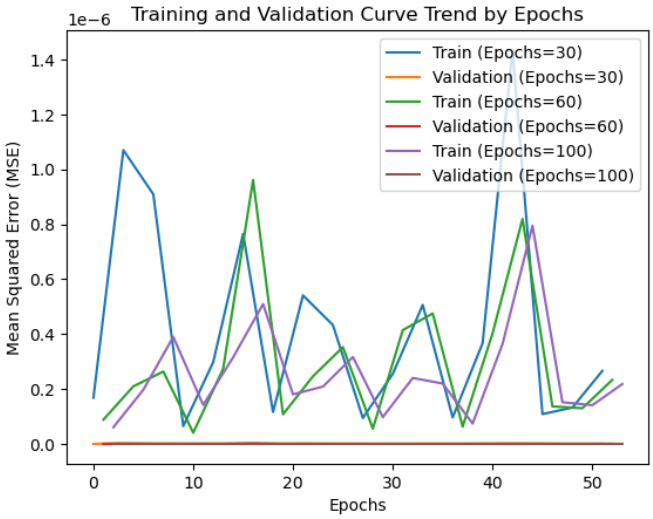


FIGURE 4 – Plot of the training and validation loss curves with CDB001, $seq\_len = 2$, $pred\_size = 2$[2]

3.1 Underfitting and Overfitting



By analyzing the importance of different parameters in the main Jupyter notebook we understand the behaviour of the model and detect the problem : the overfitting.The main cause of our overfitting is that the number of learning epochs, which represents the number of times the model iteratively learns from the data, becomes too high. Initially, as the model trains, it learns patterns and relationships in the training data, which gradually improves its performance. However, if training continues for an excessive number of epochs, the model may start to memorize the training data instead of learning general patterns. It then becomes too sensitive to specific data points in the training set and loses its ability to generalize. The consequence of overfitting is that the performance of the model on unseen data or on the validation/test set begins to degrade. The model may start to make more and more errors, indicating that it is no longer able to accurately predict data that it has never seen before. This defeats the purpose of using machine learning models to make predictions in real-world scenarios where new data points are encountered.

4 Conclusion and future work

Our LSTM model accurately predicts short-term energy consumption and production in homes by learning temporal patterns from real data. Learning and validation losses decrease significantly during learning, indicating improved model performance. Evaluation of the model on the test set confirms its predictive capabilities, with close alignment between predictions and actual values. However, overfitting can occur if the number of epochs is too high, leading to memorization of training data and decreased accuracy on unseen data. To improve the model, we can consider incorporating multi-LSTM layers and implementing minibatches for more efficiency. Overfitting and refining long-term predictions remain ongoing challenges, requiring further exploration. Concerning predicting long-term energy consumption poses challenges, particularly relying on the model's own predictions as input. Although attempts were made to address this, the obtained results in the areas 5 were unsatisfactory Overall, our LSTM model shows promising accuracy and the ability to learn from temporal patterns, highlighting its potential for predicting short-term energy consumption in residential environments.

5 Annexes

5.1 Detailed calculations of the gradient

As reminder, because they will be used in the chain rule, the results obtained in the forward propagation are the following :

$$
\begin{aligned}
f_t &= \sigma(x_t \times w_{fx} + h_{t-1} \times w_{fh} + b_f) = \sigma(z_f) \\
i_t &= \sigma(x_t \times w_{ix} + h_{t-1} \times w_{ih} + b_i) = \sigma(z_i) \\
\tilde{C}_t &= \sigma(x_t \times w_{\tilde{C}x} + h_{t-1} \times w_{\tilde{C}h} + b_{\tilde{C}}) = \sigma(z_{\tilde{C}}) \\
o_t &= \sigma(x_t \times w_{ox} + h_{t-1} \times w_{oh} + b_o) = \sigma(z_o) \\
C_t &= f_t \times C_{t-1} + i_t \times \tilde{C}_t \\
h_t &= tanh(C_t) \times o_t
\end{aligned}
$$

First, let's define the error E such as the mean square error :

$$E = (h_t - y)^2 \tag{7}$$

with h_t being the output value of the cell (i.e the value of the short-term memory, also called the hidden state, at time t), and y being the expected value.

Note : in order to obtain more readable equations, we let the derivative of the sigmoid function as $\sigma^{'}(x)$ instead of writing the whole $\sigma^{'}(x) = \sigma(x) \cdot (1 - \sigma(x)))$

The derivation of the error with respect to the weights and biases composing the gradient are the following :

$$
\begin{aligned}
\frac{\partial E}{\partial w_{fh}} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial w_{fh}} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot C_{t-1} \cdot h_{t-1} \cdot \sigma^{'}(z_f) \\
\frac{\partial E}{\partial w_{fx}} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial w_{fx}} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot C_{t-1} \cdot x_t \cdot \sigma^{'}(z_f) \\
\frac{\partial E}{\partial b_f} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial f_t}\frac{\partial f_t}{\partial b_f} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot C_{t-1} \cdot \sigma^{'}(z_f)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ih}} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial w_{ih}} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot \tilde{C}_t \cdot h_{t-1} \cdot \sigma^{'}(z_i) \\
\frac{\partial E}{\partial w_{ix}} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial w_{ix}} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot \tilde{C}_t \cdot x_t \cdot \sigma^{'}(z_i) \\
\frac{\partial E}{\partial b_i} &= \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial i_t}\frac{\partial i_t}{\partial b_i} = 2(h_t - y) \cdot o_t \cdot (1 - tanh^2(C_t)) \cdot \tilde{C}_t \cdot \sigma^{'}(z_i)
\end{aligned}
$$

$$\frac{\partial E}{\partial w_{\tilde{c}h}} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial \tilde{C}_t}\frac{\partial \tilde{C}_t}{\partial w_{\tilde{c}h}} = 2(h_t - y)\cdot o_t \cdot (1 - tanh^2(C_t))\cdot i_t \cdot h_{t-1} \cdot (1 - tanh^2(z_{\tilde{c}}))$$
$$\frac{\partial E}{\partial w_{\tilde{c}x}} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial \tilde{C}_t}\frac{\partial \tilde{C}_t}{\partial w_{\tilde{c}x}} = 2(h_t - y)\cdot o_t \cdot (1 - tanh^2(C_t))\cdot i_t \cdot x_t \cdot (1 - tanh^2(z_{\tilde{c}}))$$
$$\frac{\partial E}{\partial b_{\tilde{c}}} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial C_t}\frac{\partial C_t}{\partial \tilde{C}_t}\frac{\partial \tilde{C}_t}{\partial b_{\tilde{c}}} = 2(h_t - y)\cdot o_t \cdot (1 - tanh^2(C_t))\cdot i_t \cdot (1 - tanh^2(z_{\tilde{c}}))$$

$$\frac{\partial E}{\partial w_{oh}} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial w_{oh}} = 2(h_t - y)\cdot tanh(C_t) \cdot h_{t-1} \cdot \sigma^{'}(z_o)$$
$$\frac{\partial E}{\partial w_{ox}} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial w_{ox}} = 2(h_t - y)\cdot tanh(C_t) \cdot x_t \cdot \sigma^{'}(z_o)$$
$$\frac{\partial E}{\partial b_o} = \frac{\partial E}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial b_o} = 2(h_t - y)\cdot tanh(C_t) \cdot \sigma^{'}(z_o)$$

5.2 Training code

Here is the codes of the training with comments explaining what every step does (we omitted some lines not useful for the training for better readability) :

```
def train_lstm_forward(lstm, input_train, target_train, input_test, target_test
                                          , num_epochs, learning_rate,
                                          perform_predictions=True):

    for epoch in range(num_epochs):
        predictions = []
        for s in range(sequence_length):
            predictions.append(np.array(data_train[s]))

        for i in range(0, len(data_train) - sequence_length - predict_size,
                                                predict_size):
            y_t = data_train[i] # Get the target for this iteration

            for j in range(sequence_length):
                # Reset the gradient to prevent the error to propagate
                lstm.reset()

                # Select the data that will feed the lstm cell
                x_t = predictions[i+j]

                # Forward propagation
                cache = lstm.forward(x_t)

                # Compute the derivative of the loss, i.e the MSE
                dloss = 2 * (lstm.h_t - y_t)

                # Backward propagation
                lstm.backward(dloss, x_t, cache)

                # Update the weights
                lstm.update(learning_rate, lstm.optimizer)

            for s in range(predict_size):
                # Once the lstm has been fed, append the predicted values
                predictions.append(lstm.h_t[s])

    return predictions
```

```
def train_lstm_reverse(lstm, input_train, target_train, input_test, target_test
                                        , num_epochs, learning_rate,
                                        perform_predictions=True):

    for epoch in range(num_epochs):
        train_loss = 0.0
        val_loss = 0.0

        for i in range(len(input_train)):
            # Get the input and target for this iteration
            y_t = target_train[i]

            cache = []
            dloss = []
            for j in range(sequence_length):
                # Reset the gradient to prevent the error to propagate
                lstm.reset()
                x_t = input_train[i][j]

                # Forward pass
                cache.append(lstm.forward(x_t))
                # Compute the loss and its gradient MSE
                dloss.append(2 * (lstm.h_t - y_t))

            # Retro back propagation
            for j in reversed(range(sequence_length)):
                lstm.reset()
                x_t = input_train[i][j]
                lstm.backward(dloss[j], x_t, cache[j])

                # Update the weights
                lstm.update(learning_rate, lstm.optimizer)

            loss = (lstm.h_t - y_t) ** 2
            train_loss += loss

    return predictions
```

The train lstm retro and train lstm forward functions are both used to train the LSTM model, but they differ in the way they handle time sequences.

The train lstm retro function performs backpropagation in time (backpropagation) when training the model. It iterates over each temporal sequence in the training set and performs a forward pass for each instant in the sequence. The intermediate outputs of each instant are stored in a cache list and the loss gradients are also calculated for each instant. Then, it performs a backward pass using the gradients and the stored intermediate outputs, which updates the model weights. This back-pass approach can be useful when time dependencies are important to the model.

In contrast, the train lstm forward function takes a simpler approach : it performs a forward pass for each time in the sequence and updates the weights after each time. It does not use backpropagation in time to compute the gradients over the entire sequence. This method can be effective when the temporal dependencies are less important or when the model is able to capture short-term patterns.

In summary, the main difference between these two functions is how they handle temporal sequences when training the LSTM model. The train lstm retro function uses backpropagation in time to account for temporal dependencies over the entire sequence, while the train lstm forward function focuses more on short-term patterns by performing forward runs and weight updates for each time point individually. The choice between these two approaches depends on the characteristics of the data and the specific goals of the model.

5.3 Results of the long-term prediction

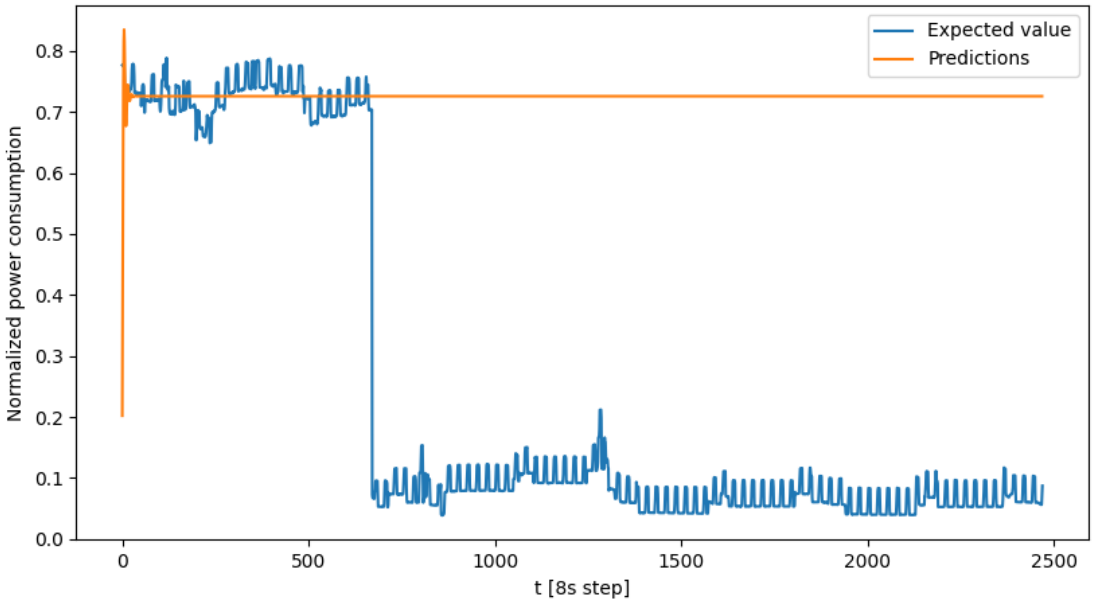


FIGURE 5 – Long-term prediction on CDB002, $seq\_len = 5$, $pred\_size = 1$, trained on 10 batches

It looks like the model doesn't learn anything about the variations of the values. It sticks on a value around 0.72. We can conclude that our implementation is clearly not adapted for long-term prediction.
The command to launch the long-term prediction model is exactly the same as the short-term one, excepted that the files are called "long_term_train.py" and "long_term_pred.py". Everything can be found in the "long_term_prediction" folder.

Références

[1] SuperDataScience Team - *Recurrent Neural Networks (RNN) - The Vanishing Gradient Problem*, 2018, https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-the-vanishing-gradient-problem/

[2] StatQuest with Josh Starmer - *Long Short-Term Memory (LSTM) Clearly Explained*, 2022, https://www.youtube.com/watch?v=YCzL96nL7j0&t=819s

[3] Layan Alabdullatef - *Complete Guide to Adam Optimization*, 2020, https://towardsdatascience.com/complete-guide-to-adam-optimization-1e5f29532c3d

[4] The 365 Team - *What Is Xavier Initialization ?*, 2021, https://365datascience.com/tutorials/machine-learning-tutorials/what-is-xavier-initialization/