Rețele Neuronale LSTM pentru serii de timp

Duzi Mihai-Nicolae, 352
Lumînăraru Ionuţ-Andrei, 342

Februarie 2026

Abstract

Această lucrare analizează diverse modele de inteligență artificială pentru serii de timp, cu scopul prezicerii consumului de curent electric al unei locuinţe. Pentru testarea modelelor, am folosit setul de date Individual Household Electric Power Consumption. Sunt prezentate și comparate modele auto-regresive, printre care și modele din sfera învățării adânci precum rețelele neuronale LSTM și transformerii.

Cuprins

1. Setul de date

Pentru analiza modelelor de mai jos, am folosit setul de date Individual Household Electric Power Consumption. Acesta este o serie multiplă de timp unde în fiecare minut au fost măsurate următoarele:

a. Global Active Power

b. Global Reactive Power

c. Voltage

d. Global Intensity

e. Submetering 1: Bucătărie (Mașină de spălat vase, microunde, cuptor)

f. Submetering 2: Camera de spălat rufe

g. Submetering 3: Încălzitor de apă și aer condiționat.

Deoarece setul inițial conținea peste două milioane de eșantioane, am decis să facem o operație de downsampling, luând măsurătorile din oră în oră, astfel reducând setul de date la aproximativ 35000 de eșantioane. Pentru fiecare oră am luat media celor 60 de minute, cu excepția intensității, unde am luat valoarea maximă.

2. Rețele Neuronale Recurente (RNN)

2.1 Ideea din spate

Rețelele neuronale sunt printre cele mai puternice modele din învățarea automată. Însă, rețelele neuronale clasice au o problemă majoră, anume lipsa de memorie. Cu o rețea clasică nu te poți folosi de date din trecut pentru a prezice datele din viitor, cum ar fi consumul de curent al unei case. Nu putem spune același lucru și despre rețelele neuronale recurente, care pot ține în memorie informații despre valorile eșantioanelor trecute, pentru a prezice valorile din viitor. Mai jos avem schema unei rețele neuronale recurente, respectiv a unei rețele neuronale recurente „desfăcute".

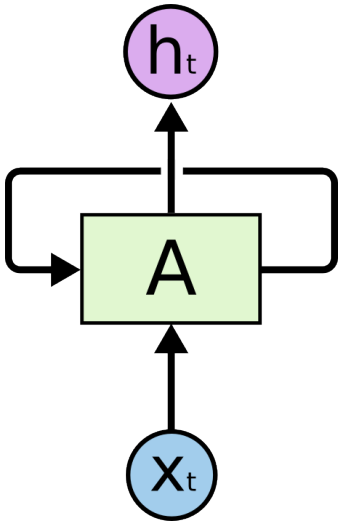


Figura 2.1: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

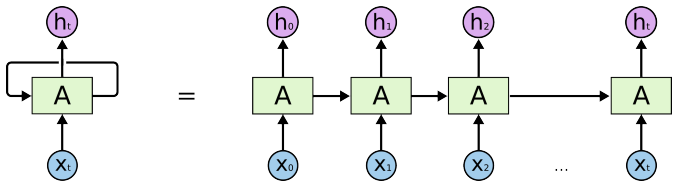


Figura 2.2: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

2.2 Cum funcționează modelul?

Avem o serie de timp pe care o notăm cu x. Aceasta are n elemente, iar noi vrem să-i prezicem următorul element. Pentru simplitate, parametrii din acest exemplu vor fi scalari. Notăm cu:

- w_1 = ponderea elementului curent
- b_1 = bias-ul elementului curent
- w_2 = ponderea memoriei anterioare
- w_3 = ponderea finală
- b_3 = bias-ul final

Fie şirul k, unde:

$$k_1 = \text{ReLU}(x_1 w_1 + b_1)$$

$$k_{i+1} = \text{ReLU}(x_{i+1} w_1 + k_i w_2 + b_1), \quad 1 < i \leq n.$$

Predicția finală pentru următorul element va fi $w_3 k_n + b_3$. Mai jos avem un exemplu pentru seria de timp [1, 0.8, 0.6], $w_1 = 0.5, w_2 = -1, w_3 = 0.5, b_1 = 0.5, b_3 = 0$

- $k_1 = \text{ReLU}(1 \cdot 0.5 + 0.5) = 1$
- $k_2 = \text{ReLU}(0.8 \cdot 0.5 + 1 \cdot (-1) + 0.5) = 0$
- $k_3 = \text{ReLU}(0.6 \cdot 0.5 + 0 \cdot (-1) + 0.5) = 0.8$
- $\hat{x}_4 = 0.5 \cdot 0.8 + 0 = 0.4$

2.3 Problema gradientului

Rețelele neuronale recurente stau la baza modelelor de învăţare adâncă precum LSTM sau Transformeri. Însă, ele nu reușesc să rețină informații despre valorile din trecutul îndepărtat din cauza problemei gradientului. Întrucât la fiecare termen nou calculat folosim aceleași ponderi, după mai multe operații vom ajunge la una din cele 2 probleme:

a. Dispariția gradientului, care are loc atunci când w_1 este în intervalul (-1, 1). Din cauza înmulţirii repetate cu valori subunitare, gradientul va ajunge să fie 0.

b. Explozia gradientului, care are loc atunci când w_1 este în afara intervalului (-1, 1). Din cauza înmulţirii repetate cu valori supraunitare, gradientul crește exponențial, iar la back-propagare se fac schimbări foarte mari care fac imposibilă convergența modelului.

3. Rețele Neuronale LSTM

3.1 Ideea din spate

După cum am menționat anterior, rețelele neuronale recurente nu reușesc să țină în memorie prea multe eșantioane din cauza problemei gradientului. Această problemă nu există însă și la rețelele neuronale LSTM, sau Long Short-Term Memory, care se folosesc de două tipuri de memorie:

a. Memoria pe termen scurt (Short-Term) - aceeași cu cea din RNN

b. Memoria pe termen lung (Long-Term) - poate reține informații pe termen lung pe care RNN-ul nu le putea reține

Mai jos avem schema unei rețele neuronale LSTM:

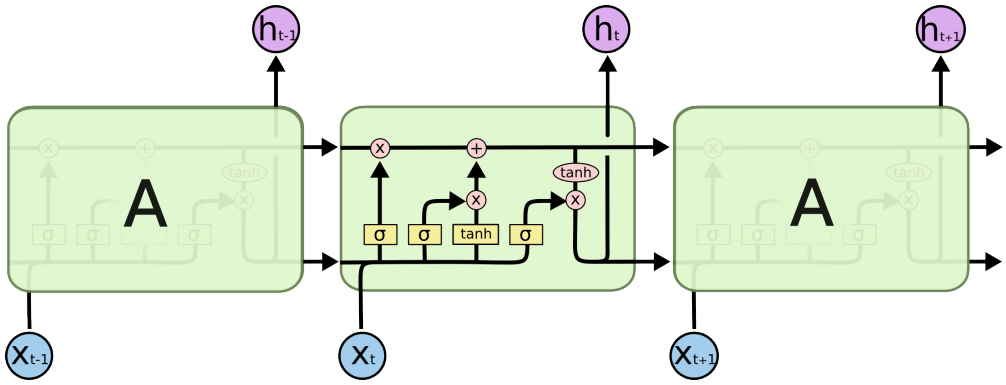


Figura 3.1: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

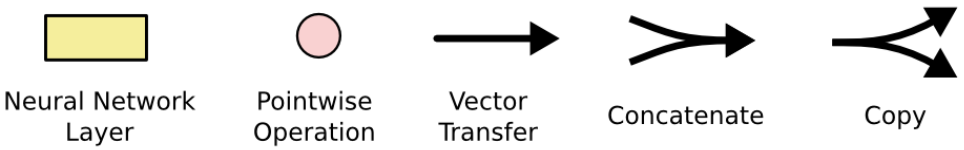


Figura 3.2: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

3.2 Cum funcționează modelul?

Notăm cu C și h memoria pe termen lung, respectiv scurt și le inițializăm cu 0. De asemenea, avem seria de timp x cu n elemente, iar noi vrem să prezicem următorul element. De asemenea, avem următoarele ponderi: W_f-Forget Gate, W_i - Input Gate, W_c Candidate Gate, W_o - Output Gate. Respectiv, bias-urile lor: b_f, b_i, b_c, b_o.

Primul pas: Forget Gate În acest pas decidem câte procente din L păstrăm folosind formula:

$$f = \text{sigmoid}(W_f \cdot [x, h] + b_f). \tag{3.1}$$

Al doilea pas: Input Gate În acest pas decidem câte procente din memoria nouă vom adăuga folosind formula:

$$i = \text{sigmoid}(W_i \cdot [x, h] + b_i). \tag{3.2}$$

Al treilea pas: Candidate Gate În acest pas calculăm noua memorie pe termen lung care trebuie adăugată folosind formula:

$$C_{nou} = \tanh(W_c \cdot [x, h] + b_c). \tag{3.3}$$

Iar mai apoi actualizăm memoria pe termen lung:

$$C = C_{vechi} \cdot f + C_{nou} \cdot i. \tag{3.4}$$

Ultimul pas: Actualizarea memoriei pe termen scurt Mai întâi calculăm câte procente din memoria de termen lung vom pune:

$$o = \text{sigmoid}(W_o \cdot [x, h] + b_o). \tag{3.5}$$

Apoi calculăm noua memorie de termen scurt:

$$h_{nou} = o \cdot \tanh(C) \tag{3.6}$$

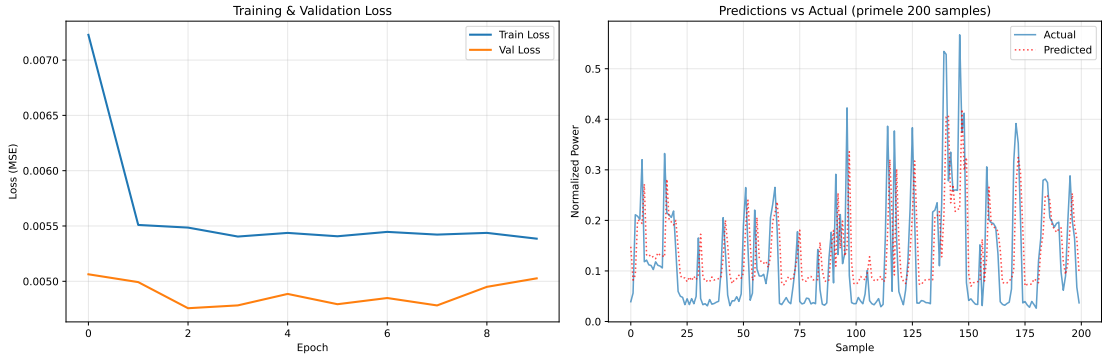
Repetăm procedeul folosind noua memorie și parcurgând seria de timp. Când ajungem la finalul ei, predicția va fi dată de valoarea sau de valorile din h_{nou}. Observăm că $\frac{\partial C}{\partial C_{vechi}} = f$, și deoarece f este o valoare între 0 şi 1 riscul de a apărea problema gradientului este mult mai scăzut.

Mai jos avem implementarea în Python a forward-ului din LSTM:

```
        self.hidden_size = hidden_size
        self.dropout = nn.Dropout(dropout)

        self.W_f = nn.Linear(input_size, hidden_size)
        self.W_i = nn.Linear(input_size, hidden_size)
        self.W_c = nn.Linear(input_size, hidden_size)
        self.W_o = nn.Linear(input_size, hidden_size)

        self.U_f = nn.Linear(hidden_size, hidden_size, bias=False)
        self.U_i = nn.Linear(hidden_size, hidden_size, bias=False)
        self.U_c = nn.Linear(hidden_size, hidden_size, bias=False)
        self.U_o = nn.Linear(hidden_size, hidden_size, bias=False)

    def forward(self, x_t, h_prev, C_prev):
        f_t = torch.sigmoid(self.W_f(x_t) + self.U_f(h_prev))
        i_t = torch.sigmoid(self.W_i(x_t) + self.U_i(h_prev))
        C_tilde = torch.tanh(self.W_c(x_t) + self.U_c(h_prev))

        C_t = f_t * C_prev + i_t * C_tilde

        o_t = torch.sigmoid(self.W_o(x_t) + self.U_o(h_prev))
        h_t = o_t * torch.tanh(C_t)

        h_t = self.dropout(h_t)

        return h_t, C_t
```

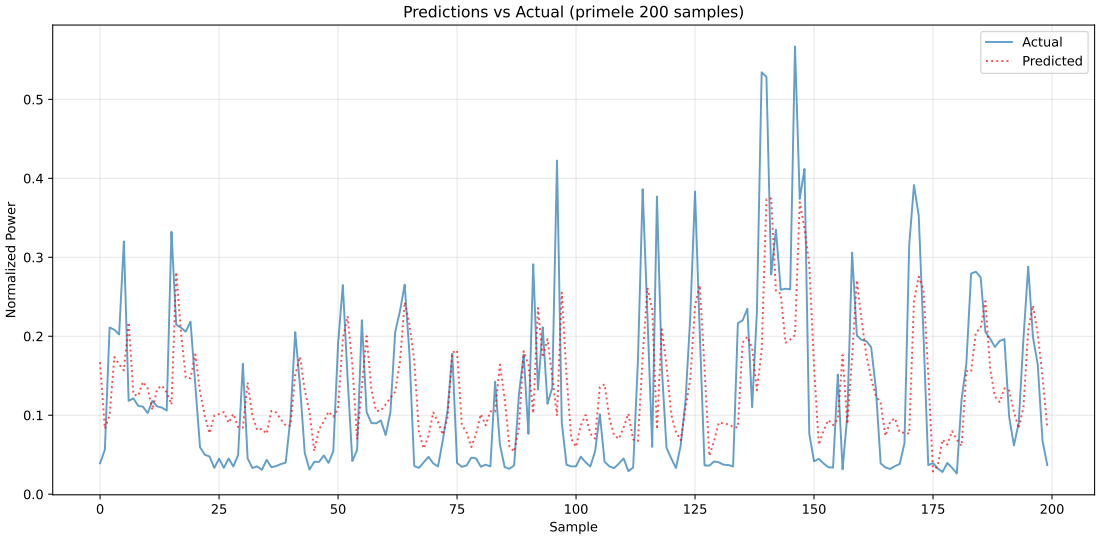
3.3 Rezultat fără date exogene

Inițial, am antrenat o rețea neuronală LSTM doar pe seria de timp a coloanei „Global_active_power" din setul de date menționat mai sus. Am normalizat seria de timp folosind MinMaxScale. Pentru antrenare am folosit secvențe de lungime 48, corespunzătoare consumului de curent pe 2 zile. Rețeaua are o memorie de lungime 64, 3 straturi și un dropout de 20%. Funcția de pierdere optimizată este Huber Loss, un hibrid între MSE și MAE, care la diferențe absolute mici se comportă ca MSE, iar la diferențe mari se comportă ca MAE. Optimizatorul folosit este Adam. Pentru LSTM fără date exogene, am obținut pierderile: Test Loss (MSE): 0.003395, Test RMSE: 0.058268.



3.4 Rezultat cu date exogene

Am repetat experimentul anterior utilizând date exogene și cu aceiași hiperparametri ca în experimentul trecut. În urma antrenării, am obținut pentru LSTM cu date exogene: Test Loss (MSE): 0.003182, Test RMSE: 0.056405.



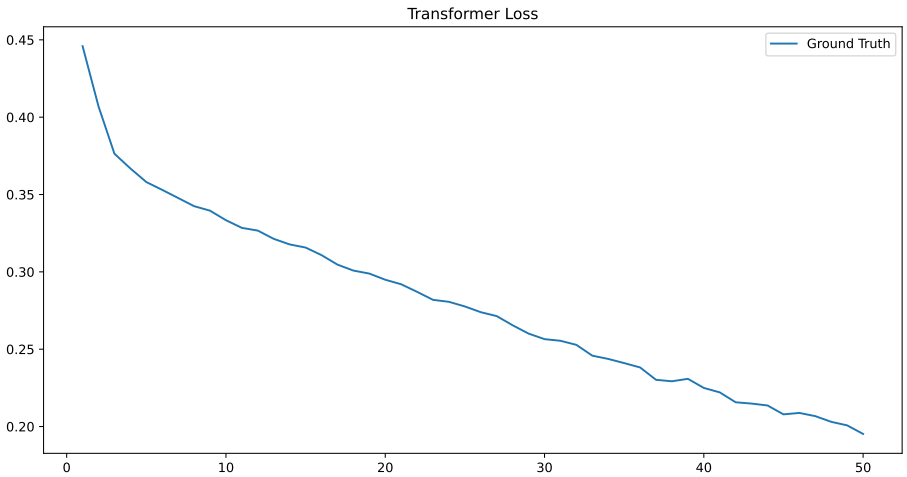
4. Transformeri

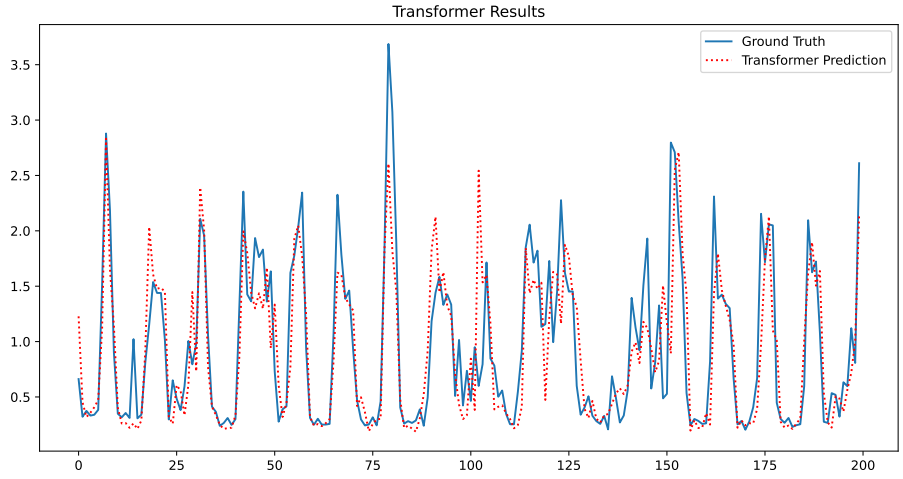
4.1 Ideea din spate

Chiar dacă rețelele neuronale LSTM sunt mult mai bune în ceea ce privește memoria pe termen lung, acestea tot sunt supuse riscului problemei gradientului. La transformeri în schimb, această problemă dispare de tot deoarece între fiecare două poziții din secvență, fie ele cuvinte, tokeni sau numere dintr-o serie de timp, „distanța" este 1. Cu alte cuvinte, un transformer se poate uita oricât de mult în spate, fără ca acesta să piardă context. Un alt avantaj important al transformerilor față de RNN sau LSTM este paralelizarea. Spre deosebire de RNN sau LSTM care trebuie să treacă prin fiecare ciclu unul după altul pentru a ajunge la rezultat, transformerii pot fi paralelizați, ei putând să se uite la toată secvența dintr-o dată, făcând transformerul să fie mult mai rapid decât RNN şi LSTM. Din aceste motive, transformer-ul stă la baza LLM-urilor precum ChatGPT, Gemini sau DeepSeek.

4.2 Rezultat

Am antrenat un transformer cu următorii hiperparametri: Lungimea secvenței: 24, Stare ascunsă: 64, Straturi: 2, Capete: 4, Epoci: 50, Normalizare: StandardScaler, Rata de învățare: $5 \cdot 10^{-4}$, Funcție de optimizat: Mean Squared Error, Optimizator: Adam.




Transformer Results
Ground Truth
Transformer Prediction
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
25
50
75
100
125
150
175
200

5. ARMA

5.1 AR

Modelul AR, sau Autoregresiv, este după cum îi spune și numele un model de regresie adaptat la serii de timp. Ideea de bază este că trecutul influențează viitorul, iar acest lucru este modelat matematic de o combinație liniară a ultimelor p valori din trecut, unde p este dimensiunea modelului. Formula matematică este următoarea: $\hat{y}_i = x^T y$ unde x reprezintă cele p ponderi ale modelului iar y reprezintă ultimele p valori înainte de cea de pe poziția i. Fiind o problemă de regresie, noi vrem să minimizăm eroarea pătratică a modelului, folosind ecuația de mai sus pentru pozițiile i, $i-1, ..., i-m+1$, unde m este orizontul modelului. Soluția problemei, adică valorile ponderilor este dată de formula $X = (Y^T Y)^{-1} Y^T y$ unde:

$$y = \begin{bmatrix} y[i] \\ y[i-1] \\ \vdots \\ y[i-p+1] \end{bmatrix} \tag{5.1}$$

$$Y = \begin{bmatrix} y[i-1] & y[i-2] & \dots & y[i-p] \\ y[i-2] & y[i-3] & \dots & y[i-p-1] \\ \vdots & \vdots & \ddots & \vdots \\ y[i-m] & y[i-m-1] & \dots & y[i-m-p+1] \end{bmatrix} \tag{5.2}$$

5.2 MA

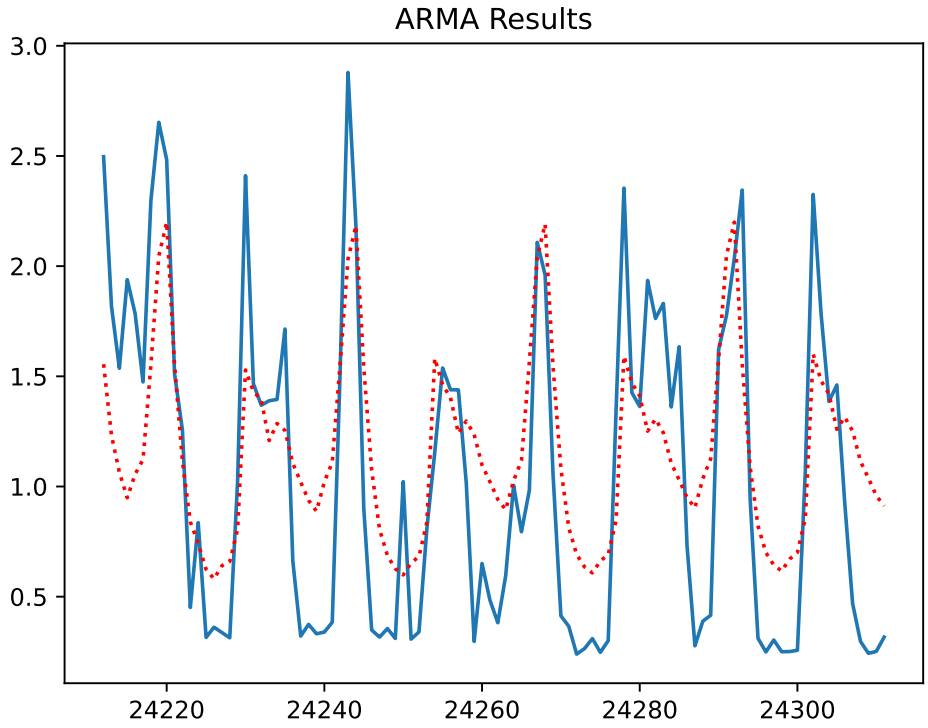
Modelul MA, sau Moving Average, urmărește aceeași idee ca și modelul AR, cu diferența că acesta folosește o combinație liniară a erorilor anterioare. Cum calculul predicției se face tot printr-un produs scalar, acesta se rezolvă prin metoda celor mai mici pătrate.

5.3 ARMA

Modelul ARMA este combinația dintre cele două modele menționate anterior. Vom avea atât parametri autoregresivi, cât și parametri de mediere. Din acest motiv, minimizarea erorii acestor parametri este mult mai complicată.

5.4 Rezultat

În urma antrenării modelului sezonal ARMA, care se uită la secvențe de câte 24 de eșantioane, sau 24 de ore. Am folosit un parametru autoregresiv, un ordin de diferențiere și un parametru de mediere, obținând media pătratelor erorilor de 23.8666.



Bibliografie

1. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
2. https://cs.unibuc.ro/ crusu/ps/lectures.html
3. https://medium.com/@ottaviocalzone/an-intuitive-explanation-of-lstm-a035eb6ab42c
4. https://youtu.be/AsNTP8Kwu80?si=vrdUGieLr-Acgfzx
5. https://youtu.be/YCzL96nL7j0?si=P_r-cm6HPRRwv-4v