Analiză comparativă a arhitecturilor LSTM, GRU și Transformer pentru predicția consumului de energie electrică

Ana-Maria Rusu
Andrei-George Drudun
Bogdan-Valeriu Țicu

Februarie 2026

1 Abstract

Această lucrare prezintă o analiză comparativă a performanței a trei arhitecturi aplicate în problema predicției seriilor de tip, specific pentru prognoza consumului orar de energie electrică. Studiul se concentrează pe evaluarea a două modele bazate pe rețele neuronale recurente (RNN) - Long Short-Term Memory (LSTM) și Gated Recurrent Unit(GRU) - în contrast cu o arhitectură modernă bazată pe mecanismul de atenție, de tip Transformer, și anume, PatchTST. Modelele au fost implementate manual aproape integral pentru a oferi un control total asuprar parametrilor și o înțelegere profundă a funcționării interne.

Evaluarea experimentală a fost realizată prin variația dimensiunii ferestrei de intrare și a orizontului de predicție, utilizând metricile MSE, MAE, MAPE și R^2 pentru măsurarea performanței în cadrul a 6 experimente. Rezultatele evidențiază compromisuri între cele trei arhitecturi analizate: modelele recurente (LSTM și GRU) oferă performanțe superioare față de Transformer în acest cadru experimental, GRU fiind mai eficient pentru predicții pe termen scurt, iar LSTM dovedindu-se mai stabil și potrivit pentru predicții pe termen lung.

2 Introducere

2.1 Contextul problemei

Predicția precisă a consumului de energie electrică reprezintă un pilon fundamental în gestionarea eficientă a rețelelor inteligente (Smart Grids). Echilibrul dintre cerere și ofertă este critic, întrucât energia electrică nu poate fi stocată eficient în cantități mari, iar supraproducția sau subproducția pot genera costuri semnificative și instabilitate în rețea. Provocarea majoră în acest domeniu constă în natura non-staționară și haotică a datelor reale, unde tiparele istorice sunt influențate de factori externi variabili. În ultimii ani, volumul imens de date colectate a făcut ca metodele statistice clasice (precum ARIMA) să devină insuficiente, lăsând loc modelelor de învățare profundă (Deep Learning) capabile să captureze dependențe complexe și neliniare.

2.2 Obiectivul proiectului

Obiectivul principal al acestei lucrări constă în analiza comparativă a performanței arhitecturilor secvențiale (LSTM/GRU) și a celor bazate pe mecanisme de atenție (Transformer) în sarcina de prognoză a consumului energetic orar. Studiul își propune să investigheze compromisul dintre acuratețea predicției și costul computațional.

2.3 Metode de rezolvare

Pentru atingerea obiectivelor propuse, proiectul utilizează și compară trei arhitecturi diferite:

LSTM (Long Short-Term Memory): O arhitectură recurentă concepută pentru a rezolva problema dispariției gradientului, capabilă să rețină informații pe perioade lungi.

GRU (Gated Recurrent Unit): O variantă optimizată a LSTM, care utilizează un număr mai mic de porți pentru a obține o viteză de antrenare mai mare, având performanțe similare cu LSTM.

Transformer: Un model bazat exclusiv pe mecanisme de auto-atenție (Self-Attention), care permite procesarea paralelă a datelor și capturarea corelațiilor globale între diferite momente de timp.

3 Setul de date

Pentru antrenarea și evaluarea modelelor a fost utilizat setul de date *AEP Hourly Energy Consumption*, disponibil pe platforma Kaggle. Acest set de date este relevant pentru studiul seriilor de timp, deoarece reflectă comportamentul real al consumatorilor și variabilitatea cererii de energie în funcție de factori temporali precum ora din zi, ziua săptămânii sau sezonalitatea.

Datele reprezintă o serie de timp univariată, eșantionată la intervale orare, formată dintr-o secvență continuă de valori reale. Fiecare observație x_t corespunde consumului energetic înregistrat la momentul t. Valorile sunt exprimate inițial în megawați (MW), însă în cadrul acestui studiu consumul a fost scalat în gigawați (GW), pentru a evita apariția unor valori foarte mari ale erorii pătratice medii (MSE), de ordinul milioanelor.

Problema este formulată ca o relație între trecut și viitor, în care modelele utilizează o fereastră istorică de observații pentru a prezice consumul energetic pe un orizont de timp stabilit, specific unei sarcini de predicție multi-step pentru serii temporale.

4 Metodologie

4.1 Serii de timp

Problema predicției consumului de energie electrică poate fi privită ca o problemă de serii de timp deoarece datele sunt ordonate în timp și valorile viitoare depind de cele din trecut. În acest context, seria de timp este formată din valorile succesive ale consumului măsurate la intervale regulate (de 1h). Pentru a rezolva problema, se folosesc valori trecute ale consumului pe o anumită perioadă de timp, ce sunt oferite unui model drept intrare, iar acesta trebuie să prezică consumul pentru un interval viitor. Astfel, problema este transformată într-o relație între trecut și viitor, iar modelele pentru serii de timp învață aceste relații pentru a putea realiza predicții cât mai corecte.

4.2 RNN

4.2.1 Descriere conceptuală

Un RNN este un tip de rețea neuronală destinată procesării datelor secvențiale. Aceasta utilizează conexiuni recurente, astfel încât starea obținută la un moment de timp va servi drept intrare la pasul următor.

Unitatea fundamentală a unui RNN este unitatea recurentă. Ea menține memoria internă a rețelei, numită și stare ascunsă. Această memorie internă este actualizată la fiecare pas în timp, ținând cont de intrarea curentă și de starea ascunsă anterioară. În acest mod, rețele sunt capabile să mențină dependențe temporale.

Cu toate acestea, RNN-urile nu pot reține dependențe pe termen foarte lung din cauza problemei dispariției gradientului. Astfel, în timp, rețeaua își pierde capacitatea de a păstra informația provenită din intrările îndepărtate din trecut.

4.2.2 Descriere matematică

După cum am menționat anterior, unitatea recurentă este responsabilă pentru memoria internă a rețelei, adică pentru starea ascunsă. Vom nota această stare la momentul de timp t cu $h_t \in \mathbb{R}^p$. De asemenea, vom nota cu $x_t \in \mathbb{R}^d$ intrarea, cu $y_t \in \mathbb{R}^q$ ieșirea la momentul de timp t, iar cu $W \in \mathbb{R}^{p \times p}$, $U \in \mathbb{R}^{p \times d}$ și $V \in \mathbb{R}^{q \times p}$ matricile de ponderi corespunzătoare conexiunilor dintre stări, dintre intrare și stare, respectiv dintre stare și ieșire. Bias-ul asociat stării va fi notat cu $b_h \in \mathbb{R}^p$, iar cel asociat ieșirii cu $b_y \in \mathbb{R}^q$.

Modelul matematic al unui RNN poate fi descris prin următoarele relații, care definesc actualizarea stării ascunse și calculul ieșirii la fiecare pas t de timp, unde funcțiile f și g reprezintă funcții de activare, f fiind asociată în majoritatea cazurilor cu tanh:

$$h_t = f(W \cdot h_{t-1} + U \cdot x_t + b_h)$$
$$y_t = g(V \cdot h_t + b_y)$$

Aceste relații sunt ilustrate în Figura 1, care prezintă structura unității recurente.

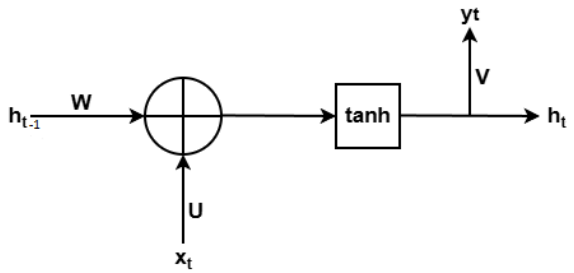

Figure 1: Unitate recurenta

Antrenarea unui RNN se realizează printr-un algoritm numit *Backpropagation Through Time* (BPTT). Acesta presupune desfășurarea rețelei pe întreaga secvență de date și ajustarea parametrilor la fiecare pas în timp.

În cadrul acestui proces, eroarea este calculată la fiecare moment de timp, iar gradientul funcției de pierdere este propagat înapoi prin toate stările anterioare ale rețelei, folosind "regula lanțului".

Un aspect crucial care trebuie menționat este faptul că matricile de ponderi asociate sunt invariante în timp. În caz contrar, rețeaua nu ar fi capabilă să generalizeze, deoarece ar învăța reguli diferite pentru fiecare moment de timp.

Deoarece ieșirea finală este obținută prin aplicarea repetată a unor transformări neliniare, pot apărea probleme precum explozia sau dispariția gradientului, cauzate de propagarea erorii printr-un număr mare de pași în timp. Ca urmare a acestor limitări, au fost propuse arhitecturi precum LSTM și GRU, iar ulterior modele de tip Transformer.

4.3 LSTM

4.3.1 Descriere conceptuală

LSTM-urile reprezintă un tip de RNN ce au fost introduse pentru a aborda problema dispariției gradientului. Arhitectura acestora este similară cu cea a unui RNN, însă unitățile recurente sunt înlocuite cu celule LSTM. Acestea mențin simultan două stări: o stare ascunsă, care reprezintă memoria pe termen scurt, și o stare a celulei, care păstrează memoria pe termen lung și contribuie la stabilizarea propagării gradientului.

4.3.2 Descriere matematică

Un LSTM are cinci componente cheie: starea ascunsă, starea celulei și trei porți care controlează fluxul informației. Structura unei astfel de celule este ilustrată în Figura 2.

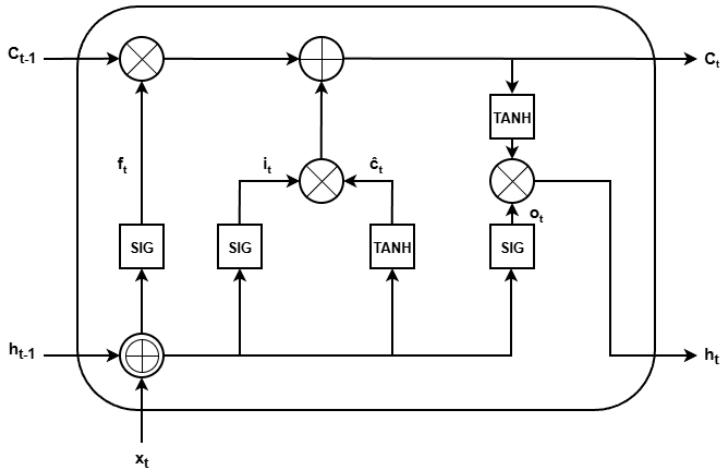

Figure 2: Celula LSTM

Starea celulei reprezintă memoria pe termen lung a unui LSTM. Ea conține informația care este transmisă de-a lungul rețelei în timp și asupra căreia se aplică modificări minime. Astfel, gradientul nu este distorsionat semnificativ și rămâne stabil pe perioade mai lungi de timp, ceea ce ajută la evitarea problemei dispariției gradientului. Vom nota această stare cu $c_t \in \mathbb{R}^p$.

Starea ascunsă reprezintă memoria pe termen scurt. Ea conține informația relevantă pentru pasul de timp curent și este utilizată atât pentru predicția curentă, cât și ca intrare pentru pasul următor al rețelei. Vom nota această stare cu $h_t \in [-1, 1]^p$.

Pentru a gestiona cele două stări descrise mai sus, LSTM utilizează trei tipuri de porți: poarta de uitare, poarta de intrare și poarta de ieșire.

Poarta de uitare este responsabilă pentru eliminarea informațiilor care nu mai sunt necesare din memoria pe termen lung (starea celulei). Aceasta primește două intrări: starea ascunsă anterioară $h_{t-1} \in [-1, 1]^p$ (ce poate fi văzută ca un trecut apropiat) și inputul curent $x_t \in \mathbb{R}^d$. Împreună, aceste două valori oferă contextul curent, care este reprezentat matematic prin concatenarea celor doi vectori:

$$[h_{t-1}, x_t] \in \mathbb{R}^{p+d}$$

După aplicarea unei funcții sigmoid asupra acestui context, se obține un coeficient cu valori între 0 și 1, care indică cât din starea celulei anterioare c_{t-1} trebuie păstrată. O valoare apropiată de 0 indică uitarea unei părți mari din c_{t-1}, în timp ce o valoare apropiată de 1 indică păstrarea aproape integrală a acesteia. Vom nota acest coeficient cu f_t, definit matematic prin relația:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

unde $W_f \in \mathbb{R}^{p \times (p+d)}$ și $b_f \in \mathbb{R}^p$ reprezintă matricea de ponderi și vectorul de bias ale porții de uitare.

Vectorul f_t este înmulțit element-cu-element cu starea celulei anterioare c_{t-1}, rezultând o stare intermediară care indică câtă informație este păstrată din memoria anterioară. Această valoare va reprezenta rezultatul oferit de poarta de uitare.

Poarta de intrare este responsabilă pentru adăugarea de informație nouă în starea celulei, contribuind astfel la definirea noii stări a celulei c_t. Funcțional, aceasta este alcătuită din două subcomponente.

Prima subcomponentă combină inputul curent x_t și starea ascunsă anterioară h_{t-1} și aplică o funcție *tanh* pentru a genera o stare candidat, care reprezintă informația nouă ce ar putea fi adăugată în memoria pe termen lung. Vom nota această stare cu $\hat{c}_t$, definită astfel:

$$\hat{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

unde $W_c \in \mathbb{R}^{p \times (p+d)}$ și $b_c \in \mathbb{R}^p$ reprezintă matricea de ponderi și vectorul de bias corespunzătoare stării candidat.

A doua subcomponentă aplică o funcție sigmoid asupra aceleiași combinații de intrări și determină cât din această stare candidat este efectiv păstrată, în funcție de contextul curent. Acest coeficient este notat cu i_t și este definit astfel:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

unde $W_i \in \mathbb{R}^{p \times (p+d)}$ și $b_i \in \mathbb{R}^p$ reprezintă matricea de ponderi și vectorul de bias ale porții de intrare.

Combinarea acestor două subcomponente determină informația nouă care este adăugată stării celulei. Aceasta se însumează cu rezultatul produs de poarta de uitare, rezultând noua stare a celulei c_t, definită matematic prin relația:

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

Poarta de ieșire nu modifică starea celulei, ci controlează ce parte din aceasta este utilizată pentru a produce starea ascunsă și ieșirea curentă a rețelei.

Aceasta primește ca intrare inputul curent x_t și starea ascunsă anterioară h_{t-1}. După aplicarea unei funcții sigmoid, se obține un coeficient care indică cât din informația stocată în starea celulei va fi utilizată. Acest coeficient este notat cu o_t și este definit astfel:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

unde $W_o \in \mathbb{R}^{p \times (p+d)}$ și $b_o \in \mathbb{R}^p$ reprezintă matricea de ponderi și vectorul de bias ale porții de ieșire.

În paralel, starea celulei c_t este trecută printr-o funcție *tanh*. Prin combinarea acestor două rezultate se obține noua stare ascunsă h_t, care reprezintă ieșirea rețelei LSTM la momentul de timp t și memoria pe termen scurt transmisă mai departe. Aceasta este definită matematic prin relația:

$$h_t = o_t \odot \tanh(c_t)$$

4.4 GRU

4.4.1 Descriere conceptuală

Arhitectura GRU a fost introdusă pentru a rezolva problema dispariției gradientului într-o manieră mai eficientă computațional decât LSTM. Spre deosebire de LSTM, GRU nu utilizează o stare a celulei (c_t), ci doar o stare ascunsă (h_t) care servește atât ca memorie pe termen scurt, cât și pe termen lung. Fluxul de informații este controlat prin două porți principale: poarta de actualizare (update gate), care decide câtă informație din trecut trebuie păstrată, și poarta de resetare (reset gate), care decide câtă informație trebuie ignorată pentru a calcula noua stare candidat.

4.4.2 Descriere matematică

Un GRU are trei componente cheie: starea ascunsă (care servește drept memorie unificată și două porți care controlează fluxul informației (poarta de actualizare și poarta de resetare). Structura unei astfel de celule este ilustrată în Figura 3.

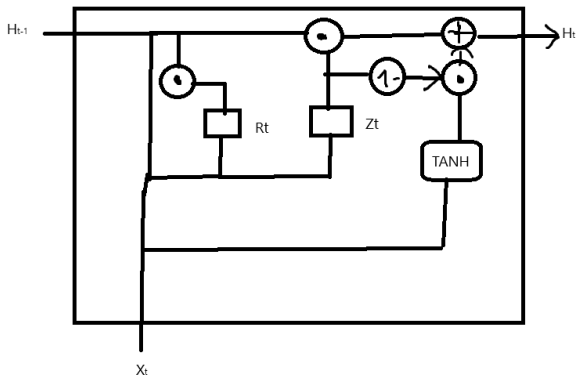

Figure 3: Celula GRU

Starea ascunsă reprezintă memoria unificată a unității GRU. Spre deosebire de LSTM, GRU nu menține o stare a celulei separată, ci utilizează acest vector unic atât pentru a transfera informația în timp (memorie pe termen lung), cât și pentru a efectua predicția la pasul curent. Vom nota această stare cu $h_t \in \mathbb{R}^p$.

Pentru a gestiona fluxul informațional și a rezolva problema dispariției gradientului, GRU utilizează două tipuri de porți: poarta de actualizare și poarta de resetare.

Poarta de actualizare (Update gate) determină cât din informația stării anterioare trebuie păstrată și câtă informație nouă trebuie adăugată. Aceasta preia rolul combinat al porților de uitare și de intrare din LSTM. Ea primește ca intrări starea ascunsă anterioară h_{t-1} și inputul curent $x_t \in \mathbb{R}^p$. Contextul este reprezentat prin concatenarea celor 2 vectori:

$$[h_{t-1}, x_t] \in \mathbb{R}^d$$

După aplicarea unei funcții sigmoid, se obține un coeficient z_t cu valori între 0 și 1. O valoare mai aproape de 0 implică păstrarea stării anterioare. Dacă este mai apropiată de 1, permite actualizarea acesteia cu noua informație candidat. Acesta este definit matematic prin relatia:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

unde $W_z \in \mathbb{R}^{p \times (p+d)}$ și $b_z \in \mathbb{R}^p$ reprezintă matricea de ponderi și vectorul de bias ale porții de actualizare.

Poarta de resetare (Reset Gate) este responsabilă pentru a decide câtă informație din trecut este relevantă pentru calculul noii stări candidate. Funcționează similar cu poarta de actualizare, generând un coeficient de "uitare" notat cu r_t, pe baza intrării și a stării anterioare:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

unde $W_r \in \mathbb{R}^{p \times (p+d)}$ și $b_r \in \mathbb{R}^p$ sunt matricea de ponderi și vectorul de bias ale porții de resetare.

Calculul noii stări se realizează în două etape. Mai întâi, se generează o **stare candidat**, care reprezintă informația nouă propusă. Un aspect specific arhitecturii GRU este că poarta de resetare r_t este aplicată multiplicativ asupra stării anterioare h_{t-1} înainte de a fi procesată de stratul *tanh*. Vom nota starea candidat cu $\hat{h}_t$:

$$\hat{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

unde $W_h \in \mathbb{R}^{p \times (p+d)}$ și $b_h \in \mathbb{R}^p$ sunt ponderile asociate stării candidat, iar $\odot$ reprezinta produsul Hadamard (element cu element).

În final, noua stare ascunsă h_t se obține printr-o interpolare liniară între starea anterioară și starea candidat, controlată de poarta de actualizare z_t. Aceasta este definită matematic prin relația:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t$$

4.5 Transformer

4.5.1 Descriere conceptuală

Arhitectura Transformer a fost introdusă pentru prima dată în lucrarea *"Attention Is All You Need"* [3] în 2017 și a marcat o schimbare de paradigmă profundă în Deep Learning. Spre deosebire de Rețelele Neuronale Recurente (RNN) și LSTM, care procesează datele în mod secvențial, Transformerul utilizează un mecanism de procesare paralelă și conceptul de *Self-Attention*.

Modelul primește ca intrare întreaga fereastră de timp simultan, fiind capabil să stabilească relații între oricare două puncte ale intrării prin mecanismul de Atenție, independent de distanța temporală dintre ele.

În mod clasic, arhitectura este compusă dintr-un *Encoder* – care procesează secvența brută și înțelege contextul istoric, și un *Decoder* – care generează secvența viitoare. În prezenta lucrare, vom folosi o arhitectură de tip **PatchTST** [4], care este *Encoder-only*, deoarece scopul modelului nostru este extragerea unei reprezentări vectoriale robuste a istoricului pentru a estima o valoare viitoare continuă (problemă de regresie).

4.5.2 Descriere matematică

Un Transformer, adaptat pentru serii de timp [4], este alcătuit din 6 componente matematice principale: Normalizarea Instanței, Segmentarea (Patching), Proiecția Latentă, Încapsularea Pozițională, Encoder-ul Transformer și Capul de Predicție. Structura acestei arhitecturi este ilustrată în Figura 4.

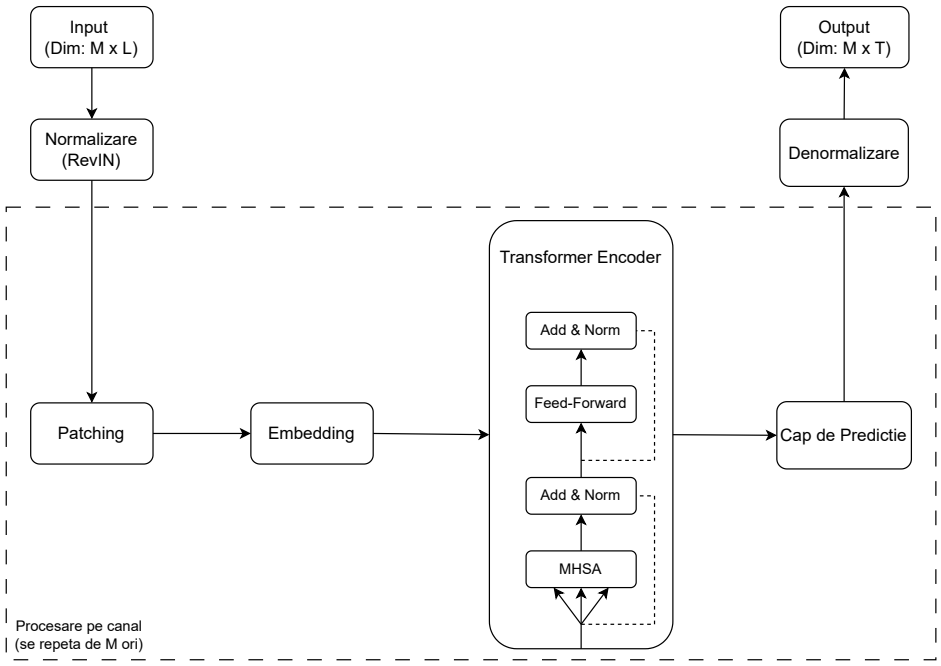

Figure 4: Arhitectura PatchTST

Normalizarea Instanței [5] este etapa prin care se combate degradarea modelului în cazul în care datele de intrare își schimbă distribuția statistică între fazele de antrenare (*train*) și testare (*test*). PatchTST utilizează **Reversible Instance Normalization (RevIN)**.

Prima etapă constă în calcularea mediei și a deviației standard, exclusiv pe axa temporală, pentru fiecare eșantion (*batch*) din lot:

$$\mu_x = \frac{1}{L}\sum_{t=1}^{L} x_t \tag{1}$$

$$\sigma_x = \sqrt{\frac{1}{L}\sum_{t=1}^{L}(x_t - \mu_x)^2 + \epsilon} \tag{2}$$

unde ϵ este o constantă mică pentru stabilitate numerică.

Ulterior, datele sunt standardizate și supuse unei transformări afine cu parametri învățabili:

$$\hat{x}_t = \gamma \odot \left(\frac{x_t - \mu_x}{\sigma_x}\right) + \beta \tag{3}$$

unde $\gamma \in \mathbb{R}$ și $\beta \in \mathbb{R}$ sunt parametri învățați prin *backpropagation*, iar $\odot$ reprezintă produsul Hadamard (înmulțirea element cu element).

Parametrii $\gamma \in \mathbb{R}$ și $\beta \in \mathbb{R}$ permit modelului să **restaureze informația de distribuție** aplicând reciproca normalizarii (Eq. 3):

$$x_t = \sigma_x \odot \left(\frac{\hat{x}_t - \beta}{\gamma} \right) + \mu_x \tag{4}$$

Segmentarea (Patching) este procesul prin care modelul agregă pași temporali adiacenți în unități numite patch-uri. Acestea pot fi suprapuse sau nesuprapuse.

Operația de segmentare transformă vectorul de intrare $x \in \mathbb{R}^{1 \times L}$ într-o matrice de patch-uri $X_p \in \mathbb{R}^{N \times P}$. Această reducere a dimensiunii de la L la N reduce complexitatea mecanismului de atenție de la $O(L^2)$ la $O(N^2)$. De exemplu, pentru $L = 336, P = 16, S = 8$, numărul de tokeni scade de la 336 la aproximativ 42, ceea ce reprezintă o reducere a costului computațional de aproximativ $64\times$ (8^2).

Notăm cu N numărul de patch-uri, calculat astfel:

$$N = \left\lfloor \frac{L - P}{S} \right\rfloor + 2 \tag{5}$$

unde L reprezintă lungimea ferestrei de analiză (*Look-Back Window*), P este lungimea patch-ului, iar S (*Stride*) este pasul de deplasare, reprezentând regiunea nesuprapusă dintre două patch-uri adiacente.

Înainte de procesarea vectorului de intrare x, se adaugă S copii ale ultimei valori x_L la finalul secvenței (operația de *padding*).

Proiecția Latentă reprezinta proiectarea matricei de patch-uri $X_p \in \mathbb{R}^{N \times P}$ intr-un spatiu latent de dimensiune D_{model}, pentru ca modelul sa o poata procesa.

Se aplică o transformare liniară $W_p \in \mathbb{R}^{P \times D_{model}}$:

$$X_d = X_p W_p + \mathbf{b}_p \in \mathbb{R}^{N \times D_{model}} \tag{6}$$

Încapsularea Pozițională (Positional Encoding) este o funcție care mapează poziția unui element (*patch*) dintr-o secvență într-un vector de dimensiune D_{model}. Scopul acesteia este de a adăuga informația de timp și ordine, necesară deoarece mecanismul de atenție procesează *patch*-urile în paralel, iar astfel ordinea temporală originală este pierdută.

În arhitectura originală *Transformer* [3], se utilizează sinusoide de frecvențe diferite pentru a genera câte un vector unic de dimensiune D_{model} pentru fiecare poziție din matricea de intrare $X_d \in \mathbb{R}^{N \times D_{model}}$. Deși în arhitectura *PatchTST* [4] se utilizează adesea *learnable positional encoding* (încapsulare poziționlă cu parametri învățabili), pentru experimentul din lucrarea curentă vom folosi varianta bazată pe sinusoide, definită prin următoarele ecuații:

$$PE_{(pos,2i)} = \sin \left(\frac{pos}{10000^{\frac{2i}{D_{model}}}} \right) \tag{7}$$

$$PE_{(pos,2i+1)} = \cos \left(\frac{pos}{10000^{\frac{2i}{D_{model}}}} \right) \tag{8}$$

unde $i \in [0, D_{model}/2 - 1]$ reprezintă indicele dimensiunii, iar $pos \in [0, N - 1]$ este poziția *patch*-ului în secvență.

Integrarea informației temporale în matricea de *embedding* $X_d \in \mathbb{R}^{N \times D_{model}}$ se efectuează prin operația de adunare:

$$X_{final} = X_d + PE \tag{9}$$

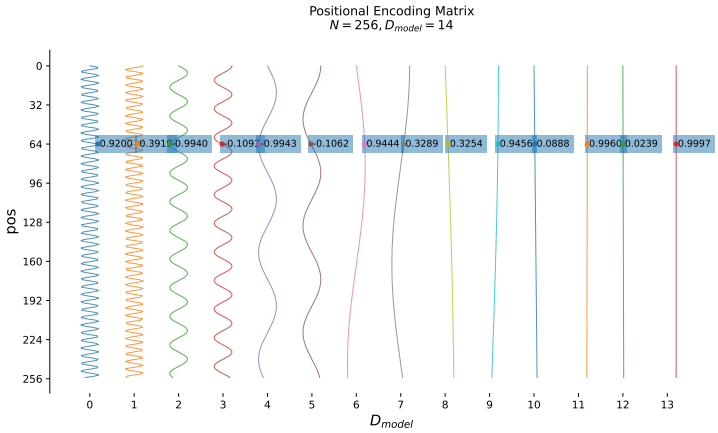

Figure 5: Matricea de Încapsulare Poziționlă (*Positional Encoding Matrix*)

unde $PE \in \mathbb{R}^{N \times D_{model}}$ reprezintă matricea de încapsulare poziționlă, determinată cu ajutorul ecuațiilor 7 și 8.

Alegerea operației de adunare în detrimentul concatenării este justificată de eficiența computațională. Într-un spațiu vectorial de dimensiune suficient de mare ($D_{model} > 128$), vectorii sunt aproape întotdeauna ortogonali între ei. Astfel, modelul învață să filtreze semnalul de poziție de cel de conținut prin intermediul matricelor de proiecție W_Q, W_K și W_V din interiorul mecanismului de atenție.

Un alt avantaj al încapsulării poziționale este capacitatea mecanismului de atenție de a decodifica poziții relative prin proiecții liniare învățate. Datorită proprietăților funcțiilor sinusoidale, orice deplasare temporală k poate fi reprezentată ca o transformare liniară a vectorului de poziție curent, facilitând învățarea dependențelor pe termen lung. În esență, pentru fiecare dimensiune i, există o matrice de rotație $M \in \mathbb{R}^{2\times 2}$ astfel încât:

$$\begin{bmatrix} PE_{pos+k,2i} \\ PE_{pos+k,2i+1} \end{bmatrix} = M \cdot \begin{bmatrix} PE_{pos,2i} \\ PE_{pos,2i+1} \end{bmatrix} \tag{10}$$

Encoder-ul Transformer este alcătuit din trei componente principale:

- **Multi-Head Self-Attention (MHSA)**: Permite modelului să identifice corelații între diferite segmente de timp, indiferent de distanța dintre ele. Cu alte cuvinte, mecanismul de atenție permite fiecărui *patch* să "privească" către toate celelalte *patch*-uri din secvență simultan, pentru a decide care dintre acestea conțin informații relevante pentru predicție.

  Pentru a putea exemplifica din punct de vedere matematic MHSA, trebuie mai întâi să ne îndreptăm atenția către **Scaled Dot-Product Attention**, care reprezintă componenta fundamentală a MHSA:

  Pasul 1: Se calculează trei matrice denumite *Query* (Q), *Key* (K) și *Value* (V), obținute prin proiecții liniare ale intrării:

  $$Q = X_{final} \times W^Q, \quad K = X_{final} \times W^K, \quad V = X_{final} \times W^V$$

  unde W^Q, W^K și W^V sunt matrici de ponderi învățabile pe parcursul antrenării.

  Scurtă descriere intuitivă:

  - Q **(Interogarea):** Determină ce anume caută un *patch* în restul secvenței.
  - K **(Cheia):** Definește modul în care un *patch* este indexat pentru a fi găsit de alte interogări.

– V **(Valoarea):** Selectează informația utilă care va fi transmisă mai departe dacă *patch*-ul este considerat relevant.

Pasul 2: Calcularea unor scoruri care determină importanța celorlalte *patch*-uri pentru elementul curent. Aceste scoruri se obțin prin determinarea corelației (produsul scalar) dintre vectorii $Q^{(i)}$ și $K^{(j)}$, unde i este indicele elementului curent, iar $j \in \{0, \ldots, N-1\}$. Scrisă sub formă matriceală, operația devine:

$$Q \times K^T$$

Pașii 3 și 4: Pentru stabilitatea gradienților, scorurile brute se împart la $\sqrt{d_k}$, unde d_k este dimensiunea vectorilor $Q^{(i)}$ și $K^{(i)}$ (setată la 64 în lucrarea originală [3]). Ulterior, asupra rezultatului se aplică funcția *Softmax*, pentru a transforma fiecare rând de scoruri într-o distribuție de probabilități.

Pasul 5: Matricea de scoruri (ponderile de atenție) obținută anterior se înmulțește cu matricea *Value* (V). Rezultatul reprezintă o sumă ponderată a rândurilor matricei V, având scopul de a amplifica informația relevantă și de a diminua influența segmentelor mai puțin importante pentru contextul curent.

Aplicând toți pașii menționați, obținem ecuația fundamentală a atenției:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{11}$$

pe care o vom denumi în continuare *Cap de Atenție*.

Utilizarea unui singur *cap de atenție* forțează modelul să genereze un singur set de greutăți, fapt ce îi limitează performanța. Această constrângere împiedică modelul să se concentreze simultan pe mai multe aspecte (de exemplu, sezonalitatea și trendul local), rezultând într-o singură medie ponderată, cu informații din toate *patch-urile*, care poate conduce la un rezultat netezit, pierzând astfel detaliile de precizie ale ambelor componente.

Din acest motiv, arhitectura Transformer utilizează **Multi-Head Self-Attention (MHSA)**, care rezolvă limitările menționate prin paralelizarea procesului de atenție, fără a crește substanțial costul computațional total.

Calcularea MHSA:

Pasul 1: Se obțin ieșirile a h capete de atenție diferite ($head_0, head_1, \ldots, head_{h-1}$), fiecare fiind capabil să capteze perspective diferite ale contextului temporal în D_{model}/h subspații de reprezentare distincte.

Pasul 2: Deoarece stratul *feed-forward* acceptă ca intrare o singură matrice, cele h ieșiri sunt concatenate. Pentru a asigura comunicarea între spațiile de reprezentare paralele și pentru a reveni la dimensiunea originală a modelului, se aplică o proiecție liniară utilizând matricea de ponderi $W^O \in \mathbb{R}^{hd_v \times D_{model}}$. Această operație recalibrează datele, pregătindu-le pentru procesarea non-liniară din straturile următoare.

Din pașii descriși rezultă ecuația finală a MHSA:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{12}$$

unde fiecare cap de atenție (head_i) este calculat independent:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{13}$$

unde W_i^Q, W_i^K, W_i^V sunt matricile de proiecție specifice fiecărui cap de atenție.

- **Position-wise Feed-Forward Network (FFN)**: După ce fiecare *patch* a fost contextualizat prin mecanismul de atenție, se aplică un strat FFN separat și identic asupra fiecărui *patch* în parte (conceptul de *Position-wise*). Acest mecanism nu permite comunicarea între segmente în cadrul acestui strat, întreaga interacțiune temporală fiind deja rezolvată anterior de mecanismul de atenție.

  Din punct de vedere matematic, această rețea *feed-forward* constă în două transformări liniare separate de o funcție de activare non-liniară, de regulă ReLU.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{14}$$

  Rețeaua utilizează parametri diferiți de la un strat la altul, însă procesarea rămâne identică pentru toate pozițiile din cadrul aceluiași strat:

  Primul strat proiectează vectorul de *embedding* de dimensiune D_{model} într-un spațiu de dimensiune mult mai mare, $d_{ff} = 4 \times D_{model}$. Prin expandarea spațiului latent, modelul poate extrage și învăța relații semantice mult mai complexe.

  Al doilea strat comprimă spațiul înapoi la dimensiunea originală D_{model} pentru a păstra doar informațiile esențiale și pentru a permite adunarea cu intrarea originală prin intermediul conexiunilor reziduale.

- **Normalizare și Rezidualitate**: În arhitectura Transformer, normalizarea și rezidualitatea funcționează împreună pentru a permite antrenarea rețelelor adânci, prevenind degradarea semnalului.

  Conexiunile Reziduale presupun adunarea intrării unui strat la ieșirea acestuia, având scopul de a adăuga informație nouă fără a degrada reprezentarea inițială:

$$\text{Output} = \text{Layer}(x) + x$$

  Pe lângă evitarea pierderii de informație din *patch*-uri, conexiunile reziduale rezolvă o problemă critică: dispariția gradientului (*vanishing gradient*). În rețelele neuronale adânci, în timpul procesului de *back-propagation*, gradienții tind să scadă exponențial. Conexiunile reziduale creează "punți" prin care gradientul poate circula direct, evitând multiplicările excesive.

  Normalizarea: Se utilizează tehnica *Layer Normalization* [6], care se diferențiază de *Batch Normalization* prin faptul că normalizează valorile de-a lungul dimensiunii caracteristicilor (D_{model}) pentru fiecare eșantion în mod independent.

  Pentru un vector de intrare x de dimensiune D_{model}, normalizarea se realizează conform următoarei formule:

$$\text{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{15}$$

  unde:

  - μ este media aritmetică a tuturor elementelor din vectorul x;
  - σ^2 reprezintă varianța valorilor din x;
  - ϵ este o constantă de valoare foarte mică, adăugată pentru stabilitate numerică;
  - γ și β sunt parametri învățabili (*scale* și *shift*) care permit modelului să re-scaleze rezultatul pentru a optimiza performanța;
  - $\odot$ reprezintă produsul element cu element.

Capul de Predicție Reprezinta etapa finala a arhitecturii, avand rolul de a transforma reprezentarile abstracte returnate de encoder-ul transformer in secvente de valori numerice (seria de timp prezisa).

- **Aplatizare (Flattening):** Matricea de ieșire a *Transformer-ului*, de dimensiune $N \cdot D_{model}$, este transformată într-un vector unidimensional.

$$y_{flat} = \text{Flatten}(Z) \in \mathbb{R}^{(N \cdot D_{model})}$$

- **Proiecția Liniară:** Vectorul aplatizat este trecut printr-un strat liniar (*Fully Connected*), care mapează trăsăturile extrase către orizontul de predicție stabilit, L.

$$\hat{Y} = y_{flat} \cdot W_{head} + b_{head}$$

Denormalizarea: După ce vectorul de ieșire al *Capului de Predicție* este generat, procesul de prezicere se încheie prin aplicarea pasului de denormalizare descris anterior în cadrul mecanismului **RevIN**. Deși transformările succesive ale modelului sunt executate asupra datelor normalizate, prognoza finală trebuie să păstreze magnitudinea și trendul original al seriei temporale. Astfel, utilizând statisticile (μ_x, σ_x) salvate înainte de segmentarea în *patch*-uri, modelul aplică ecuația de denormalizare (4).

5 Implemenatrea modelelor

GitHub Repository: https://github.com/anamaria-rusu/Ps_Proiect

În implementare au fost utilizate aceleași etape de preprocesare a datelor, pentru o comparație corectă între arhitecturi. Astfel, toate modelele folosesc aceeași metodă de creare a eșantioanelor pe ferestre temporale, aceeași împărțire a datelor în seturi de antrenare, validare și testare, precum și un set comun de hiperparametri de bază. De asemenea, au fost aplicate tehnici comune precum gradient clipping, același tip de optimizator, respectiv un mecanism de early stopping.

Toate arhitecturile, cât și toate mecanismele/tehnicile folosite în procesul de implemenatre au fost implementate manual. Singura etapă care nu a fost implementată explicit este calculul gradientelor prin metoda de backpropagation, acesta fiind realizat cu ajutorul mecanismului de autograd din pytorch.

Librării folosite

Am utilizat Pytorch pentru motorul de calcul tensorial și mecanismul de autograd. De asemenea, am folosit Numpy și Pandas pentru procesarea datelor și gestionarea structurată a acestuia.

Crearea și împărțirea datelor

Împărtirea datelor s-a realizat în 80% antrenare, 10% validare și 10% testare. Utilizarea ferestrelor glisante a transformat seria temporală într-o problemă de învățare supervizată, oferind modelelor contextul istoric necesar pentru prognozarea orizontului viitor stabilit.

Hiperparametrii comuni utilizați

Hiperparametru	Însemnătate	Valoare
Dimensiunea ferestrei (win_len)	Număr de pași utilizați ca input	168, 336, 504
Orizont (horizont)	Număr de pași prezișI	24, 48, 72
Learning rate (lr)	Rata de învățare	0.001
Epoci (epochs)	Număr de epoci de antrenare	20
Dimensiune batch (batch_size)	Număr de exemple per batch	64
Patience	Epoci fără îmbunătățire	5
Delta	Îmbunătățire minimă acceptată	0.0001
Beta1	(Adam) Rata de descreștere - gradient	0.9
Beta2	(Adam) Rata de descreștere - gradient pătrat	0.999

Table 1: Hiperparametrii comuni utilizați

Inițializarea greutăților

Am inițializat greutățile cu Xavier pentru a oferi stabilitate numerică la începutul antrenamentului, prevenind overfitul în stratul rețelelor. Această metodă echilibrează varianța intrărilor și iesșirilor fiecărui strat, oferind un punct de plecare optim pentru o convergență mai rapidă.

Optimizator

În cadrul antrenării modelelor au fost testate succesiv mai multe metode de optimizare. Inițial, a fost utilizat SGD clasic, care s-a dovedit instabil, urmat de SGD cu momentum, care a îmbunătățit parțial convergența. În final, a fost implementat manual optimizatorul Adam, care a oferit cele mai bune rezultate. Adam combină avantajele momentum-ului cu o rată de învățare adaptivă, menținând atât media gradientului, cât și media pătratului gradientului. Aceste estimări sunt corectate pentru bias în primele iterații, iar actualizarea parametrilor se face prin normalizarea gradientului, ceea ce conduce la o convergență mai stabilă și mai rapidă.

Gradient clipping

Această tehnică a fost implementată pentru a proteja procesul de învățare împotriva fenomenului de explozie a gradientului. În acest sens, am folosit normalizarea L2. Prin limitarea acestor actualizări, am menținut integritatea ponderilor și stabilitatea procesului de optimizare pe parcursul epocilor.

Mecanismul de early stopping

Am integrat acest mecanism pentru a identifica momentul optim de oprire a antrenării, evitând fenomenul de overfitting prin care modelul ar memora zgpomotul din date. Utilizarea sa ne-a salvat cele mai performante configurații a parametriilor obținută pe setul de validare dacă nu se găsește o configurație mai bună după 5 epoci.

5.1 LSTM

Modelul LSTM a fost implementat manual folosind o clasă în care au fost încapsulate metodele și componentele specifice acestei arhitecturi. Au fost definite funcții auxiliare pentru calculul funcțiilor de activare sigmoid și tanh, utilizate în cadrul porților. De asemenea, a fost implementată o funcție de inițializare a greutăților bazată pe metoda Xavier descrisă anterior.

Cea mai importantă parte a clasei este reprezentată de metoda __init__, unde sunt inițializați parametrii modelului. Aceștia sunt:

- input – dimensiunea intrării, care în acest caz este 1, deoarece seria temporală are un singur canal (consumul de energie)
- hidden – dimensiunea stării ascunse a celulei LSTM, setată la 128
- output – dimensiunea ieșirii modelului, egală cu orizontul de predicție (horizon)
- W_f, W_i, W_o, W_c – matricile de greutăți asociate porților de forget, input, output și informației candidat, inițializate folosind metoda Xavier
- W_y – matricea de greutăți care proiectează starea ascunsă finală în vectorul de predicții
- b_f, b_i, b_c, b_o, b_y – bias-urile corespunzătoare, unde b_f este inițializat cu valoarea 1 pentru a favoriza inițial păstrarea informației în celulă, iar celelalte biase sunt inițializate cu 0.

Pasul de forward a fost implementat manual, respectând structura matematică a unei celule LSTM descrisă la început. Funcția primește un batch de ferestre temporale și procesează fiecare fereastră pas cu pas. Pentru fiecare batch, starea ascunsă h_t și starea celulei c_t sunt reinițializate la zero, deoarece fiecare fereastră este tratată ca un eșantion independent (LSTM stateless). Această abordare este necesară și datorită faptului că ferestrele sunt amestecate aleator între epoci, iar propagarea stărilor între ferestre ar introduce informație eronată. Funcția returnează un vector de predicții pentru fiecare fereastră din batch, corespunzător orizontului de predicție.

Ca workflow general, programul citește datele brute, le împarte în seturi de antrenare, validare și test și creează eșantioane folosind ferestre glisante, conform descrierii anterioare. Datele sunt apoi standardizate prin scăderea mediei și împărțirea la deviația standard, astfel încât distribuția rezultată să aibă medie aproximativ zero și deviație standard unu.

Pentru partea de antrenare, algoritmul parcurge datele pe epoci, iar la începutul fiecărei epoci se aplică un shuffle asupra setului de antrenare. Fiecare fereastră este considerată independentă, iar shuffle-ul modifică doar ordinea ferestrelor ca eșantioane, nu și ordinea valorilor din interiorul fiecărei ferestre. Datele sunt apoi procesate pe batch-uri, pentru fiecare batch fiind calculate predicțiile și eroarea de tip MSE. Calculul gradientelor este realizat cu ajutorul mecanismului de autograd, iar pentru stabilizarea antrenării este utilizată tehnica de gradient clipping. Actualizarea parametrilor se face folosind optimizatorul Adam.

După fiecare epocă, modelul este evaluat pe setul de validare, iar valoarea MSE este utilizată pentru mecanismul de early stopping. Dacă eroarea de validare nu se îmbunătățește timp de un număr prestabilit de epoci, antrenarea este oprită. Parametrii corespunzători celei mai bune performanțe pe validare sunt salvați și restaurați ulterior pentru faza de testare, unde predicțiile sunt denormalizate și comparate cu valorile reale.

În final, modelul este salvat în memorie și se restaurează cu cei mai buni parametri reținuți în pasul de early stopping. După acestea, începe faza de testare.

5.2 GRU

Modelul GRU a fost implementat manual sub forma clasei GRU, construită de la zero pentru a oferi control total asupra parametrilor și pentru a optimiza viteza de execuție prin operații vectorizate. Spre deosebire de implementările standard, această clasă gestionează explicit tensori concatenați pentru a reduce numărul e operații matriceale.

În cadrul metodei de inițializare __init__, parametrii nu sunt definiți separat pentru fiecare poartă. În schimb, s-au creat două matrici principale de greutăți pentru a gestiona simultan poarta de actualizare(z), poarta de resetare(r) și starea candidat($\hat{h}$):

- $W \in \mathbb{R}^{H \times 3H}$: Matricea de greutăți recurente, care procesează starea ascunsă anterioară.
- $U \in \mathbb{R}^{D \times 3H}$: Matricea de greutăți de intrare, care procesează datele observate(x_t).
- $b \in \mathbb{R}^{3H}$: Vectorul de bias concatenat.

Inițializarea acestor parametri s-a făcut folosind o metodă statistică (bazată pe deviația standard), care ajută modelul să înceapă învățarea într-un punct stabil.

Pasul de procesare(*forward*) a fost optimizat pentru viteză. În loc să calculăm impactul intrării pas cu pas, calculăm de la început contribuția datelor de intrare pentru toți pașii de timp deodată. Apoi parcurgem secvența temporală și calculăm valorile porțiilor folosind funcțiile *tanh* și *sigmoid*, prezentate la secțiunea descrierii matematice pentru GRU.

La fel ca la LSTM, tratăm fiecare fereastră de timp independent, adica la fiecare batch de date starea ascunsă (h_t) este resetată.

Pentru antrenare, algoritmul parcurge datele pe epoci, si calculează pentru fiecare epocă predicțiile și eroarea MSE. Dacă după mai multe epoci nu se îmbunătățește eroarea MSE(pe validare) atunci antrenarea este oprită de mecanismul "early stopping". De asemenea, este folosit gradient clipping pentru stabilizarea antrenării iar actualizarea parametrilor se face folosind optimizatorul Adam.

5.3 Transformer

Modelul PatchTST a fost implementat modular, cu fiecare componentă a arhitecturii Transformer încapsulată în module separate pentru claritate și reutilizare.

S-a folosit librăria PyTorch pentru: clasele de bază (nn.Module, nn.Linear, nn.Dropout), operații tensoriale (torch.matmul, torch.sin, torch.cos), mecanismul de autograd pentru backpropagation și suportul GPU. În rest, totul a fost implementat manual.

Cea mai importantă parte a implementării este reprezentată de clasa `PatchTST` din fișierul `patchtst.py`, unde sunt inițializați toți parametrii modelului. Aceștia sunt:

- `seq_len` – lungimea secvenței de intrare, reprezentând numărul de pași temporali din fereastra de intrare (în experimentele noastre, 168, 336 sau 504 ore)
- `pred_len` – lungimea orizontului de predicție, egală cu intervalul pentru care se fac predicții (24, 48 sau 72 ore)
- `in_channels` – numărul de canale/caracteristici, setat la 1 pentru seria univariată de consum energetic
- `patch_len` – dimensiunea fiecărui patch temporal, reprezentând lungimea unei sub-secvențe
- `stride` – pasul între patch-uri consecutive, permițând overlapping
- `d_model` - dimensiunea spațiului de embedding
- `num_heads` – numărul de attention heads paralele în mecanismul de multi-head attention
- `num_layers` – adâncimea encoder-ului Transformer, reprezentând numărul de blocuri encoder
- `d_ff` – dimensiunea rețelei feed-forward, de obicei de 4 ori mai mare decât `d_model`
- `dropout` – probabilitate de dropout pentru regularizare și prevenirea overfitting-ului

Parametru	Valoare
patch_len	16
stride	8
d_model	256
num_heads	4
num_layers	3
d_ff	1024
dropout	0.2

Table 2: Hiperparametrii modelului PatchTST

Fișierele cheie și rolurile lor sunt:

- `patchtst.py` – definește clasa `PatchTST`, inițializează componentele (RevIN, PatchEmbedding, Encoder, head) și implementează metoda `forward`.
- `embeddings.py` – implementează `PatchEmbedding` (padding, `permute`, `unfold`, proiecție liniară, `reshape`) și `PositionalEncoding`.
- `attention.py` – definește `ScaledDotProductAttention` și `MultiHeadAttention` (proiecții Q/K/V, concatenare heads, proiecție finală).
- `encoder.py` – definește `EncoderLayer` și `TransformerEncoder` (suprapunere de layere, conexiuni reziduale).
- `layers.py` – definește `PositionWiseFeedForward`, `ResidualConnection` și `ReversibleInstanceNormalization`.
- `train_patchtst.py` – pipeline complet: încărcare date, split, construire ferestre, antrenare, validare, test și salvare metrici/grafice.
- `run_experiments.py` – bucle de experimente și agregarea rezultatelor în CSV/JSON.

Fluxul efectiv din cod pornește în `PatchTST.forward`. Inputul are forma (B, L, C), unde B este batch size, L este lungimea ferestrei și C numărul de canale. Dacă `use_revin=True`, se aplică `ReversibleInstanceNormalization` și se returnează și statisticile (mean/var) folosite ulterior la denormalizare. Apoi, în `PatchEmbedding.forward` se face:

- padding prin copierea ultimei valori pentru a acoperi pasul `stride`;
- permutare la (B, C, L) pentru ca `unfold` să opereze pe dimensiunea temporală;
- `unfold` cu `size=patch_len` și `step=stride`, obținând $(B, C, N, \text{patch\_len})$;
- proiecție liniară `nn.Linear(patch_len, d_model)` peste ultimul ax, rezultând $(B, C, N, d_{\text{model}})$;
- `reshape` la $(B \cdot C, N, d_{\text{model}})$ pentru procesare channel-independent;
- adăugare `PositionalEncoding`.

Reprezentările intră în `TransformerEncoder`, care aplică secvențial `EncoderLayer` (self-attention + FFN cu `ResidualConnection`). Outputul encoder-ului este aplatizat cu `nn.Flatten(start_dim=1)`, trecut prin `Dropout` și proiectat cu `head` la `pred_len`. Rezultatul este rearanjat în $(B, \text{pred\_len}, C)$ prin `reshape` și `permute`, apoi (opțional) denormalizat cu `revin.inverse` folosind statisticile salvate din intrare.

Inițializarea parametrilor liniari se face în `PatchTST._init_weights` prin Xavier Uniform pentru toate straturile `nn.Linear`. Fișierele `attention.py` și `encoder.py` folosesc această inițializare implicită a modulelor din `PatchTST`.

În `train_patchtst.py`, datele sunt citite cu `pandas`, indexate pe timp, apoi împărțite în *train*, *val*, *test*. Funcția `create_samples` generează ferestre de dimensiuni (win_len, horizon). Seturile sunt normalizate cu `/1000` și reshape-uite în $(N, \text{win\_len}, C)$ respectiv $(N, \text{horizon}, C)$. Antrenarea parcurge epocile cu shuffle pe setul de train, iar în fiecare batch se execută: forward, calcul MSE, `backward`, `clip_grad_norm_` și `optimizer.step`. Se salvează checkpoint-ul `TRF/results/patchtst.pth` când MSE-ul de validare se îmbunătățește.

În `run_experiments.py`, configurațiile sunt definite într-o listă de dicționare; pentru fiecare configurație se antrenează modelul de la zero, se colectează metrici (MSE/MAE/R^2/MAPE) și se scriu fișierele salvate în `results/experiments/`.

6 Experimente

În această secțiune sunt prezentate experimentele realizate pentru a evalua performanța modelelor. Au fost testate mai multe combinații ale dimensiunii ferestrei de intrare (win_len) și ale orizontului de predicție (horizon), cu scopul de a observa cum influențează aceste alegeri capacitatea modelului de a învăța dependențele temporale din date.

6.1 Metode de evaluare

MSE: Penalizează sever erorile mari.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{16}$$

MAE: oferă o interpretare directă a erorii în unitatea de măsură a datelor originale.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{17}$$

MAPE: măsoară eroarea relativă și permite compararea performanței modelelor pe seturi de date diferite.

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{18}$$

R^2 : cuantifică cât de mult din dinamica seriei de timp este explicată de model în raport cu o predicție constantă bazată pe medie.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{19}$$

6.2 Rezultate experimentale

Pentru analiza rezultatelor experimentelor, s-a ales mai întâi fixarea orizontului de predicție la h = 48, iar dimensiunea ferestrei de intrare a fost variată (w = 168, 336, 504), pentru a observa cum influențează fereastra performanța modelului. Ulterior, dimensiunea ferestrei a fost fixată la w = 336, iar orizontul de predicție a fost variat (h = 24, 48, 72), pentru a analiza cum influențează orizontul rezultatele obținute.

6.2.1 horizont = 48 & win_len = 168

LSTM

Epoca	Train MSE	Val MSE
1	0.217508	0.183373
2	0.130604	0.143922
3	0.113732	0.141521
4	0.105076	0.137835
5	0.098418	0.144294
6	0.094067	0.141472
7	0.086126	0.162420
8	0.080954	0.152009
9	0.076835	0.155115

Table 3: Eroare MSE – antrenare LSTM

MSE	MAE	MAPE (%)	R^2
0.8577	0.6588	4.4677	0.8582

Table 4: Performanța modelului

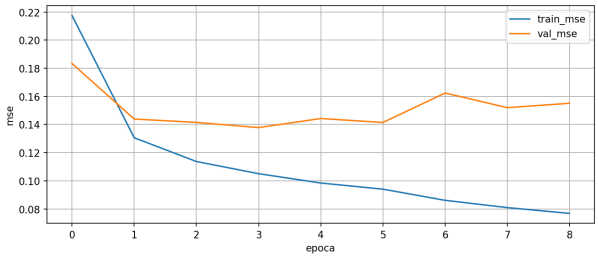

Figure 6: Curba de antrenare și validare

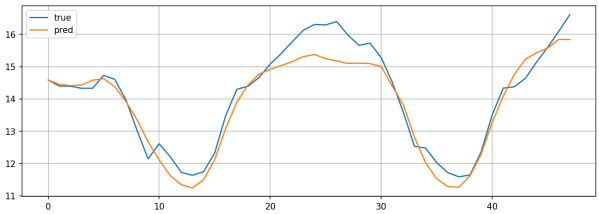

Figure 7: Predicția pe prima fereastră de test

GRU

Epoca	Train MSE	Val MSE
1	0.249998	0.206782
2	0.162562	0.193613
3	0.148948	0.173802
4	0.136961	0.149368
5	0.125219	0.146909
6	0.116181	0.140017
7	0.109270	0.152399
8	0.103992	0.146154
9	0.098648	0.139131
10	0.094958	0.138180
11	0.092287	0.130679
12	0.088953	0.131975
13	0.086988	0.142377
14	0.083831	0.134809
15	0.081710	0.140315
16	0.081710	0.145147
17	0.076139	0.150164

Table 5: Eroare MSE – antrenare GRU

MSE	MAE	MAPE (%)	R^2
0.8524	0.6373	4.33	0.8591

Table 6: Performanța modelului

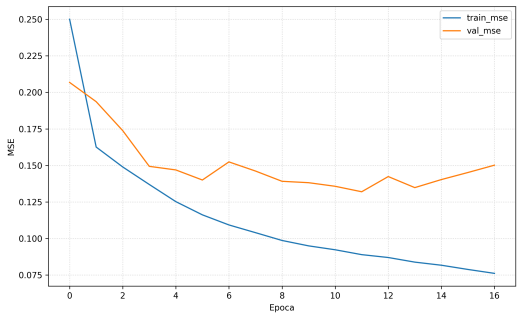

Figure 8: Curba de antrenare și validare

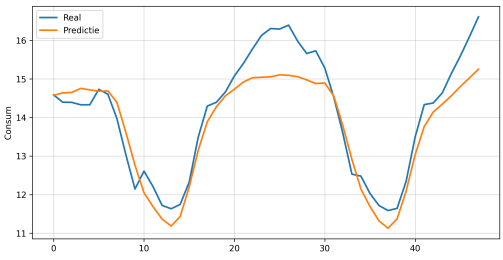

Figure 9: Predicția pe prima fereastră de test

Transformer

Epoca	Train MSE	Val MSE
1	2.673337	1.555910
2	1.236029	1.267683
3	1.070236	1.190688
4	1.000070	1.105759
5	0.958396	1.128985
6	0.928151	1.106965
7	0.902588	1.058488
8	0.884236	1.060973
9	0.868713	1.044681
10	0.859482	1.030036

Epoca	Train MSE	Val MSE
11	0.844542	1.039378
12	0.838414	1.016884
13	0.827503	1.024482
14	0.820741	1.040645
15	0.813858	1.008741
16	0.807802	1.020420
17	0.801565	1.014345
18	0.796471	1.002399
19	0.792375	1.009301
20	0.790019	1.025285

Table 7: Eroare MSE – antrenare Transformer

MSE	MAE	MAPE (%)	R^2
0.9327	0.6858	4.6116	0.8458

Table 8: Performanța modelului

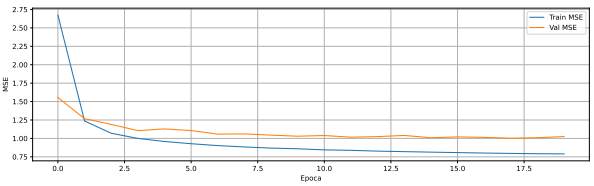

Figure 10: Curba de antrenare și validare

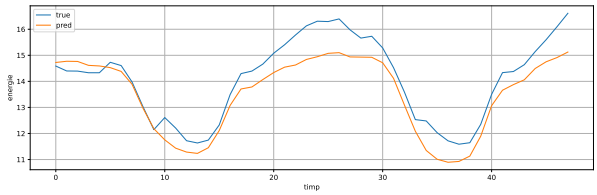

Figure 11: Predicția pe prima fereastră de test

6.2.2 horizont = 48 & win_len = 336

LSTM

Epoca	Train MSE	Val MSE
1	0.222222	0.164785
2	0.135827	0.171672
3	0.146701	0.303977
4	0.120208	0.165154
5	0.104747	0.129234
6	0.111295	0.135686

Epoca	Train MSE	Val MSE
7	0.094849	0.131026
8	0.089981	0.140406
9	0.101775	0.135476
10	0.089026	0.155500
11	0.094653	0.137543
12	0.095042	0.150638

Table 9: Eroare MSE – antrenare LSTM

MSE	MAE	MAPE (%)	R^2
0.8217	0.6378	4.3353	0.8648

Table 10: Performanța modelului

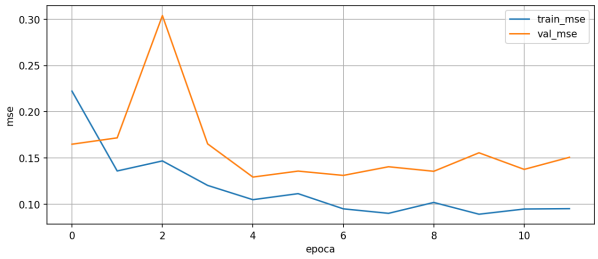

Figure 12: Curba de antrenare și validare

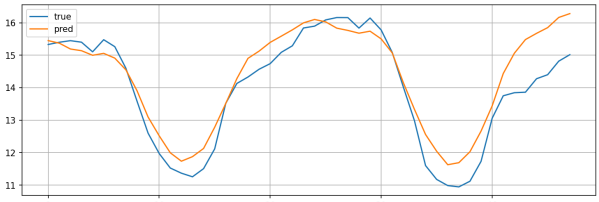

Figure 13: Predicția pe prima fereastră de test

GRU

Epoca	Train MSE	Val MSE
1	0.249652	0.183748
2	0.160389	0.158052
3	0.146593	0.153947
4	0.132902	0.148865
5	0.118776	0.140703
6	0.109491	0.146519

Epoca	Train MSE	Val MSE
7	0.103580	0.126932
8	0.099200	0.123813
9	0.095696	0.133573
10	0.092736	0.126469
11	0.089232	0.131412
12	0.087184	0.131853
13	0.084339	0.134316

Table 11: Eroare MSE – antrenare GRU

MSE	MAE	MAPE (%)	R^2
0.7982	0.6214	4.23	0.8687

Table 12: Performanța modelului

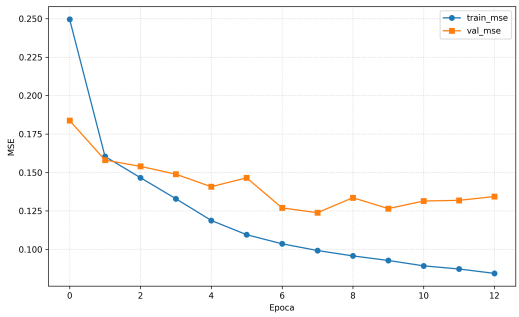

Figure 14: Curba de antrenare și validare

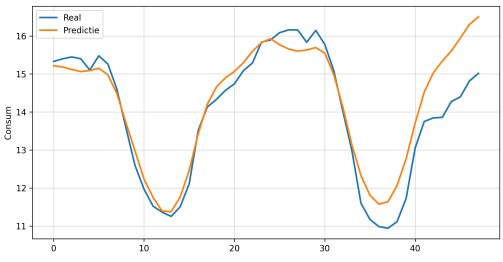

Figure 15: Predicția pe prima fereastră de test

Transformer

Epoca	Train MSE	Val MSE
1	4.414885	1.659479
2	1.355224	1.311694
3	1.127880	1.239177
4	1.042914	1.165118
5	0.993694	1.149280
6	0.960240	1.099431
7	0.933687	1.138200
8	0.911433	1.061072
9	0.890148	1.060587
10	0.876584	1.062924

Epoca	Train MSE	Val MSE
11	0.862574	1.034827
12	0.850595	1.049688
13	0.840464	1.106901
14	0.830654	1.016483
15	0.824013	1.043114
16	0.813581	1.059119
17	0.808161	1.059771
18	0.805523	1.023027
19	0.796137	1.055839

Table 13: Eroare MSE – antrenare Transformer

MSE	MAE	MAPE (%)	R^2
0.9549	0.7061	4.7891	0.8428

Table 14: Performanța modelului

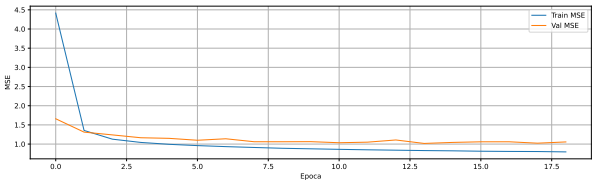

Figure 16: Curba de antrenare și validare

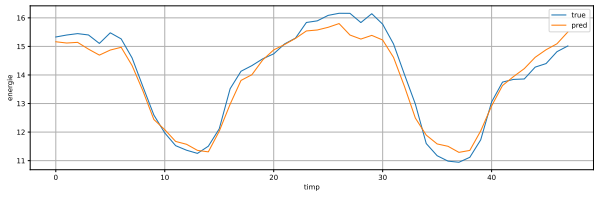

Figure 17: Predicția pe prima fereastră de test

6.2.3 horizont = 48 & win_len = 504

LSTM

Epoca	Train MSE	Val MSE
1	0.212242	0.164514
2	0.135539	0.163798
3	0.120813	0.145948
4	0.118425	0.145765
5	0.138053	0.137036
6	0.100062	0.130023
7	0.167837	0.143396

Epoca	Train MSE	Val MSE
8	0.107094	0.126134
9	0.090899	0.125777
10	0.097913	0.131763
11	0.083571	0.137702
12	0.091082	0.136021
13	**0.076175**	**0.131679**

Table 15: Eroare MSE – antrenare LSTM

MSE	MAE	MAPE (%)	R^2
0.8431	0.6537	4.4846	0.8611

Table 16: Performanța modelului

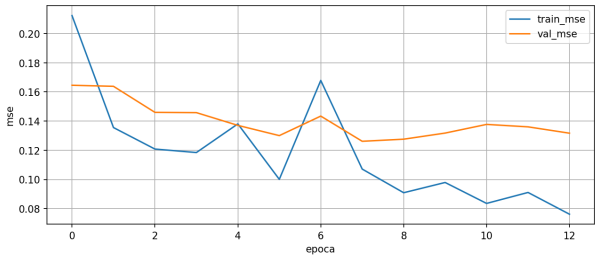

Figure 18: Curba de antrenare și validare

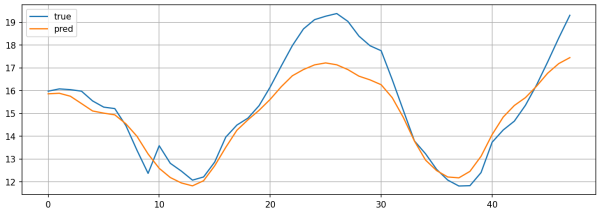

Figure 19: Predicția pe prima fereastră de test

GRU

Epoca	Train MSE	Val MSE
1	0.252492	0.173348
2	0.159402	0.154696
3	0.142704	0.144552
4	0.125715	0.137535
5	0.114431	0.129314
6	0.107316	0.126284

Epoca	Train MSE	Val MSE
7	0.103994	0.134507
8	0.099159	0.129469
9	0.095461	0.148096
10	0.092331	0.144370
11	0.091032	0.136931

Table 17: Eroare MSE – antrenare GRU

MSE	MAE	MAPE (%)	R^2
0.8454	0.6456	4.37	0.8608

Table 18: Performanța modelului

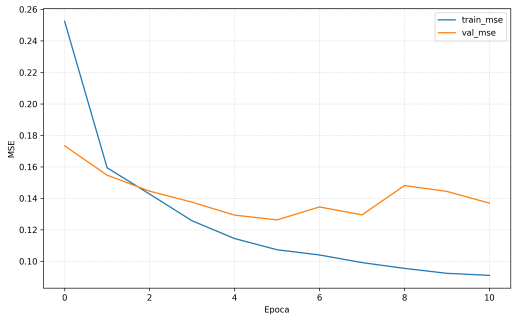

Figure 20: Curba de antrenare și validare

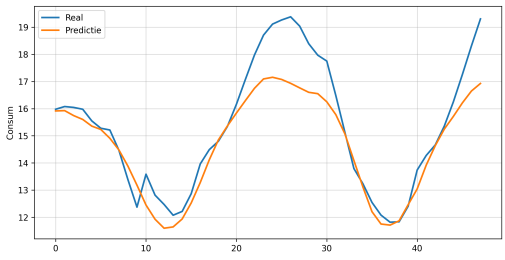

Figure 21: Predicția pe prima fereastră de test

Transformer

Epoca	Train MSE	Val MSE
1	6.822407	1.895819
2	1.496809	1.364852
3	1.230709	1.289821
4	1.124188	1.180408
5	1.058123	1.153926
6	1.012098	1.173830
7	0.978691	1.105621
8	0.949520	1.106973
9	0.926764	1.066042
10	0.909514	1.079002

Epoca	Train MSE	Val MSE
11	0.890790	1.100855
12	0.872047	1.059619
13	0.861265	1.020075
14	0.847743	1.034097
15	0.836790	1.019564
16	0.829039	1.034992
17	0.816925	1.110684
18	0.808859	1.059904
19	0.802939	1.062749
20	0.792069	1.080813

Table 19: Eroare MSE - antrenare Transformer

MSE	MAE	MAPE (%)	R^2
1.0335	0.7369	4.9933	0.8298

Table 20: Performanța modelului

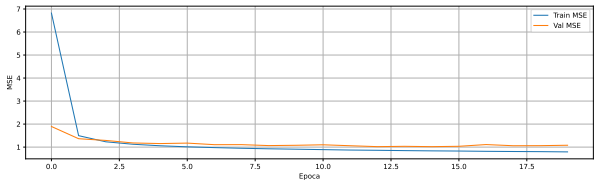

Figure 22: Curba de antrenare și validare

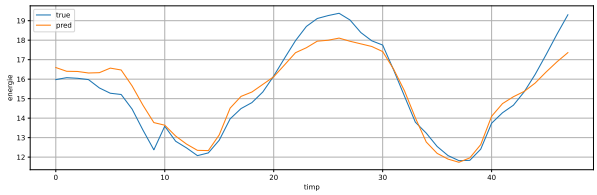

Figure 23: Predicția pe prima fereastră de test

6.2.4 horizont = 24 & win_len = 336

LSTM

Epoca	Train MSE	Val MSE
1	0.144827	0.121756
2	0.087599	0.102020
3	0.075193	0.098297
4	0.065708	0.102007
5	0.061887	0.096242
6	0.055688	0.092432
7	0.052557	0.094616
8	0.050651	0.096812

Epoca	Train MSE	Val MSE
9	0.046085	0.095457
10	0.044663	0.090396
11	0.052814	0.093059
12	0.039394	0.096481
13	0.039836	0.096833
14	0.035227	0.098910
15	**0.033684**	0.107384

Table 21: Eroare MSE – antrenare LSTM

MSE	MAE	MAPE (%)	R^2
0.5466	0.5127	3.4869	0.9103

Table 22: Performanța modelului

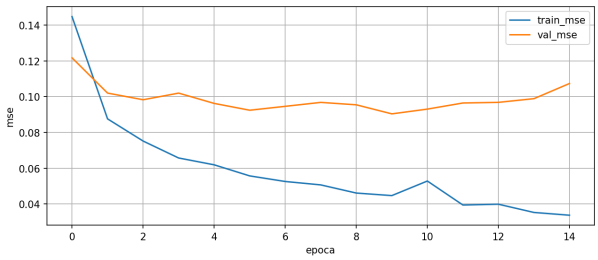

Figure 24: Curba de antrenare și validare

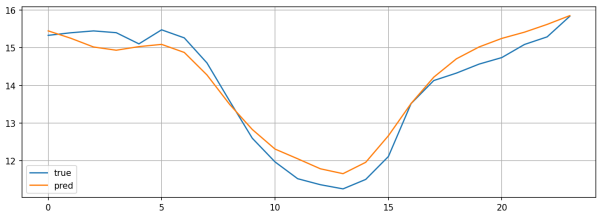

Figure 25: Predicția pe prima fereastră de test

GRU

Epoca	Train MSE	Val MSE
1	0.171233	0.115115
2	0.104706	0.117894
3	0.096891	0.111488
4	0.090830	0.113807
5	0.085545	0.095535
6	0.080485	0.107702
7	0.077049	0.091300
8	0.073471	0.090141
9	0.071344	0.089517
10	0.068354	0.088063

Epoca	Train MSE	Val MSE
11	0.066981	0.082920
12	0.064526	0.088556
13	0.062603	0.090027
14	0.060512	0.086760
15	**0.059198**	0.086762
16	0.057103	0.079941
17	0.055814	0.081473
18	0.054097	0.079236
19	0.053443	0.083943
20	0.052026	0.080670

Table 23: Eroare MSE – antrenare GRU

MSE	MAE	MAPE (%)	R^2
0.4922	0.4949	3.38	0.9193

Table 24: Performanța modelului

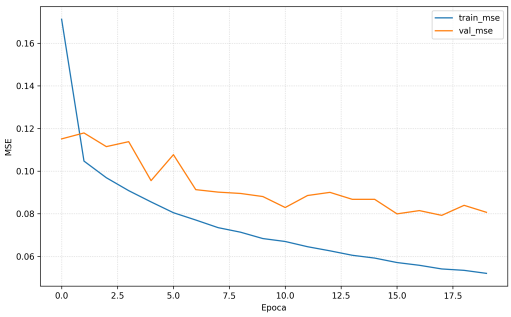

Figure 26: Curba de antrenare și validare

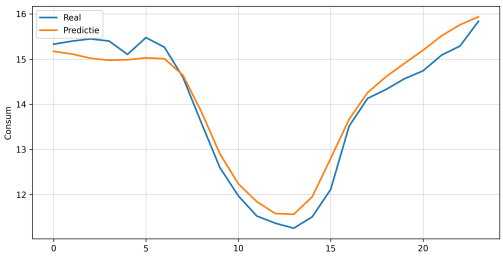

Figure 27: Predicția pe prima fereastră de test

Transformer

Epoca	Train MSE	Val MSE
1	4.256993	1.246582
2	1.029383	1.047637
3	0.847572	0.888318
4	0.769500	0.847698
5	0.724257	0.806251
6	0.696583	0.797217
7	0.671955	0.797742
8	0.653484	0.755423
9	0.641056	0.742537
10	0.625130	0.756694

Epoca	Train MSE	Val MSE
11	0.616019	0.770708
12	0.608981	0.724714
13	0.600592	0.767244
14	0.597887	0.717297
15	0.591267	0.719162
16	0.584332	0.720673
17	0.580539	0.744541
18	0.576853	0.732378
19	0.572734	0.747801

Table 25: Eroare MSE - antrenare Transformer

MSE	MAE	MAPE (%)	R^2
0.6490	0.5902	4.0169	0.8935

Table 26: Performanța modelului

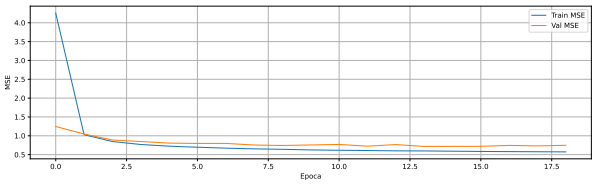

Figure 28: Curba de antrenare și validare

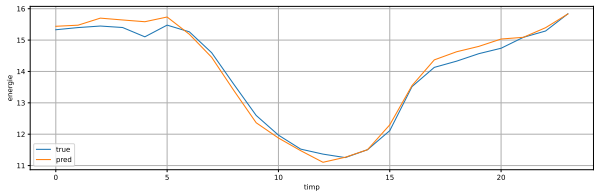

Figure 29: Predicția pe prima fereastră de test

6.2.5 horizont = 72 & win_len = 336

LSTM

Epoca	Train MSE	Val MSE
1	0.259058	0.205412
2	0.163990	0.225693
3	0.150961	0.182558
4	0.171241	0.189109

Epoca	Train MSE	Val MSE
5	0.139313	0.184057
6	0.128079	0.194738
7	0.135843	0.182820
8	0.293964	0.218655

Table 27: Eroare MSE – antrenare LSTM

MSE	MAE	MAPE (%)	R^2
1.2021	0.7908	5.3510	0.8013

Table 28: Performanța modelului

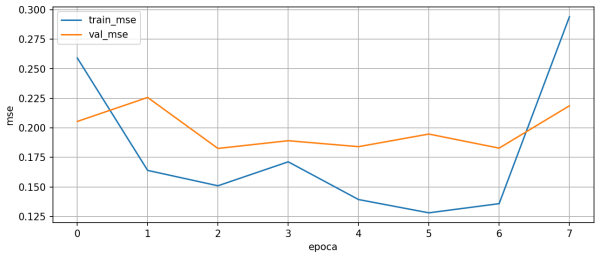

Figure 30: Curba de antrenare și validare

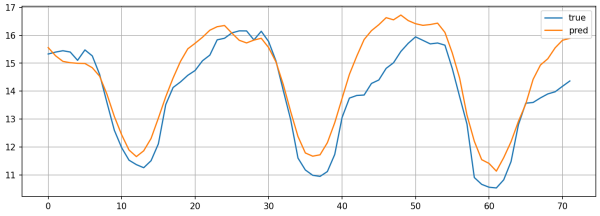

Figure 31: Predicția pe prima fereastră de test

GRU

Epoca	Train MSE	Val MSE
1	0.313955	0.243342
2	0.210307	0.223625
3	0.181359	0.195644
4	0.162085	0.183585
5	0.152569	0.191352
6	0.145666	0.182399
7	0.138360	0.190330

Epoca	Train MSE	Val MSE
8	0.132461	0.178060
9	0.127237	0.190376
10	0.121632	0.207886
11	0.115161	0.208047
12	0.109596	0.212420
13	0.105994	0.231393

Table 29: Eroare MSE – antrenare GRU

MSE	MAE	MAPE (%)	R^2
1.2513	0.7981	5.41	0.7932

Table 30: Performanța modelului

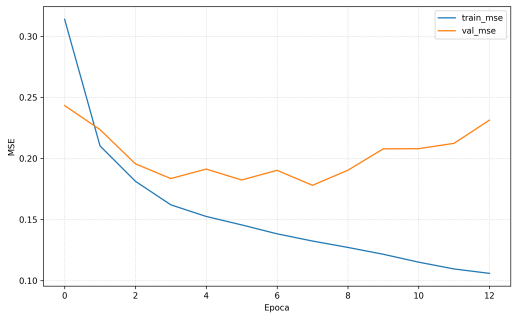

Figure 32: Curba de antrenare și validare

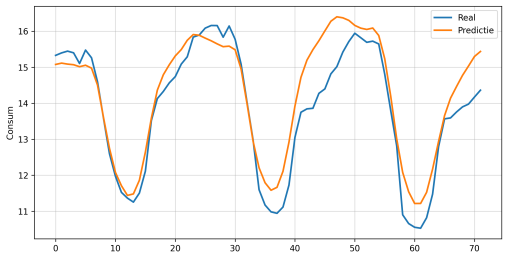

Figure 33: Predicția pe prima fereastră de test

Transformer

Epoca	Train MSE	Val MSE
1	4.888011	2.213878
2	1.662076	1.696233
3	1.416417	1.591183
4	1.324091	1.477344
5	1.273217	1.518156
6	1.229915	1.478228
7	1.197877	1.479815
8	1.175452	1.396964
9	1.151047	1.418779
10	1.130658	1.387215

Epoca	Train MSE	Val MSE
11	1.112462	1.398826
12	1.096979	1.388430
13	1.084537	1.407825
14	1.074785	1.365416
15	1.059508	1.383528
16	1.052696	1.397587
17	1.038985	1.441950
18	1.025903	1.392155
19	1.020284	1.411812

Table 31: Eroare MSE - antrenare Transformer

MSE	MAE	MAPE (%)	R^2
1.4024	0.8608	5.7759	0.7682

Table 32: Performanța modelului

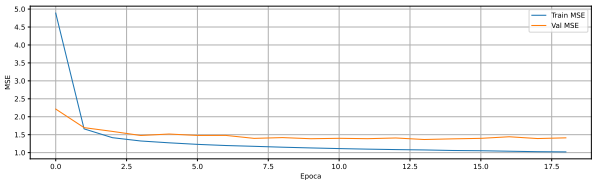

Figure 34: Curba de antrenare și validare

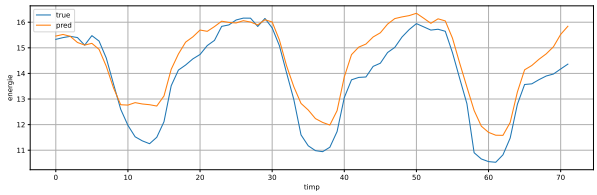

Figure 35: Predicția pe prima fereastră de test

6.3 Interpretări

h	w	LSTM				GRU				TRF			
		MSE	MAE	MAPE	R^2	MSE	MAE	MAPE	R^2	MSE	MAE	MAPE	R^2
48	168	0.8577	0.6588	4.4677	0.8582	0.8524	0.6373	4.33	0.8591	0.9327	0.6858	4.6116	0.8458
48	336	0.8217	0.6378	4.3353	0.8648	0.7982	0.6214	4.23	0.8687	0.9549	0.7061	4.7891	0.8428
48	504	0.8431	0.6537	4.4846	0.8611	0.8454	0.6456	4.37	0.8608	1.0335	0.7369	4.9933	0.8298
24	336	0.5466	0.5127	3.4869	0.9103	0.4922	0.4949	3.38	0.9193	0.6490	0.5902	4.0169	0.8935
72	336	1.2021	0.7908	5.3510	0.8013	1.2513	0.7981	5.41	0.7932	1.4024	0.8608	5.7759	0.7682

Table 33: Performanțele modelelor pentru diferite ferestre și orizonturi

LSTM

Prima serie de experimente a constat în fixarea dimensiunii ferestrei de intrare la win_len = 336 și variarea orizontului de predicție h = 24, 48 și 72. S-a observat că performanța modelului scade pe măsură ce orizontul de predicție crește. Modelul a obținut cele mai bune rezultate pentru h = 24, rezultate mai slabe pentru h = 48, iar pentru h = 72 performanța a fost semnificativ mai slabă. Acest lucru arată că modelul întâmpină dificultăți atunci când trebuie să facă predicții pe perioade mai lungi.

A doua serie de experimente a constat în fixarea orizontului de predicție la h = 48 și variarea dimensiunii ferestrei de intrare win_len = 168, 336 și 504. S-a observat că pentru o fereastră mică de 168, modelul nu reușește să învețe bine tiparele din date, iar pentru o fereastră mare de 504, modelul nu mai folosește eficient toată informația disponibilă. Cea mai bună performanță a fost obținută pentru win_len = 336, în acest cadru.

Per total, cel mai bun rezultat a fost obținut pentru combinația h = 24 și win_len = 336, ceea ce arată că modelul LSTM funcționează cel mai bine atunci când primește o cantitate moderată de date din trecut și trebuie să facă predicții pe termen scurt. În ceea ce privește costul de calcul, cele cinci experimente au avut un timp total de antrenare de aproximativ 9-10 ore pe GPU.

GRU

În prima serie de experimente când setăm win_len = 336 si modificăm orizontul de predicție h = 24, 48, 72 observăm cum performanța modelului scade invers proporțional cu h.

În a doua serie de experimente când păstrăm constant orizontul de predicție și variem win_len observăm că cea mai bună performanță se obține pt fereastra cea mai scurtă, când win_len = 168.

Rezultatul cel mai bun este obținut pentru combinația h=24 și win_len = 336. Deci, modelul funcționează bine când primește o cantitate medie de informație și face predicții cât mai scurte. Modelul a avut un timp total de antrenare de 8-9 ore pe GPU.

Transformer

Din prima serie de experimente, s-a observat o scădere a performanței odată cu creșterea orizontului de prognoză. Modelul a obținut cele mai bune rezultate pentru h = 24 (MSE $\approx$ 0.65), erorile crescând moderat pentru h = 48 (MSE $\approx$ 0.95) și devenind semnificativ mai mari pentru h = 72 (MSE $\approx$ 1.40). Această tendință confirmă dificultatea predicției pe termen lung, chiar și pentru arhitecturile complexe de tip Transformer.

Din a doua serie de experimente, am observat un comportament interesant, unde cea mai bună performanță a fost obținută pentru fereastra cea mai scurtă, win_len = 168 (MSE $\approx$ 0.93). Creșterea dimensiunii ferestrei la 336 și la 504 a dus la o ușoară degradare a metricilor. Acest lucru arată că, pentru acest set de date și configurație, modelul Transformer reușește să extragă trăsăturile esențiale dintr-un istoric mai scurt, iar contextul foarte lung poate introduce zgomot sau redundanță care nu contribuie la acuratețea predicției pe acest orizont.

Per total, cel mai bun rezultat a fost înregistrat pentru combinația h = 24 și win_len = 336. Din punct de vedere al costului de calcul, arhitectura TRF s-a dovedit a fi considerabil mai eficientă decât LSTM și GRU datorită capacității de paralelizare a mecanismului de atenție și a convergenței rapide. Timpul total de antrenare pentru toate cele cinci experimente a fost de aproximativ o oră, comparativ cu durata mult mai mare necesară modelelor recurente secvențiale.

7 Concluzii

7.1 Comparații

Pe baza rezultatelor obținute în Secțiunea 6, pot fi formulate concluzii clare privind influența orizontului de predicție, a dimensiunii ferestrei de intrare și a arhitecturii modelului asupra performanței în predicția multi-step.

Pentru primul set de experimente, în care dimensiunea ferestrei a fost fixată la $w = 336$, toate cele trei modele analizate prezintă o degradare clară a performanței pe măsură ce orizontul de predicție este crescut. În termeni de MSE, diferența dintre $h = 24$ și $h = 48$ este de ordinul 0.2–0.3, iar între $h = 48$ și $h = 72$ crește la aproximativ 0.4–0.5. Aceeași tendință este observată și pentru MAE și MAPE, indicând o acumulare a erorilor pe măsură ce predicția se îndepărtează de momentul prezent. Coeficientul R^2 scade constant pentru toate modelele odată cu creșterea orizontului de predicție.

Comparând modelele recurente, LSTM și GRU au performanțe foarte apropiate pentru orizonturi scurte și medii ($h = 24$ și $h = 48$). Pentru $h = 48$, diferența de MSE dintre cele două modele este mică, de ordinul 0.02–0.03, GRU având valori ușor mai bune. În schimb, pentru un orizont mai mare ($h = 72$), LSTM obține rezultate mai bune decât GRU pe toate metricile, cu diferențe de aproximativ 0.04–0.05 în termeni de MSE și o scădere mai accentuată a R^2 în cazul GRU. Acest comportament sugerează că LSTM este mai stabil pentru predicții pe termen lung.

Modelul Transformer obține rezultate mai slabe comparativ cu LSTM și GRU pentru toate orizonturile analizate. Diferențele de MSE față de modelele recurente sunt de ordinul 0.1–0.2 pentru orizonturi medii și cresc pentru orizonturi mai mari, iar coeficientul R^2 este mai mic cu aproximativ 0.05–0.10. Aceste rezultate indică o capacitate mai redusă a Transformer-ului de a generaliza dependențele temporale pe măsură ce orizontul de predicție crește.

În al doilea set de experimente, în care orizontul de predicție a fost fixat la $h = 48$, influența dimensiunii ferestrei de intrare este mai evidentă. Pentru LSTM și GRU, diferențele dintre ferestrele mici ($w = 168$) și mari ($w = 504$) sunt reduse, de ordinul 0.01–0.02 în termeni de MSE, însă ambele conduc la performanțe mai slabe comparativ cu fereastra de dimensiune medie.

O fereastră de dimensiune medie ($w = 336$) oferă cele mai bune rezultate pentru ambele modele recurente. Diferențele de MSE față de ferestrele $w = 168$ și $w = 504$ sunt de aproximativ 0.03–0.04. Analizând variația performanței, se observă că LSTM prezintă variații mai mici decât GRU la schimbarea dimensiunii ferestrei, ceea ce indică o stabilitate mai bună.

Pentru modelul Transformer, performanța este mai slabă indiferent de dimensiunea ferestrei, iar creșterea acesteia conduce la o degradare graduală a rezultatelor. Diferențele de MSE între ferestre sunt de ordinul $0.02 - 0.05$, iar valorile R^2 scad constant.

În concluzie, rezultatele arată că modelele recurente domină arhitectura Transformer pentru problema analizată. GRU este eficient pentru predicții pe termen scurt, însă pentru orizonturi mai mari LSTM se dovedește mai stabil și mai performant, fiind mai puțin afectat de creșterea orizontului de predicție și de variațiile dimensiunii ferestrei.

7.2 Direcții viitoare

O posibilă continuare a acestui studiu este extinderea orizontului de predicție, pentru a observa până unde modelele LSTM și GRU își mențin performanța și în ce punct diferențele dintre ele devin mai evidente pentru predicții pe termen lung.

De asemenea, ar fi utilă o analiză mai detaliată a alegerii dimensiunii ferestrei de intrare în raport cu orizontul de predicție. Testarea mai multor combinații de w și h ar putea ajuta la identificarea unor alegeri mai bune ale ferestrei, în funcție de cât de departe se dorește realizarea predicției.

În ceea ce privește modelul Transformer, performanța acestuia ar putea fi îmbunătățită prin utilizarea unor volume mai mari de date sau prin ajustări suplimentare ale parametrilor.

References

[1] R. Mulla, *AEP Hourly Energy Consumption*, Kaggle dataset, 2018. Available at: https://www.kaggle.com/datasets/robikscube/hourly-energy-consumption

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, MA, USA, 2016. [Online]. Available: http://www.deeplearningbook.org

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[4] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, "A Time Series Is Worth 64 Words: Long-Term Forecasting with Transformers," in *International Conference on Learning Representations (ICLR)*, 2023.

[5] T. Kim, J. Kim, Y. Tae, C. Kim, J.-H. Choi, and H.-S. Kim, "Reversible Instance Normalization for Accurate Time-Series Forecasting against Distribution Shift," in *International Conference on Learning Representations (ICLR)*, 2022.

[6] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[7] R. M. Schmidt, "Recurrent Neural Networks (RNNs): A Gentle Introduction and Overview," *arXiv preprint arXiv:1912.05911*, 2019.

[8] G. Chen, "A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation," *arXiv preprint arXiv:1610.02583*, 2018.

[9] B. Ghojogh and A. Ghodsi, "Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey," *arXiv preprint arXiv:2304.11461*, 2023.

[10] R. C. Staudemeyer and E. Rothstein Morris, "Understanding LSTM – A Tutorial into Long Short-Term Memory Recurrent Neural Networks," *arXiv preprint arXiv:1909.09586*, 2019.

[11] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk și Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," *arXiv preprint arXiv:1406.1078*, 2014.

[12] J. Chung, C. Gulcehre, K. Cho și Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," *arXiv preprint arXiv:1412.3555*, 2014.