UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENȚĂ

Învățarea supervizată în predicția textelor

Conducător ştiinţific
Prof. univ. dr. Czibula Gabriela

Absolvent
Teodora Ioana NASTE

CUPRINS

INTRODUCERE

Oamenii nu pot să scrie la fel de repede precum gândesc. O persoană tastează aproximativ 41,4 cuvinte pe minut; recordul mondial fiind de 216 cuvinte pe minut [20]. Însă pentru o persoană cu anumite deficiențe, această viteză este dificil de atins. O opțiune ar fii dispozitivele de recunoaștere a vocii, dar acestea nu pot fii folosite de persoanele cu deficiențe de vorbire. De aceea, în prezent, multe aplicații au un sistem de predicție automată a textului scris. Un astfel de sistem încearcă să prezică porțiuni de text, bazându-se pe textul deja introdus de utilizator. Astfel, tastarea caracter cu caracter este înlocuită de o simplă selecție a cuvântului dorit imediat ce acesta este oferit de către sistem, realizând astfel o comunicare mai ușoară și mai rapidă.

Procesarea limbajului natural este o ramură a inteligenței artificiale care se concentrează pe antrenarea calculatorului să proceseze și să interpreteze cantități mari de text. În acest domeniu se clasifică și predicția automată de texte. În proiectul prezentat, s-a implementat folosind Modele Markov cu stări ascunse, o metodă de a prezice cuvântul următor dintr-o secvență de mai multe cuvinte, primind înainte date de antrenament.

În acest scop, a fost dezvoltată o aplicație de tip email care asistă utilizatorul în scrierea de mesaje, sugerându-i completări în timp real, în funcție de textul scris până în momentul curent =, eficientizând astfel întreg procesul.

Lucrarea este structurată în 4 capitole care prezintă concepte și noțiuni pentru o mai bună înțelegere a algoritmilor și tehnicilor utilizate. În continuare voi prezenta pe scurt conținutul celor 4 capitole.

În Capitolul 1 sunt prezentate aspect teortice despre rețele neuronale și modul lor de funcționare. Capitolul acoperă atât rețele neuronale clasice,cât și rețele neuronale recurente. Tot aici sunt descries rețelele neuronale LSTM (*Lonn Short-Term Memory*).

În Capitolul 2 este prezentat modelul Markov cu stări ascunse. Aici se găsesc noțiuni de bază care defines modelul, de exemplu lanț Markov și process Markov, precum și metode de utilizare al acestui model în rezolvarea de problem specifice.

Capitolul 3 prezintă abordări din literature în ceea ce privește predicția textelor. Sunt prezentați metode diverse de implementare a unui mecanism de predicție, precum și rezultatele opțiunte de cercetători.

În final, Capitolul 4 prezintă aplicația SpeedyTalk, etapele de dezvoltare ale acesteia, urmate de un scurt manual de utilizare și de rezultatele experimentale. Având în vedere că oriunde mai este loc de îmbunătățiri, aplicația poate fii extinsă în funcție de nevoile utilizatorilor.

CAPITOLUL 1
Rețele neuronale artificiale

Introducere

Ideea de rețea neuronală a apărut încă din anul 1943, când McCulloch și Pitts au introdus pentru prima dată conceptul de neuroni. Aceștia erau elemente conceptuale ale unui circuit care puteau efectua simple operații. Însă interesul în rețelele neuronale a apărut abia în anii ’80, când dezvoltarea tehnologiei a crescut capacitatea de procesare a calculatoarelor. [10] De-a lungul timpului modelul de rețea neuronală s-a dezvoltat pentru a putea rezolva probleme cât mai diverese, de exemplu detecția obiectelor în imagini sau recunoașterea vorbirii; astfel au apărut modele mai complicate ale rețelelor neuronale clasice.

În acest capitol, voi prezenta pe scurt modul de funcționare al rețelelor neuronale tradiționale, urmând să detaliez rețelele neuronale recurente și un model aparte, anume LSTM (*Long Short-Term Memory*).

1.1 Rețele neuronale artificiale

1.1.1 Modul de funcționare

Rețelele neuronale artificiale sunt modele inspirate din structura creierului uman. Acestea sunt formate din un număr foarte mare de neuroni artificiali, denumiți noduri, fiind uniți de muchii unidirecționale. Fiecare nod j are asociată o funcție de activare l_j , iar fiecare muchie are asociată o pondere $\omega_{jj'}$, ce smenifică ponderea de la nodul j la nodul j' și nu este echivalentă cu $\omega_{j'j}$, care reprezintă ponderea în sens invers. Astfel , valoarea v_j a nodului j , calculată aplicând funcția de activare este:

$$v_j = l_j(\sum_{j'} \omega_{jj'} \cdot v_{j'})$$

În figura 1 [11], este prezentat felul în care un nod calculează valoarea unei funcții nonlineare , folosind suma ponderilor datelor sale de intrare.

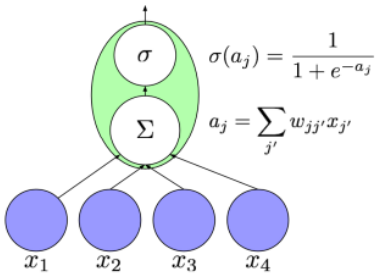

Figura 1. Modul de funcționare al unui neuron artificial

Printre proprietățile particulare ale rețelelor neuronale se numără abilitatea de a învăța și a se adapta, de a generaliza o problemă sau de a organiza datele.

1.1.2 Funcția de activare

Funcția de activare poate să fie o funcție liniară simplă, astfel valoarea unui nod s-ar calcula după formula [10]:

$$v_j = \sum_{j'} \omega_{jj'} \cdot v_{j'} + \theta$$

Funcțiile de activare cel mai des folosite sunt funcțiile sigmoid $\sigma(z) = \frac{1}{(1+e)^{-z}}$ și funcția tanh $\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Acestea sunt foarte practicate pentru rețelele *feedforward* și au fost aplicate și pentru rețelele neuronale recurente. O altă funcție de activare este funcția ReLU (*rectified linear unit*) $l_j(z) = \max(0, z)$, care este folosită, în principal, la rețelele neuronale cu structuri adânci (*deep neural networks*) pentru obiective precum detecția imaginilor, dar a fost folosită și în rețelele neuronale recurente de către Bengio et al. (2013) [11].

1.1.3 Propagarea erorii

Propagarea erorii este procesul prin care rețeaua învață. Valoarea obținută de noduri este trimisă la nodurile de ieșire care compară cu rezultatul dorit. Astfel o să avem o eroare e_o pentru fiecare nod de ieșire. Scopul principal este minimizarea acestei erori. Metoda cea mai simplă pentru atingerea acestui scop este să se distribuie eroare unui nod de ieșire către toate nodurilor conectate cu acesta, proporţional cu valorile ponderilor asociate acestor conexiuni; după care se modifică ponderile de pe stratul de intrare și stratul ascuns.

1.1.4 Limitări

Rețelele neuronale clasice sunt capabile să genereze aproximarea unei funcții nonlineare, fiind dat un set de exemple. Însă , este greu de stabilit cât de mare ar trebui sa fie rețeaua (câte noduri și câte straturi să conțină) pentru a produce un rezultat cât mai bun. Pe de altă parte, o altă problemă este timpul de antrenare al rețelei, care poate deveni foarte mare pentru rețelele cu multe straturi. În același timp, aceste rețele depind foarte mult de puterea de procesare a calculatorului [14].

1.2 Rețele neuronale recurente

1.2.1 Modul de funcționare

Spre deosebire de rețelele neuronale tradiționale, cele recurente au , în plus, muchii între noduri adiacente, formând cicluri și introducând noțiunea de timp în model. Aceste muchii se numesc muchii recurente. Pot exista și cicluri de dimensiune 1, adică un nod să aibă o muchie care indică spre el însuși. Astfel, la un moment t, nodurile cu o muchie recurentă primesc informație din datele curente $x(t)$, dar și de la valorile anterioare a nodurilor ascunse $h(t-1)$. Un model de rețea neuronală recurentă este prezentată în figura 2 [11].

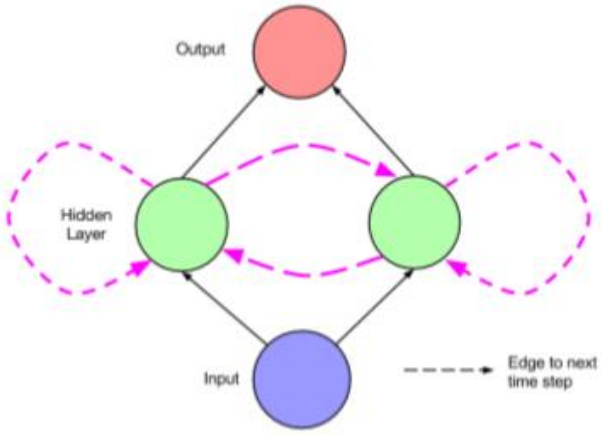

Figura 2. Rețea neuronală recurentă

Fiind dată o secvență de date de intrare $(x_1, \dots, x_T)$, o RNN calculează secvența de date de ieșire $(y_1, \dots, y_T)$, iterând următoarea ecuație [22] :

$$h_t = \sigma(W^{hx}x_t + W^{hh}h_{t-1})$$
$$y_y = W^{yh}h_t$$

Aici W^{hx} reprezintă matricea ponderilor dintre datele de intrare și stratul ascuns, iar W^{hh} este matricea ponderilor recurente la momente diferite de timp.

1.2.2 Antrenarea rețelei

Antrenarea unei rețele RNN poate să fie foarte dificilă din cauza dependințelor foarte largi. Principalele probleme care apar în antrenarea rețelei sunt *vanishing* și *exploding gradients*. Acestea au loc în timpul propagării erorii. Presupunem că avem că avem o rețea simplă cu un singur nod recurent, iar la momentul τ rețeaua primește date de intrare și calculează o eroare la momentul *t* . Astfel , contribuția pe care o au datele de intrare de la momentul τ pentru datele de ieșire de la momentul *t* ori se va apropia de 0 sau va ”exploda” cu cât diferența t - τ devine mai mare. Probabilitatea ca oricare dintre aceste fenomene să aibă loc depinde de ponderea de pe muchiile recurente $(\omega_{jj} < 1 \, sau \, \omega_{jj} > 1)$, dar și de funcția de activare. De exemplu, dacă funcția de activare este sigmoid, atunci cel mai probabil valoarea va risca să se apropie foarte mult de 0 și să apară *vanishing gradient*, dar dacă se folosește funcția ReLU max(1,x) , atunci e ușor să ne imaginăm probabilitatea ca valoarea să crească foarte mult și să apară *exploding gradient.*

1.2.3 Limitări

Deși inițial rețelele neuronale recurente erau greu de antrenat, putând să ajungă la milioane de parametrii, arhitecturile dezvoltate în ultimii ani au avut performanțe uimitoare în sarcini foarte variate, precum: traduceri, generarea de etichete pentru imagini sau recunoașterea scrisului de mână [11]. Cu toate aceastea , nu este clar cum ar putea fii aplicate RNN pentru probleme ale căror date de intrare și ieșire au relații complicate non-monotone și diferă ca dimensiuni [22]. E important să nu uităm nici de problemele de antrenare ale rețelei.

1.3 Rețele neuronale LSTM

O RNN simplă este greu de antrenat pentru dependințe foarte lungi, din cauza problemei *vanishing* și *exploding gradients* [11]. Rețelele Long Short-Term Memory (LSTM) sunt cunoscute pentru capacitatea lor de a învăța dependințe temporale largi.

1.3.1 Componente

LTSM rezolvă problema *vanishing gradient* introducând un spațiu intermediar de memorie prin intermediul celulei de memorie, care dispune de un sistem format din 3 ”porți”, *forget gate, output gate* și *input gate*. Aceasta înlocuiește nodul simplu din RNN. În Figura 3 [17] este prezent modul de lucru al unei rețele LSTM. După cum reiese și din nume modelul disune de 2 tipuri de memorie. Memoria pe durată lungă (*long term* memory), de care dispune ți o RNN simplă, vine în forma ponderilor calculate la fiecare pas. Memoria de scurtă durată (*short-term* memory) este reprezentată de niște valori temorare care sunt trecute de la un nod la nodurile următoare. În continuare o să detaliez componentele unei celule dintr-o LSTM și rolul fiecăreia.

- *Nodul de intrare (Input node)* este nodul care ia datele de intrare standard x_t de la momentul curent și datele din stratul ascuns h_{t-1} din momentele anterioare. Deobicei, suma ponderilor este trecută printr-o funcție *tanh* , dar în lucrări mai vechi se poate observa folosirea funcției sigmoid [11]. Valoarea nodului de intrare o vom nota cu v_n.

- *Input gate,* precum nodul de intrare preia atât informația curentă , cât și cea anterioară, urmând ca acestora să le fie aplicată o funcție sigmoid. Denumirea de ”poartă” vine de la faptul că valoarea acesteia va fii înmulțită cu valoarea altui nod, astfel dacă valoarea este 0, fluxul celuilalt nod va fii întrerupt, iar dacă valoarea este 1, atunci toată informația este trecută la următorul nod. Valoarea pentru *input gate* o vom nota cu v_i.

- *Starea internă* este un nod cu o funcție de activare liniară ce are o muchie recurentă ce indică spre el însuși. Valoarea notată cu v_s a stării interne va fii : $v_s = v_n^t \odot v_i^t + v_s^{t-1}$.

- *Forget gate* pune la dispoziție un sistem prin care rețeaua învață să sorteze informația pe care o consideră neimportantă din starea internă. Atfel , valoarea stării interne va deveni conform formulei $v_s = v_n^t \odot v_i^t + v_f^t \odot v_s^{t-1}$, unde v_f este valoarea obținută de *forget gate.*

- *Output gate* calculează valoarea finală a celulei, înmulțind valoarea ei cu cea a stării interne calculate anterior.

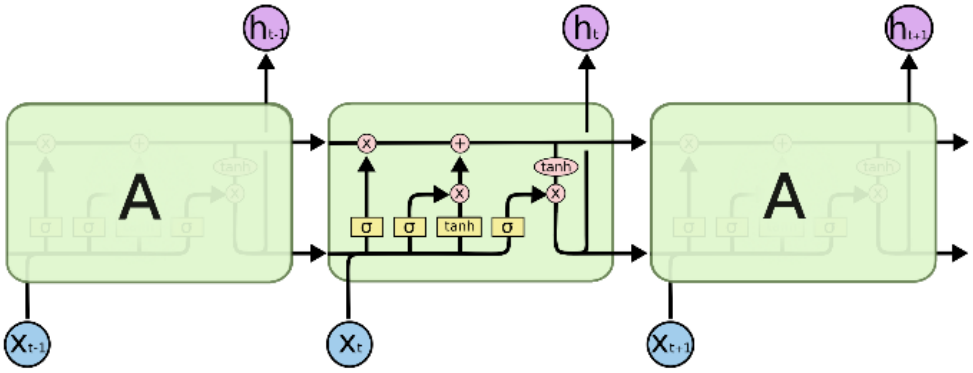

Figura 3. Structura unei rețele LSTM

1.3.2 Mod de funcționare

Mai întâi, informația anterioară h_{t-1} este concatenată cu datele de intrare curente x_i, după care sunt trecute prin *forget gate*. Aceasta decide prin intermediul unei funcții sigmoid ce informație poate sa fie „uitată", valorile mai apropaite de 1 sunt considerate relevante și sunt trecute mai departe, iar valorile mai apropiate de 0 sunt considerate nerelevante și "uitate". În continuare valorile concatenate trec prin *input gate,* care decide , tot prin intermediul unei funcții sigmoid , ce valori ar trebui modificate. În același timp, o funcție tanh generează valosrile candidat din $h_{t-1} \times x_i$. Valorile candidat se înmulțesc cu rezultatul obținut de la *input gate,* selectând valorile importante din candidat. Următorul pas este modificarea stării celulei c_i prin înmulțirea vechii stării cu rezultatele obținute de la cele 2 "porți" anterioare. Pasul final este calcularea stării ascunse, care va fii trimisă mai departe prin rețea. Acest lucru este realizat de *output gate*. Starea celulei actuale este normalizată, după care se înmulțește cu valorile candidat din $h_{t-1} \times x_i$, care a fost trecută printr-o funcție sigmoid. Astfel, se alege informația relevantă care ar trebui sa fie transmisă următoarei celule [11].

1.3.3 Aplicații

LSTM au numeroase aplicații în diferite domenii. Printre acestea se numără: generarea de text, recunoașterea scrisului de mână, traducerea automată, captarea imaginilor sau predicții, dar utilizabilitatea lor nu se oprește aici. Rezultatele atinse folosind astfel de rețele sunt multe și foarte fascinante [16].

CAPITOLUL 2
Modelul Markov cu stări ascunse

Introducere

Modelul Markov are o istorie largă, fiind considerat o noțiune de bază a matematicii și statisticii. Recunoșterea vorbirii (*speech recognition*) sau bioinformatica sunt unele dintre doemniile în care modelul Markov este foarte des utilizat. În acest capitol, voi defini componentele care alcătuiesc modelele ascunse markov , precum și modul în care acestea funcționează.

2.1 Modelul Markov

Un model Markov definește un sistem probabilistic prin intermediul stării sistemului și a timpului de observare. Atât timpul, cât și starea sistemuui pot să fie continue sau discrete. În cazul în care sistemul are starea discrete și timpul de observare continuu, se obține un process Markov [18].

2.1.1 Procese Markov

Considerăm un proces stocastic $(X_k)_{k\geq 0}$, unde fiecare X_k este o variabilă aleatoare din spațiul de probabilități $(\Omega, \mathcal{G}, P)$. X_k este starea modelului la momentul k. Se spune despre procesul $(X_k)_{k\geq 0}$ că are proprietatea Markov dacă:

$$P(X_{k+1} \in A \,|X_0, \dots, X_k) = P(X_{k+1} \in A \mid X_k)$$

Pentru ∀ A, k [25]. Altfel spus, proprietatea Markov asigură faptul că într-un proces, starea următoare depinde doar de starea prezentă , fără să țină cont de stările anterioare. De exemplu, legile fizicii asigură faptul că mișcarea unei particule într-un pas mic de timp este determinată doar de poziția sa curentă și de velocitatea sa, nu are importanță cum a ajuns în starea respectivă.

2.1.2 Lanțuri Markov

Distribuția probabilității condiționale a stării următoare depinde de starea curentă și nu de alte stări anterioare. Adică *s(t)* depinde doare de *s(t-1)* , unde *s(t)* este starea la momentul *t.* Acest caz reprezintă un model Markov de ordinul întâi. În general, dacă starea curentă depinde de ultimele *n* stări anterioare, avem un model markov de ordinul n.

O secvență de evenimente aleatoare ce respectă un model Markov reprezintă un lanț Markov. Aceste secvențe pot să fie cuvinte, etichete sau simboluri reprezentând orice, de exemplu vremea. Figura 4 [7] reprezintă un lanț Markov care asignează probabilitatea unei secvențe de evenimente meteorologice, pentru care vocabularul este definit de cuvintele HOT, COLD și WARM. Stările sunt reprezentate ca noduri în graf, iar tranzițiile ca și muchii, având fiecare o probabilitate cu care se trce dintr-o stare în alta. Suma valorilor muchiilor care ies dintr-un nod trebuie să fie 1.

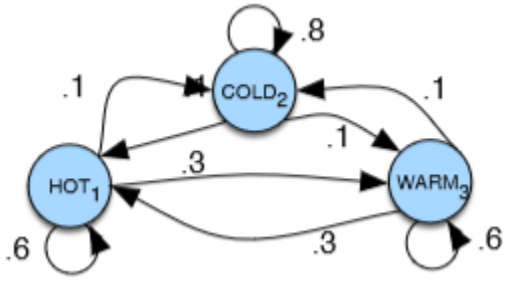

Figura 4. Lanț Markov pentru vreme

Un lanț Markov este specificat de următoarele componente [7]:

- un set de *N* stări, $Q = q_1 q_2 \dots q_N$
- o matrice $A = a_{11} a_{12} \dots a_{n1} \dots a_{nm}$ a probabilităților de tranziție, unde a_{ij} reprezintă probabilitatea de a trece din starea i în starea j, cu proprietatea că $\sum_{j=1}^{n} a_{ij} = 1 \, , \forall \, i$
- o distribuție inițială a probabilităților $\pi = \pi_1, \pi_2, \dots, \pi_N$, unde π_i reprezintă probabilitatea ca lanțul Markov să înceapă în starea i, pentru care $\sum_{i=1}^{n} \pi_i = 1$. Unele stări j pot să aibă $\pi_j = 0$, ceea ce înseamnă că aceastea nu pot să fie stări inițiale

2.2 Model Markov cu stări ascunse (HMM)

Un lanț Markov este folositor când vrem să calculăm probabilitatea pentru o secvență de evenimente observabile. Însă, în multe cazuri evenimentele sunt ”ascunse” ; nu le putem observa direct. Secvența de evenimente este un lanț Markov; starea următoare depinde doar de starea curentă. Funcția probabilistică pentru acest lanț este un proces stocastic generat de 2 mecanisme: un lanț Markov cu un număr finit de stări și un set de funcții aleatoare asociate cu fiecare stare. Pentru timp discret , se presupune că procesul este într-o anumită stare și o observație este generată de funcția corespunzătoare stării curente. Lanțul Markov își schimbă după starea conform matricii de probabilități de tranziție. Observatorul vede doar rezultatul obținut de funcția asociată stării și nu poate observa direct stările lanțului Markov; de aici provine și numele de stări *ascunse*.

Ideea de bază este faptul că HMM este un model finit care descrie o distribuție de probabilități pentru o infinitate de posibile secvențe.

HMM poate fii definit de următoarele componente [7]:

- un set de *N* stări, $Q = q_1 q_2 \dots q_N$
- o matrice $A = a_{11} a_{12} \dots a_{n1} \dots a_{nm}$ a probabilităților de tranziție, unde a_{ij}
- o secvență $O = o_1 o_2 \dots o_T$ de T observații, fiecare aparținând unui vocabular $V = v_1, v_2, \dots, v_V$
- o secvență de posibile observații $B = b_i(o_t)$, fiecare exprimând probabilitatea ca observația o_t să fie generată din starea i
- o distribuție inițială a probabilităților $\pi = \pi_1, \pi_2, \dots, \pi_N$, unde π_i reprezintă probabilitatea ca lanțul Markov să înceapă în starea i, pentru care $\sum_{i=1}^{n} \pi_i = 1$. Unele stări j pot să aibă $\pi_j = 0$, ceea ce înseamnă că aceastea nu pot să fie stări inițiale

Pentru un lanț Markov de ordinul I se pot face 2 presupuneri:

- **presupunerea Markov**: probabilitaea stării curente depinde doar de starea anterioară , $P(q_i | q_1 \dots q_{i-1}) = P(q_i | q_{i-1})$

- **independența rezultatului**: probabilitatea ca rezultatul unei observații o_i depinde doar de starea care produce observația q_i , $P(o_i|q_1 \dots q_i, \dots, q_T, o_1, \dots o_i, \dots, o_T) = P(o_i|q_i)$

Deobicei, folosind HMM se rezolvă 3 probleme fundamentale [5]. În primul rând, dându-se HMM și o secvență , vrem să aflăm probabilitatea ca HMM să genereze secvența sau pentru a determina secvența optimă de stări pentru a genera secvența. Alt exemplu ar fii, dacă se dă un număr mare de date și vrem să determinăm structura HMM care reprezintă cel mai bine datele. Acestea se mai pot numi și *problema de evaluare, problema de decodare* și , respectiv, *problema de antrenare.*

2.2.1 Problema de evaluare

Problema poate fii generalizată astfel: dându-se o HMM $\lambda = (A, B)$ [2] și o secvență de observat *O*, să se determine probabilitatea $P(O|\lambda)$. *A* este vectorul de tranziții , memorând probabilitatea stării *j* care urmează după starea *i* . Probabilitățile tranzițiilor d estare sunt independente de timp.

$$A = [a_{ij}], a_{ij} = P(q_t = s_j \,|q_{t-1} = s_i)$$

B reprezintă vectorul de observare care memorează probabilitatea ca observația *k* să fie produsă de starea *j* , independent de timp.

$$B = [b_i(k)] \,, b_i(k) = P(x_t = v_k \,|q_t = s_i)$$

Altfel spus, vrem să evaluăm cât de bine prezice modelul o secvență. Pentru o secvență $O = o_1, o_2, \dots, o_T$ și o secvență de stări $Q = q_0, q_1, \dots., q_T$, probabilitatea pentru secvența de observare este:

$$P(O|Q, \lambda) = \prod_{t=1}^{T} P(o_t|q_t, \lambda) = b_{q_1}(o_1) \times b_{q_2}(o_2) \dots b_{q_T}(T)$$

Probabilitatea pentru secvența de stări este:

$$P(Q|\lambda) = \pi_{q_1} a_{q_1 q_2} a_{q_2 q_3} \dots a_{q_{T-1} q_T}$$

Astfel se poate calcula probabilitatea observațiilor modelului folosinf formula:

$$P(O|\lambda) = \sum_{Q} P(O|Q, \lambda) P(Q|\lambda) = \sum_{q_1 \dots q_T} \pi_{q_1} b_{q_1}(o_1) a_{q_1 q_2} b_{q_2}(o_2) a_{q_2 q_3} \dots a_{q_{T-1} q_T}$$

Însă, complexitatea acestei formule crește exponențial față de *T*, dar se poate observa că există multe calcule redundante . Pentru a remedia acest fapt se folosește algoritmul *forward*. Acesta presupune implementarea unei unei structuri care seamănă cu o ”împletitură de nuiele” (fiecare nod care o legătură spre fiecare nod de pe următoarea coloană) pentru fiecare stare, care calculează suma tuturor stărilor la momentul anterior de timp, astfel ca în final, ultima coloană să fie echivalentă cu probabilitatea secvenței de observare. O schemă pentru acest model se poate observa în Figura 5 [2]. Valoarea sumei este notată cu α și reprezintă probabilitatea ca secvența parțială $o_1, o_2, \dots, o_t$ să fie generată din starea s_i la momentul t. Astfel, α este definit în felul următor:

$$\alpha_t(i) = P(o_1, o_2, \dots, o_T, q_t = s_i \,|\lambda)$$

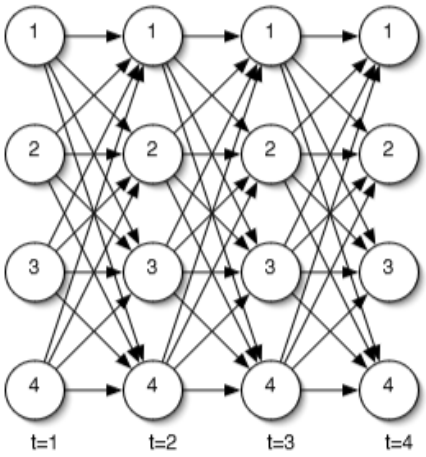

Figura 5: Model tip ”împletitură de nuiele”

Pașii algoritmului sunt următorii:

1. **Inițializarea**: $\alpha_1(i) = \pi_i b_i(o_1), 1 \le i \le N$.
2. **Recursivitatea**:

$$\alpha_{t+1}(j) = [\sum_{i=1}^{N} \alpha_t(i) a_{ij}]\, b_j(o_{t+1}) \,, 1 \le t \le T-1, 1 \le j \le N$$

Pentru fiecare stare s_j, $\alpha_j(t)$ memorează probabilitatea de a ajunge în starea respectivă, observând secvența obținută până în momentul *t*.

3. **Finalizarea**:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

2.2.1 Problema de decodare

Problema poate fii generalizată; pentru un HMM $\lambda = (A, B)$ și o secvență de observații $O = o_1, o_2, \dots, o_T$, vrem să aflăm cea mai probabilă secvență de stări $Q = q_1 q_2 q_3 \dots q_T$. [25] Altfel spus, se caută secvența de stări *Q* care are probabilitatea cea mai mare de a genera secvența de observare *O*. Cel mai comun algoritm folosit pentru rezolvarea problemei este *algoritmul Viterbi.* Acesta este similar cu algoritmul Forward de la subcapitolul anterior, diferența constă în faptul că la fiecare pas în loc de suma probabilităților, se calculează maximul, adică se ia starea cu probabilitatea cea mai mare din secvența parțială, care se definește astfel [2]:

$$\delta_t(i) = max_{q_1, q_2, \dots, q_{t-1}} P(q_1 q_2 \dots q_t = s_i, o_1, o_2 \dots o_t|\lambda)$$

Mai mult, algoritmul folosește de ”*backpoints*” , deoarece algoritmul calculează o probabilitate și o cale optimă.

Pașii algoritmului sunt următorii:

- **Inițializarea:** $\delta_t(i) = \pi_i b_i(o_1),\ 1 \le t \le T,\ \psi_1(i) = 0,$

  Unde: ψ_t reprezintă mulțimea de ”*backpoints*”, care va fii folosită în pasul 4 pentru a determina sevența optimă

- **Recusivitatea:**

$$\delta_t(j) = max_{1 \le i \le N}\left[\delta_{t-1}(i) a_{ij}\right] b_j(o_t), 2 \le t \le T, 1 \le j \le N,$$

$$\psi_t(j) = argmax_{1 \le i \le N}\left[\delta_{t-1}(i) a_{ij}\right], 2 \le t \le T, 1 \le j \le N$$

  Pentru fiecare moment de timp *t*, cunoscându-se probabilitatea de a fi în starea *t-1*, se calculează cea mai probabilă rută a stărilor care duce spre starea curentă. În același timp, se memorează câte un ”*backpoint”* pentru stările care fac parte din secvența dorită.

- **Finalizare:**

$$P^* = max_{1 \le i \le N}[\delta_T(i)]$$

$$q_T^* = argmax_{1 \le i \le N}[\delta_T(i)]$$

- **Backtracking pentru aflarea secvenței optime:**

$$q_t^* = \psi_{t+1}(q_{t+1}^*), t = T-1, T-2, \dots, 1$$

2.2.3 Problema de antrenare

Problema poate fii generalizată astfel: fiind cunoscută o secvență de observare *O* și un set de stări HMM posibile, să se învețe parametrii *A* și *B*.

În funcție de felul în care sunt oferite datele de antrenare, există 2 abordări ale problemei: supervizată, când datele conțin și rezultatele așteptate, și nesupervizată, când nu cunoaștem rezultatul pe care ar trebui să îl obținem.

Cea mai simplă abordare supervizată pentru a determina modelul λ este de a porni de la un set foarte mare de date de antrenare etichetate. De exemplu, se pot defini 2 seturi de etichete [2]:

- $t_1 \dots t_N$ este setul de etichete pe care le alipim stărilor $s_1 \dots s_N$ din HMM
- $w_1 \dots w_M$ este setul de cuvinte pe care le alipim setului de observații $v_1 \dots v_M$

Folosind acest sistem de etichetare, putem afla parametrii modelului λ folosind următoarele formule:

- **Matricea de tranziții**

$$a_{ij} = P(t_i | t_j) = \frac{Count(t_i, t_j)}{Count(t_i)}$$

Unde $Count(t_i, t_j)$ reprezintă de câte ori t_j a urmat după t_i în datele de antrenamet

- **Matricea de observare**

$$b_j(k) = P(w_k | t_j) = \frac{Count(w_k, t_j)}{Count(t_j)}$$

Unde $Count(w_k, t_j)$ reprezintă de câte ori w_k a fost etichetat cu t_j în datele de antrenament

- **Probabilitățile inițiale**

$$\pi_i = P(q_1 = t_i) = \frac{Count(q_1 = t_i)}{Count(q_1)}$$

Pentru abordarea nesupervizată, algoritmul standard folosit este algoritmul ”*forward-backword*” sau algoritmul **Baum-Welch**. Algoritmul estimează iterativ probabilitățile inițiale pentru tranziții și observații , după care le folosește pentru a genera probabilități derivate din ce în ce mai bune. Acest lucru se realizează calculând, prima dată, probabilitatea ”*forward*” , împărțind-o după pe toate drumurile care au contribuit la calcularea acestei probabilități. Astfel, se definește probabilitatea ”*backward”* β, care reprezintă probabilitatea de a găsi observații de la momentul *t+1* la final. Presupunând că ne aflăm în starea *i* în momentul *t* , formula probabilității este următoarea [25] :

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, \dots, o_T | q_t = i, \lambda)$$

Pașii pentru a calcula probabilitatea ”*backward”* :

1. **Inițializarea:**

$$\beta_T(i) = 1 \, , 1 \leq i \leq N$$

2. **Recursivitatea:**

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), \;\; 1 \leq i \leq N, 1 \leq t < T$$

3. **Finalizarea:**

$$P(O|\lambda) = \sum_{j=1}^{N} \pi_j b_j(o_1) \beta_1(j)$$

Estimarea probabilității de tranziție a_{ij} se realizează ușor împărțind numărul estimate de tranziții din *i* în *j* la numărul estimate de tranziții din *i*. Pentru a calcula numărul de tranziții

din *i* în *j* se definește o probabilitate ξ_t care e probabilitatea de a fi în starea *i* la momentul *t* și în starea *j* la momentul *t+1*. Formula pentru această probabilitate este următoarea:

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)}$$

Numărul estimat de tranziții din starea *i* în starea *j* se calcuează făcând suma probabilităților ξ_t , cu $t = 1:T-1$. Numărul de tranziții din starea *i* se calculează realizând suma probabilităților $\xi_t(i,j)$ din toate tranzițiile stării. În final formula pentru *a* va arăta în felul următor:

$$a_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N}\xi_t(i,k)}$$

Pasul următor este să se calculeze probabilitatea de observare a unui simbol v_k din vocabular în starea *j*. Aceasta este câtul împărțirii numărului estimate de câte ori eram în starea *j* și observam simbolul v_k la numărul estimate de câte ori eram în starea *j*. Din nou trebuie calculată probabilitatea de a fi în starea *j* la momentul *t*, care se notează cu $\gamma_t(j)$. Astfel , numărătorul se calculează ca sumă din $\gamma_t(j)$ pentru fiecare moment $t = 1:T-1$ în care observația o_t este simbolul v_k. Numitorul va fii pur și simplu sumă din $\gamma_t(j)$ în toate momentele $t = 1:T-1$. În final, formula pentru *b* va arată în felul următor:

$$b_j(v_k) = \frac{\sum_{t=1 s.t. o_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}$$

Cunoscând acești termini, se pot reestima probabilitățile de tranziție *A* și de observare *B* dintr-o secvență *O*. Aceste reestimări stau la baza algoritmului "*forward – backward"*. Pe scurt, modul de lucru al algoritmului Baum-Welch începe cu o aproximare initial a parametrilor HMM $\lambda = (A,B)$, după care se calculează probabilitățile ξ_t și γ_t, urmând ca în pasul următor să se folosească aceste două probabilități pentru a recalcula *A* și *B*.

CAPITOLUL 3
Abordări folosite în analiza textelor

Introducere

Oamenii au început să scrie informații pe hârtie acum mii de ani. În acest timp, creierul a câștigat foarte multă experiență în înțelegerea limbajului. Putem să citim un articol și să înțelegem ce înseamnă de fapt, putem să compunem un mesaj de la 0 sau să selectăm informația relevantă dintr-un mesaj. Toate aceste lucrui se încearcă să fie automatizate cu ajutorul inteligenței artificiale, iar motivele pentru care am dori asta sunt diverse. Prin procesarea unui text , calculatorul poate să proceseze limbajul nostrum natural. Astfel, un calculator poate să învețe informația relevantă dintr-un text și să ofere un rezumat rapid, să facă o traduceri dintr-o limbă în alta sau să ofere sugestii de completare a unui mesaj. Toate aceste abilități reușesc să facă viața de zi cu zi a fiecărui om mai ușoară și comunicarea cu societatea mai simplă și mai rapidă. În acest articol, o să mă axez , în principal , pe modelarea limbajului și predicția de texte. O să prezint câteva modele din literatură care și-au propus să acopere aceste domenii, folosind diferite rețele neuronale și obținând rezultate relevante.

3.1 Rețele neuronale recurente

Rețelele neuronale recurente (RNN) au arătat rezultate promițătoare în diferite sarcini ce necesită învățare automată, mai ales atunci când numărul datelor de intrare sau ieșire variază, după cum este cazul în modelarea limbajelor. Ce este interesant de observat este faptul că majoritatea succeselor recente nu s-au realizat cu o rețea neuronală recurentă tradițională, ci , mai degrabă, cu o rețea sofisticată cu noduri ascunse, precum *long short-term memory* (LSTM) [3]. Acestea au aplicații mai ales în recunoașterea vorbirii, generarea de text și traduceri automate.

În continuare , o să prezint abordarea folosită de Kim Yoon et al, pentru predicția textului caracter cu caracter. Modelul propus de ei folosește 2 rețele neuronale: o rețea neuronală de convoluție (CNN) pentru procesarea intrărilor la nivel de caracter și o rețea neuronală recurentă LSTM, care primește rezultatele obținute de CNN [8].

3.1.1 Arhitectura modelului

Deși majoritatea experimentelor realizate pentru modelarea limbajului natural combinau *word embeddings* ca intrare cu caracteristici ale analizei caracter cu caracter, modelul propus de Kim Yoon et al folosește ca intrare, rezultatul obținut la nivel de caracter de o CNN cu un singur strat și cu *max-over-time pooling*, acest fapt reducând foarte mult numărul de parametrii.

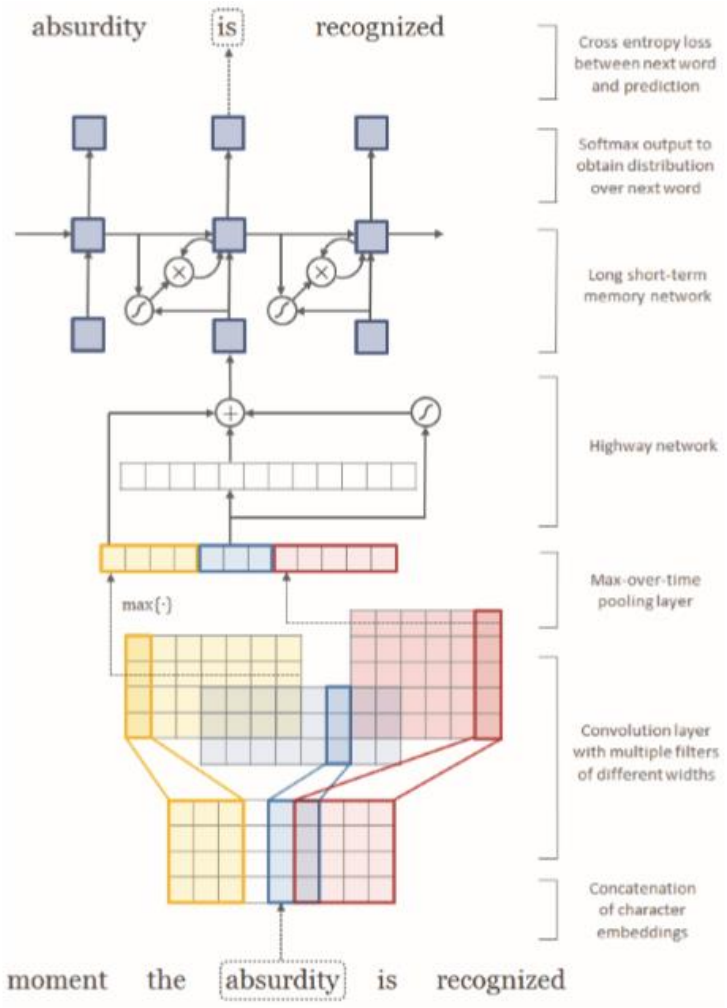

Figura 6. Arhitectura modelului aplicată pe un exemplu concret

Arhitectura modelului este prezentată în Figura 6. Aici modelul ia cuvântul *absurdity* ca și date de intrare la momentul current și îl combină cu informația învățată anterior pentru a prezice următorul cuvânt *is*. Primul strat realizează o căutare printer seturile de caractere și le aranjează într-o matrice C^k. După, CNN aplică operații între C^k și mai multe matrici de filtrare . În

exemplul current se pot observa 12 astfel de filtre : 3 de lățime 2 (albastru), 4 de lățime 2 (galben) și 5 de lățime 4 (roșu). În următorul pas, operația de *max-over-time pooling* este aplicată pentru a obține o reprezenare de dimensiune fixă a cuvântului, care este trimisă unei rețele intermediare *highway* , care are rolul de optimizare. Rezultatele sunt folosite ca și date de intrare pentru LSTM. În final, se mai aplică o transformare afină și o funcție softmax pentru normalizarea datelor pentru a obține distribuția următorului cuvânt.

3.1.2 Modelarea limbajului folosind RNN

Rețelele recurente neuronale sunt un tip de rețele potrivite pentru procesarea secvențelor. La fiecare pas t , o RNN primește x_t, care este starea neuronului la momentul t și starea ascunsă h_{t-1}, care reprezintă informația strânsă până în momentul curent.

Pentru modelul prezentat în [8], se presupune un vocabular de cuvinte V de dimensiune fixă, modelarea specifică o distribuție peste w_{t+1} , fiind cunoscută secvențele anterioare $w_{1:t}$ = $[w_1, \dots, w_t]$. Un model recurent realizează aceasta aplicând o transformare afină stratului ascuns, urmată de o normalizare softmax.

Având în vedere că $w_{1:t}$ = $[w_1, \dots, w_t]$ este o secvență de cuvinte, antrenamentul rețelei presupune minimizarea funcției *likelihood* (Figura 7 [8]) a secvenței. Acest lucru este deobicei realizat cu propagare in timp (*backpropagation through time).*

$$NLL = -\sum_{t=1}^{T} \log \Pr(w_t | w_{1:t-1})$$

Figura 7. Formula funcției likelihood

3.1.3 CNN la nivel de caracter

În [8], intrările în momentul t sunt rezultatele obținute de CNN la nivel de caracter. Modelul folosește *character embeddings*, față de alegerea populară de a folosi *word embeddings.* Acest lucru este avantajos, deoarece orice cuvânt se poate forma, chiar dacă nu face parte din vocabular, cât timp caracterele există în *embedding.* Un alt benefiu este faptul că poate să

potrivească cuvinte scrise greșit , emoticoane și se descurcă mai bine cu cuvinte cu frecvență rară [13].

Reprezentarea la nivel de caracter a unui cuvânt k format din secvența de caractere $[c_1, \dots, c_l]$ este data de matricea $C^k \epsilon \mathbb{R}^{d\ x\ l}$, unde coloana j conține un *character embedding* asociat lui c_j.

În continuare se aplică *narrow convolution* între C^k și un filtru $H \epsilon \mathbb{R}^{d\ x\ w}$ și *max-over-time,* pentru a capta caracteristica cea mai importantă (cea cu valoarea cea mai mare) pentur fiecare filtru. În esență, un filtru alege o n-gramă în care dimensiunea corespunde cu lățimea filtrului. Acesta este procesul descris pentru un singur filtru, dar modelul abordat folosește mai multe filtre, de lățimi diferite pentru a obține vectorul pentur cuvântul *k*. Dacă avem un total de *h* filtre, atunci $y^k = [y_1^k, \dots, y_h^k]$ este reprezentarea cuvântului k , ce va avea rolul de dată de intrare pentru LSTM. În mod normal, pentru procesarea limbajului natural, h se alege din intervalul [100, 1000].

3.1.3 Rețele neuronale highway

Datele de intrare pentru modelul propus sunt repzentările y^k formate anterior de CNN, însă Kim Yoon et al au observant îmbunătățiri adâugând o rețea intermediară care procesează y^k înainte de intrarea în LSTM. Acea rețea intermediară este o rețea neuronală *highway,* care a fost recent introdusă de Srivastava et al.

Ideea de a implementa acest nou tip de rețea a pornit de la faptul că este dificil de antrenat o rețea cu foarte multe straturi (de ordinal sutelor). Numele de rețele *highway* (autostradă) vine de la faptul că arhitectura lor permite informației să treacă prin mai multe straturi pe ceea ce cercetătorii numesc *information highways* (autostrăzi de informație) suferind doar modificări minore. Asemănător cu LSTM și aceste rețele dispun de 2 ”porți”, *transform gate* și *carry gate.* Acestea adaugă 2 transformări neliniare, față de o rețea simplă *feedforward,* având rolul de a exprima ce procent din rezultat este produs din transformarea datelor de intrare și, respectiv, din transportarea acesteia , de unde provin și numele *transform gate* (transformare) și *carry gate* (transportare). Așa cum, într-o rețea simplă, straturile sunt alcătuite din unități de calcul, așa au

creat autorii conceptul de bloc pentru rețelele *highway*. Pentru blocul *i* se descriu starea blocului *Hi(x)* și rezultatul obținut prin *transform gate Ti(x).* Astfel, rezultatul produs de un bloc y_i are formula următoare: $y_i = H_i(x) * T_i(x) + x_i * (1 - T_i(x))$ [21].

În Figura 8 [23], este prezentat modul de funcționare al rețelei. *H* reprezintă o funcție de transformare urmată de o funcție de activare, *T* este funcția de transformare din *transform gate* , iar *C* este funcția de transportare din *carry gate*. Un caz particular este ca *C = 1 – T*.

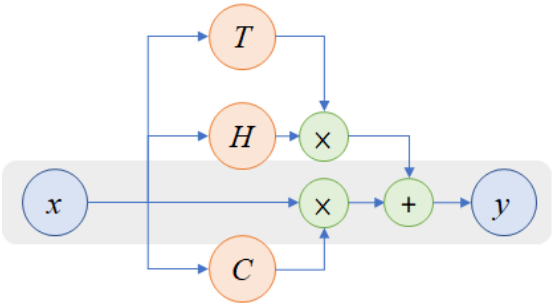

Figura 8. Circuit de funcționare al rețelei neuronale highway

3.1.4 Rezultate

La testarea modelului s-au folosit 2 seturi de date, unul mai mic (DATA-S) și unul mai mare (DATA-L), pentru fiecare limbă aleasă. În Tabelul 1 [8], sunt trecute limbile alese și dimensiunile datelor folosite. |*V*| este dimensiunea vocabularului de cuvinte, |*C*| este dimensiunea vocabularului de caractere, iar *T* este numarul de simboluri din setul de antrenament. Datele pentru limba engleză au fost luate din *Penn Treebank*, pentru limba arabă de la *News-Commentary corpus*, iar restul sunt luate de la *2013 ACL Workshop on Machine Translation*.

	DATA – S			DATA - L		
	|*V*|	|*C*|	*T*	|*V*|	|*C*|	*T*
Engleză (EN)	10.000	51	1 mil.	60.000	197	20 mil.
Cehă (CZ)	46.000	101	1 mil.	206.000	195	17 mil.
Germană (DE)	37.000	74	1 mil.	339.000	260	51 mil.
Spaniolă (ES)	27.000	72	1 mil	152.000	222	56 mil.
Franceză (FR)	25.000	76	1 mil.	137.000	225	57 mil.

Rusă (RU)	62.000	62	1 mil.	497.000	111	25 mil.
Arabă (AR)	86.000	132	4 mil.	-	-	-

Tabel 1. Dimensiunea datelor de intrare pentru testare

În Tabelul 2 [8], sunt prezentate arhitecturile pentru modelele folosind cele 2 seturi de date. Aici, *d* este dimensiunea pentru *character embeddings, w* este lățiea filtrelor, *h* este numărul de matrici filtru, dată ca și o funcție pe baza lățimii, *f,g* sunt funcții nonliniare, *l* este numărul de straturi și *m* este numărul de unități ascunse.

		Dimensiune mică	Dimensiune mare
CNN	*d*	15	15
	w	[1,2,3,4,5,6]	[1,2,3,4,5,6,7]
	h	[25 * *w*]	[min{200,50 * *w*}]
	f	Tanh	Tanh
Highway	*l*	1	2
	g	ReLU	ReLU
LSTM	*l*	2	2
	m	300	650

Tabel 2. Arhitectura modelelor pentru testare

În modelarea limbajelor, perplexitatea (PPL) este o măsurătoare care determină cât de bine un model prezice datele de testare. Un model bun este acela care e ce mai puțin ”confuz” când îi sunt prezentate noi date, adică cel cu perplexitatea mai mică.

Pentru a se observa mai clar eficiența modelului propus, se face o comparație cu alte modele existente în literatură. Mai mult, Kim Yoon et al, compară 2 modele LSTM implementate, unul cu 200 de unități ascunse (LSTM-Small) și unul cu 650 de unități ascunse (LSTM-Large). Pentru fiecare dintre acestea s-au folosit, pe rând, ca date de intrare cuvinte (LSTM-Word) sau caractere (LSTM-Char). În Tabelul 3 sunt prezentate rezultatele obținute pe setul de date pentru limba engleză, comparate cu alte modele care au folosit același set de date din *Penn Treebank*. Coloana *Dimensiune* se referă la numărul aproximativ de parametrii al

modelului. Aici am trecut doar câteva dintre modelele din literatură comparate deoarece aveau rezultate asemănătoare.

	PPL	Dimensiune
LSTM-Word-Small	97.6	5 mil
LSTM-Char-Small	92.3	5 mil
LSTM-Word-Large	85.4	20 mil
LSTM-Char-Large	78.9	19 mil
KN-5 (Mikolov et al, 2012)	141.2	2 mil
RNN (Mikolov et al, 2012)	124.7	6 mil
Deep RNN (Rascanu et al, 2013)	107.5	6 mil
genCNN (Wang et al, 2015)	116.4	8 mil
LSTM-1 (Zaremba et al, 2014)	82.7	20 mil
LSTM-2 (Zaremba et al, 2014)	78.4	52 mil

Tabel 3. Rezultate pentru setul de date din Penn Treebank

Se poate observa că modelul cu mai puține unități ascunse propus de Kim Yoon et al depășește ca și performanță modelele cu aceleași dimensiuni existente, iar cel cu mai multe unități obține aproximativ aceeași perplexitate ca și literatura existentă (Zaremba et al, 2014), cu toate că ca cu aproximativ 60% mai puțini parametrii.

Pentru celelalte limbi, s-a făcut o comparație cu modelul dezvoltat de Botha (2014), care folosește un model KN-4 și un model MLBL., mai puțin pentur limba arabă, unde autorii au antrenat un model KN-4. KN-4 este un model Kneser-Ney, care nu folosește o rețea neuronală , iar MLBL este un model morfologic biliniar. În Tabelul 4 și Tabelul 5 se pot observa rezultatele obținute pentru setul de date de dimensiune mic, respectiv cel de dimensiune mare. Word , Morph și Char sunt modele care primesc ca și input cuvinte,morfeme, și respectiv caractere .

		DATA-S					
		CZ	DE	ES	FR	RU	AR
Botha	KN-4	545	366	241	274	396	323

	MLBL	465	296	200	225	304	-
200 unități ascunse	Word	503	305	212	229	352	216
	Morph	414	278	197	216	290	230
	Char	401	260	182	189	278	196
650 unități ascunse	Word	493	286	200	222	357	172
	Morph	398	263	177	196	271	**148**
	Char	**371**	**239**	**165**	**184**	**261**	**148**

Tabel 4. Rezultatele perplexității pentru setul mic de date

		DATA-L					
		CZ	DE	ES	FR	RU	AR
Botha	KN-4	862	463	219	243	390	291
	MLBL	643	404	203	227	**300**	273
200 unități ascunse	Word	701	347	186	202	353	236
	Morph	615	331	189	209	331	233
	Char	**578**	**331**	**169**	**190**	313	**216**

Tabel 5. Rezultatele perplexității pentru setul mare de date

Este evident din Tabelul 4 că modelul bazat pe caractere are cea mai mare performanță dintre toate celelalte modele prezentate.

Din cauza constrângerilor de memorie, a fost antrenat doar modelul LSTM mai mic (Tabelul 5). Și aici se poate observa superioriatea modelui bazat pe caractere, deposit doar cu puțin la limba rusă de modelul MLBL al lui Botha.

3.2 Completarea automată a codului

Modelele pentru procesarea limbajului se folosesc și pentru completarea de cod. În acest domeniu modelele tradiționale statistice nu sunt considerate cele mai potrivite, deoarece nu reușesc să acopere la fel de bine mulțimea mare de simboluri necunoscute din variabile și nume de funcții după cum au realizat Subhasis și Chinmayee [4]. Ei au încercat diferite modele care s-

au dovedit mai eficiente, folosind vector de cuvinte pentru modelarea simbolurilor din cod , după cum este prezentat în [15] ,și tehnici de învățare bazate pe rețele neuronale.

În [4] , datele de antrenament și de testare sunt împărțite 50 – 50 și dându-se o secvență de cod, se prezice următorul simbol. Secvențele se aleg aleator dintre liniile complete din setul de date pentru antrenament , simbolul imediat următor fiind cel considerat corect că ar urma în secvența. În continuare , voi prezenta metoda de modelare a simbolurilor și câteva dintre metodele de învățare abordate de Subhasis Das și Chinmayee Shah.

3.2.1 Modelarea simbolurilor

Primul pas în modelarea simbolurilor este să se construiască un dicționar cu toate simbolurile sau cuvintele care apar în cod, care va ajuta la prezicerea următorului simbol. Pentru aceasta , similar cu metoda prezentată pentru modelele Markov, se citesc toate liniile din fișier și sunt trecute în dicționar seturile de simboluti alfanumerice și spațiu, seturi de caractere speicale, spațiu și simbolurile pentru a marca o linie nouă. De exemplu, pentru linia de cod: `for (i = 0; i < n; i++) {` vor fii trecute în dicționar următoarele: `for`, `i`, `=`, `0`, `;` , `i`, `<`, `n`, `;` , `i` , `++) {`. E de remarcat faptul că separarea nu respectă neapărat limbajul, astfel `++) {` este tratat ca un singur simbol, deși , în mod normal , conține 3.

După separarea tuturor datelor, se poate observa că dicționarul nu este complet, deoarece pot apărea simboluri noi în cod care încă nu au fost introduse. De asemenea, în codul folosit pentru antrenament pot fii anumite denumire specifice doar acelor fișiere , care nu vor mai apărea deloc. Pentru a rezolva această problemă, autorii articolului au ales să împartă simbolurile în 2 categorii:

a) *Simboluri cheie.* Simbolurile care apar frecvent vor fii catalogate ca și simboluri cheie. În principal, aceste simboluri vor fii simboluri specifice limabjului de programare în care este scris codul.
b) *Simboluri de poziție.* Simbolurile mai rar întâlnite, de exemplu numele de variable sau de funcții, vor fii categorizate ca și simboluri de poziție. Numele lor vine de la faptul că în cod orice simbol care nu este simbol cheie este înlocuit cu un șir de caractere de forma POS_TOK_ii , unde ii este poziția simbolului în secvență. Astfel, pentru exemplul de cod

dat mai sus s-ar realiza următoarea înlocuire: `for, POS_TOK_2, =, 0, ; , POS_TOK_2, <, n, ; , POS_TOK_2 , ++) {`. În acest fel, dacă apare un simbol de poziție într-o secvență ca cea de mai sus, se va înlocui șirul de caractere cu simbolul de poziție corespunzător. Dacă simbolul nu a mai apărut până în momentul respectiv în secvență , o să fie etichetat cu UNKNOWN (NECUNOSCUT).

3.2.2 Metode de învățare

Subhasis Das și Chinmayee Shah au exemplificat , în [4] , 5 dintre modelele pe care le-au încercat. Dându-se K + W simboluri cheie și de poziție, și cele K + W + 1 simboluri de ieșire s-au construiit modele care au la baza un vector de cuvinte, notați cu v_i , unde $1 \leq i \leq K + W$ [15]. Pasul următor este , ca pentru un set de simboluri $[t_1, t_2, \dots, t_W]$, să se calculeze un scor pentru fiecare simbol j de ieșire. Acest scor este reprezentat printr-o funcție: $s_j = f_j(v_{t_1}, v_{t_2}, \dots, v_{t_W})$. Eroarea pentru această metodă este definită prin funcția: $L = \log(\frac{e^{s_{t_0}}}{\sum_j e^{s_j}})$. În continuare, o să prezint 2 dintre modelele abordate și rezultatele obținute

a) Modelul ***feed-forward with soft attention*** presupune folosirea unei rețele neuronale feed-forward,care are asociată fiecărei poziții din vectorul de cuvinte rezultat din modelarea simbolurilor o pondere cu valoare între 0 și 1. Aceasta este rezultatul obținut aplicând funcția sigmoid asupra unei combinații liniare a vectorilor concatenați.

Rețelele feed-forward sunt un element esențial în deep learning. Numele acestora provine de la faptul că informația este transmisă prin intermediul rețelei, de la datele de intrare până la cele de ieșire. Dacă acest model s-ar extinde să conțină conexiuni de feedback în sensul opus parcurgerii normale, atunci ar deveni rețele neuronale recurente [24]. Fiecare neuron calculează suma ponderilor neuronilor de intrare, urmând să aplice o funcție de activare pentru a normaliza această sumă. Pentru acest caz particular, funcția folosită este funcția sigmoid.

Modul de funcționare al acestor rețele este după cum urmează: datele de antrenament sunt transmise prin rețea, obținându-se un rezultat. Acesta este comparat cu rezultatul așteptat,

calculându-se o eroare. În continuare se aplică algoritmul backpropagation; eroarea calculată este propagată înapoi în rețea și este folosită pentru a actualiza ponderile fiecărui neuron, astfel încât eroarea să devină cât mai mică. Măsura cu care se modifică fiecare pondere se numește rată de învățare și aceasta este o constată pozitivă [26].

Subhasis și Chinmayee extind acest model, adăugând soft attention. Ideea de a include „atenție" în antrenarea rețelelor neuronale a devenit foarte populară recent , permițând modelelor să realizeze o legătură între 2 entități, de exemplu o imagine și descrierea sa. Luong Minh-Thang et al au realizat un model pe bază de atenție pentru o mașină neuronală care să realizeze traducerea unui text din engleză în română. Experimentând cu 2 tipuri de atenție: globală, care ia în considerare toată stările codificatorului, și locală, care se concentrează pe o porțiune mai mică, au obținut rezultate cu 5.0 puncte BLEU mai multe decât alte modele fără atenție [12]. Rafel și Daniel au implementat un model pentru a rezolva problemele de „adunare" și „multiplicare" din LSTM, care s-a dovedit că poate să ofere rezultate foarte bune, atât pentru secvențe de mărime fixă, cât și pentru secvențe de mărime variabilă, într-un timp foarte scurt (254 secunde pentru modelul bazat pe atenție și 917 secunde pentru rețeaua neuronală recurentă) [19].

Pentru acest exemplu în particular, modelul va lua forma următoare [4]:

$$
\begin{aligned}
u &= [v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}] \\
a &= \sigma(Au) \\
z &= [a_1 v_{t_1}; a_2 v_{t_2}; a_3 v_{t_3}; \ldots; a_W v_{t_W}] \\
s_j &= \text{NL-3}(z; Q_1, Q_2, Q_3, p_j) \\
L &= \log\left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}}\right)
\end{aligned}
$$

Unde NL-k(z;Q1,Q2,Q3,pj) este o funcție care merge de la *u* la s_j pentru un model *feed-forward neural network* prezentat în [4], în care *k* reprezintă numărul de straturi neliniare conținute în model; a este atenția calculată; *z* reprezintă concatenarea vectorilor de cuvinte, la care s-au aplicat atențiile calculate.

b) Un alt model încercat a fost unul bazat pe ***rețele neuronale de tip GRU (Gated Recurrent Unit)***. Față de rețelele neuronale clasice, acestea pot decide ce informație ar trebui să fie transmisă spre rezultat. Totodată, pot să fie antrenate să păstreze informații pentru o perioadă

foarte lungă de timp sau să uite informație care nu este relevantă pentru predicție. Rețelele GRU sunt foarte asemănătoare cu cele LSTM, deoarece sunt similare ca design și produc rezultate la fel de bune [9].

Rețelele GRU dispun de 2 porți de intrare, *update gate* și *reset gate*. În update gate se decide cât din informația curentă se va transmite la timpul t+1. În reset gate se decide cât din informația din trecut poate fii „uitată”. Modul de funcționare al acestor rețele, în general, este precum în Figura 9 [9].

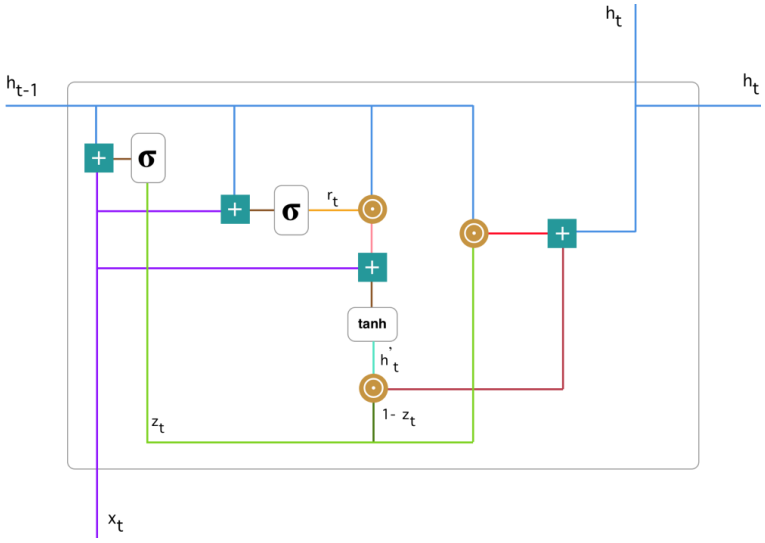

Figura 9. Rețea neuronală recurentă GRU

În primul pas, intrarea x_t este concatenată cu informația din cele $t-1$ celule anterioare, notată h_{t-1}, fiecare fiind înmulțită cu ponderea sa specifică. Trecând prin *update gate*, li se aplică o funcție de activare sigmoid cu ajutorul căreia se decide câtă din informația inițială ar trebui pasată mai departe. În pasul 2, aceleași valori trec prin *reset gate*, aplicându-li-se funcția de activare sigmoid pentru a decide cât din informație se uită. Valorile mai apropiate de 1 se păstrează, iar cele apropiate de 0 se uită. Pasul următor constă în construirea unei noi celule de memorie h'_t, ce se folosește de informația primită de la reset gate pentru a stoca informația relevantă de până acum, folosind funcția *tanh*. Pasul final este de construi memoria finală h_t, care o să fie trecută în continuare prin rețea. Aceasta selectează din informațiile trecute h_{t-1} și din memoria curentă h'_t datele relevante folosind rezultatul obținut de update gate [9].

Mai specific, în [4] , modelul a avut forma următoare:

$$[g_1; g_2; g_3; \ldots; g_W] = \text{GRU}([v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}])$$
$$s_j = p_j^T[g]$$
$$L = \log\left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}}\right)$$

Unde, g_i reprezintă datele de ieșire pentru a i-a celulă GRU.

1.2.3 Rezultate

În această secțiune , voi prezenta rezultatele obținute folosind cele 2 metode descrise anterior. Acuratețea fiecărui model poate fii văzută în tabelele de mai jos.

Metodă,DimFereastră,#Cheie	**Acuratețe cunoscută**	**Acuratețe absolută**	**Acuratețe chei**	**Acuratețe poziție**	**Top 3**
Atenție, 40, 1000	67.5	56.0	79.9	48.4	83.6
GRU,-,-	-	-	-	-	-

Test de acuratețe (%) proiect Linux

Metodă,DimFereastră,#Cheie	**Acuratețe cunoscută**	**Acuratețe absolută**	**Acuratețe chei**	**Acuratețe poziție**	**Top 3**
Atenție, 40, 500	47.9	40.6	64.6	17.6	66.5
GRU,40, 500	41.6	35.2	64.4	4.9	59.1

Test de acuratețe (%) folosind biblioteca Twisted din Python

În prima coloană a tabelului sunt trecute metoda, dimensiunea ferestrei și numărul de cuvinte cheie. Testul proiectului Linux nu are rezultatele raportate pentru GRU , deoarece s-au obținut performanțe foarte slabe.

Acuratețea cunoscută este acuratețea pentru pedicțiile în care următorul simbol este un simbol cheie sau un simbol de poziție din fereastra considerată. Acuratețea absolută este

acuratețea pentru toate cazurile , incluzând cazurile în care următorul simbol nu este nici simbol cheie, nici un simbol de poziție deja cunoscut. Acuratețe chei este acuratețea cu care se prezic corect simboluri cheie, iar acuratețe poziție este acuratețea cu care se prezic corect simboluri de poziție. Top 3 reprezintă procentajul cazurilor când următorul simbol se află între primele 3 predicții, ignorând cazurile când apar variabile sau nume de funcții noi.

Se poate observa că că modelul ***feed-forward with soft attention*** a dat rezultate mai bune decât cel folosind GRU. Chiar mai mult, în [4], s-a ajuns la concluzia că acest model a performat mai bine decât toate celelalte modele încercate. Acest lucru se datorează, în special , genericității modelului și a factorului de atenție. Se mai poate observa și că acuratețea pentru simbolurile de poziție este mai mare pentru Linux, un proiect C , decât pentru Python. Însă , predicția pentru top 3 este ridicată pentru ambele, ceea ce arată că programul învață anumite șabloane de limbaj pe care le folosește pentru a obține predicții cât mai bune. De exemplu, atunci când o variabilă apare pentru prima dată (este inițializată) înaintea unei instrucțiuni `if` , programul prezice că această variabilă va fii folosită în interiorul condiției.

CAPITOLUL 4
Aplicație practică:
SpeedyTalk

Introducere

În acest capitol voi descrie aplicația practiă cu scopul de a studia aplicabilitatea modelului Markov în problema predicției automate de teste. Aplicația a fost denumită speedyTalk, fiind o aplicație tip email, care facilitează utilizatorul în scrierea de mesaje prin autocompletarea propozițiilor. Se vor prezenta modul de dezvoltare, diverse diagrame, detalii legate de tehnologiile folosite și un scurt manual de utilizare. De asemenea, tot aici se regăsesc rezultatele experimentale.

4.1 Dezvoltarea aplicației

Aplicația SpeedyTalk este destinată tuturor persoanelor care doresc să beneficieze de o metodă de comunicare ușoară și rapidă cu alte persoane. Aplicația ar putea folosită atât de corporații, cât și de personae pentru comunicarea de zi cu zi, datorită interfeței prietenoase și simple.

Aplicaţia a fost dezvoltată urmând etapele specifice procesului de dezvoltare a unui produs software şi anume: definirea şi specificarea problemei, analiza şi proiectarea, implementarea şi testarea.

4.1.1 Definire şi specificare

Pentru dezvoltarea aplicației am antrenat un model Markov cu stări ascunse care să fie capabil să prezică textul dorit de utilizator în timp ce acesta compune mesajul. Modelul continuă să își modifice stările și tranzițiile cu fiecare mesaj scris de utilizator, astfel încât să învețe mai bine șabloanele cele mai des folosite de către utilizator.

4.1.2 Analiză și proiectare

Aplicația permite utilizatorului să navigheze pe paginile aplicației, să își vizualizeze mesajele primite și mesajele trimise și să scrie și să trimită un mesaj nou, specificând numele utilizatorului destinatar, subiectul mesajului și un text pentru mesaj. La scrierea mesajului, utilizatorul va fii asistat de un system de autocompletare a textului care îi va sugera completări.

În continuare, voi prezenta câteva diagrame , care ilustrează interacțiunea utilizatorului cu sistemul și a componentelor între ele.

4.1.2.1 Diagrama de arhitectură

Aplicația este dezolvată sub forma unei aplicații web cu client și server care comunică prin intermeriul cererilor HTTP. Serverul RESTful a fost creat folosind șablonul *Spring Boot*, modul lui de funcționare fiind foarte simplu. Acesta primește cereri HTTP de tipul *GET, PUT, POST* sau *DELETE*, pe care le îndeplinește apelând la baza de date prin intermediul unui *Repository*. Procesul de comunicare între client și server poate fii urmărit în Figura 10 [27]**.**

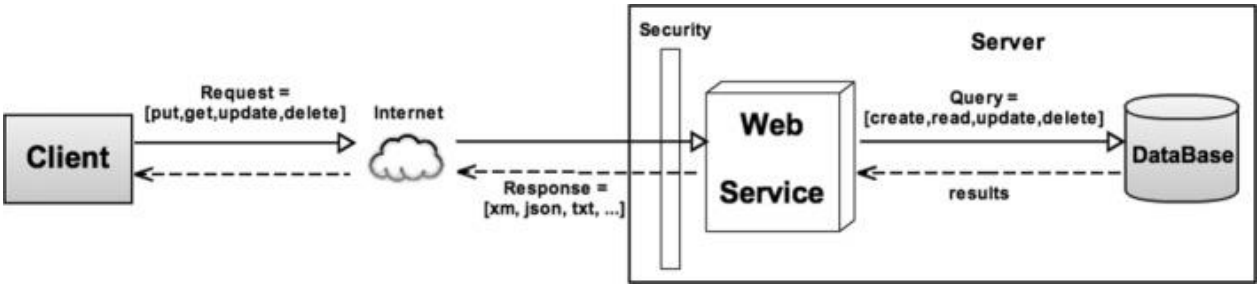

Figura 10: Arhitectura unui server RESTful

După cum se poate observa, după primirea rezultatelor din baza de date, serverul trimite resursele cerute la client , în cazul aplicației SpeedyTalk, sub formă de json.

Cererile *GET* sunt folosite pentru când dorim să extragem o resură, de exemplu când cerem serverului mesajele unui anumit utilizator, *PUT* pentru a modifica o resursă, când modificâm modelul antrenat conform mesajelor utilizatorului, *POST* pentru a crea o nouă resursă,

cum facem când un nou mesaj este trimis, iar cererile *DELETE* când se dorește ștergerea unei resurse.

4.1.2.2 Diagrama de bazei de date

Pentru persistarea datelor din aplicație am folosit SQLite, o bază de date relațională. Diagrama acesteia poate fii observată în Figura 11. Tabela *Users* conține câțiva utilizatori adăugați manual în prealabil. Un utilizator are un nume de utilizator (username) care trebuie să fie unic, după care este identificat , și o parolă (password), care va fii folositoare la o viitoare extindere a aplicației. Tabela *Users* este în relație One-to-Many cu tabela *Messages*, un utilizator putând să aibă mai multe mesaje trimise sau primite, dar un mesaj poate să aibă un singur destinatar sau expeditor. Câmpurile receiver, subject, text și attachment sunt completate de utilizator în momentul când acesta compune un mesaj, iar restul câmpurile sunt completate de program după cum urmează: sender este utilizatorul care este autentificat, date este data la care a fost trimis mesajul, iar status arată dacă mesajul a putut fii trimis cu success.

Modelul învățat e și acesta persistat ca să fie capabil să își continue antrenarea cu fiecare mesaj trimis de utilizator. Astfel tabelele *Prefix* și *States* păstrează parametrii învățați. Acestea se află într-o relație One-To-Many, un prefix putând să aibă mai multe stări care îl urmează. Prefixul este alcătuit din 2 cuvinte, excepție fac primul și al doilea cuvânt. Astfel, pentru primul cuvânt am stability prefixul (” ”, ” ”), iar pentru al doilea (” ”, *prefix1*). Câmpul count pentru *Prefix* reprezintă de câte ori a fost întâlnit acel prefix în tot textul, iar pentru *States* reprezintă de câte ori acea stare a urmat după prefixul din câmpul prefix. Acest câmp este util pentru a calcula mai ușor probabilitățile. La fiecare cuvânt citit se verifică dacă prefixul acestuia există în dicționar, dacă nu, se adaugă, altfel se incrementează doar evidența apariției stării, urmând ca la final să se actualizeze probabilitatea, după formula:

$$p = \frac{nr.apariții\ cuvânt\ după\ prefix}{nr.apariții\ prefix}$$

În tabela *Accuracy* am reținut un jurnal pentru a observa cum s-a dezvoltat acuratețea modelului pe mai multe date de testare.

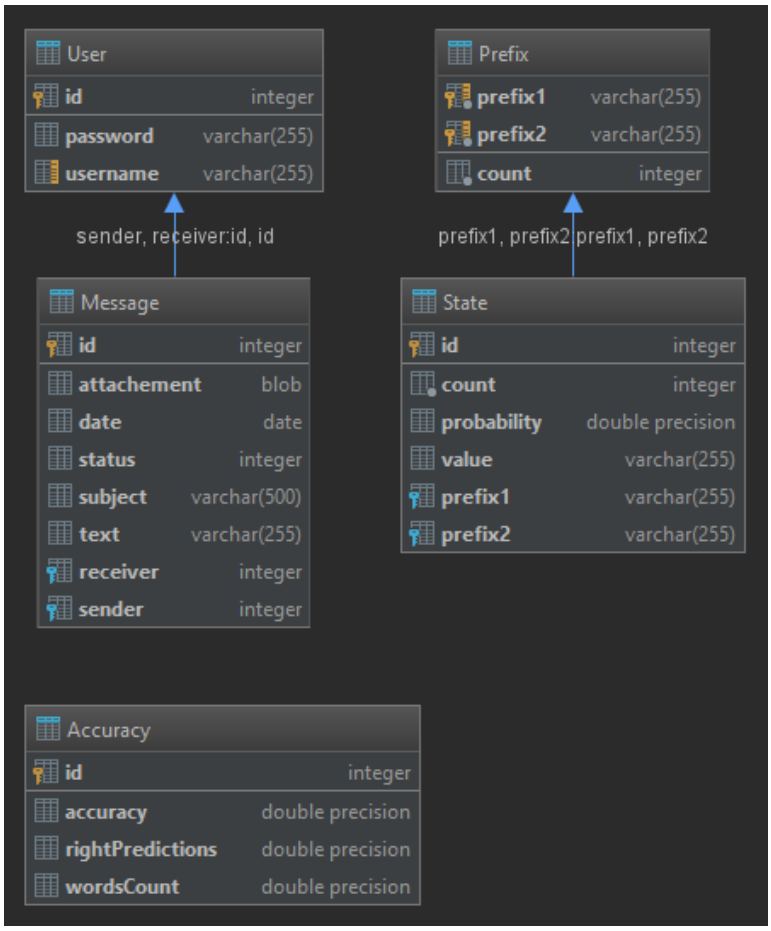

Figura 11. Diagrama bazei de date

4.1.2.3 Diagrama de clase

Structura entităților folosite pentru implementarea modelului Markov pot fii regăsite în Figura 12. Entitatea principală este *MarkovModel,* aceasta conținând metodele de antrenare și de testare a modelului. Folosind un dicționar pentru a reprezenta tranzițiile dintre stări, având ca și cheie *PrefixKey*, ce conține cele 2 cuvinte care alcătuiesc prefixul unei stări. La finalul antrenării, dicționarul a fost salvat în baza de date. Pentru antrenarea modelului se folosește clasa *DataProcessor* care conține metode de precesare a datelor de intrare, elimină semnele de punctuație care nu sunt pentru final de propoziție, separă prescurtările. Entitățile *Prefix* și *State* sunt parametrii modelului Markov cu stări ascunse.

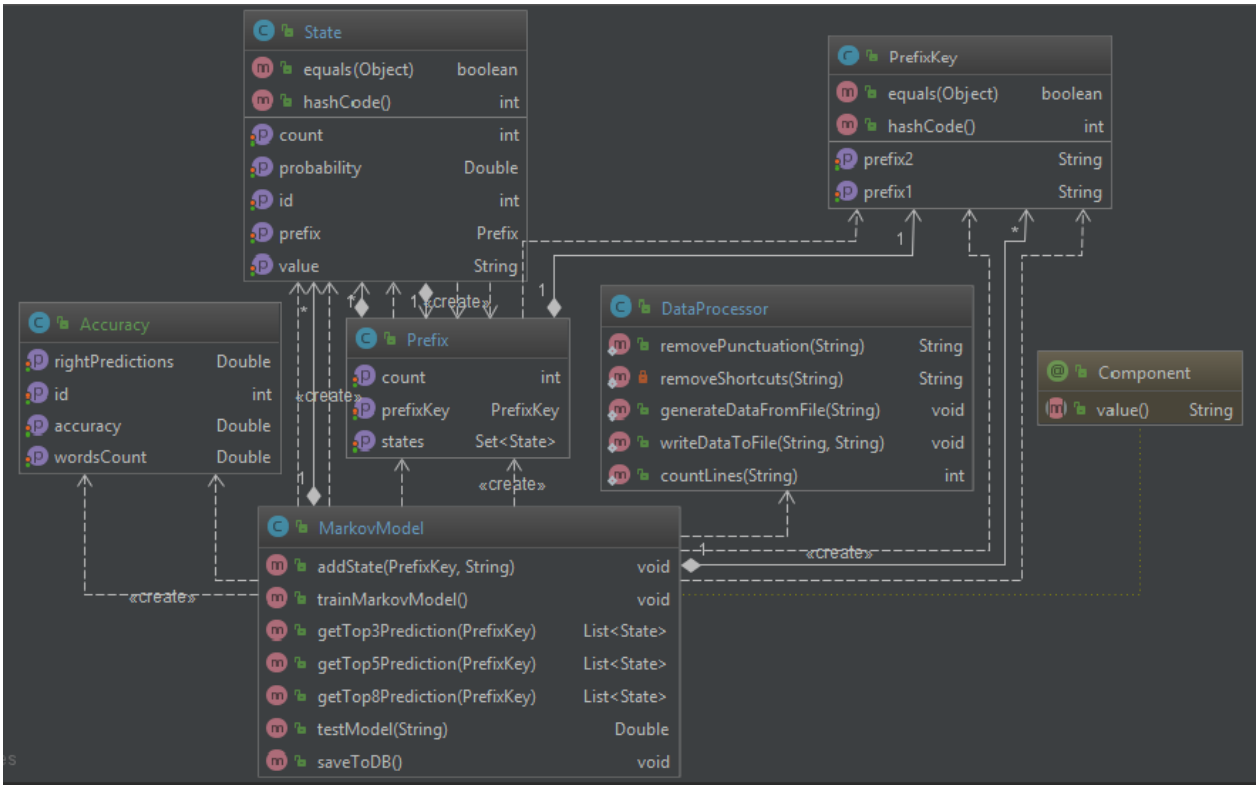

Figura 12. Diagrama de clase

4.1.2.4 Diagrama cazurilor de utilizare

Aplicația poate fii utilizată de orice tip de utilizator, de la cel mai experimentat până la cel cu ani de experiență. Funcționalitățile oferite sunt prezentate în diagram de utilizare din Figura 13.

Când deschide aplicația, utilizatorul poate alege dintr-un meniu oricare dintre funcționalitățile de mai sus. Mesajele primite/trimise sunt prezentate într-o listă din care utilizatorul poate să aleagă un mesaj pentru a-l vedea în întregime. Atunci când utilizatorul alege opțiunea de a scrie un mesaj nou, utilizatorul poate să scrie mesajul după care îl poate vedea în lista mesajelor trimise. În timpul scrierii mesajului, utilizatorului îi sunt afișate opțiune pentru autocompletare pe care acesta poate decide să le folosească.

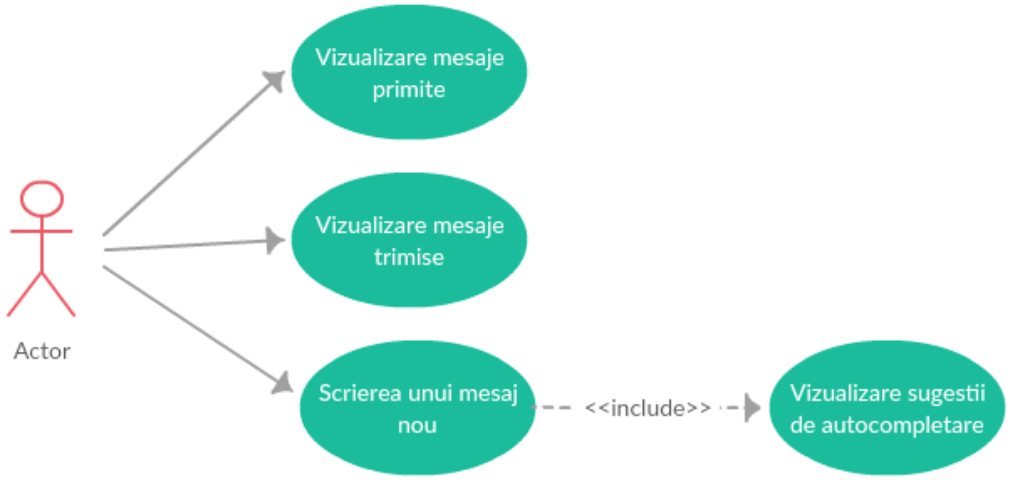

Figura 13. Diagrama cazurilor de utilizare

4.1.3 Implementare

Aplicația web SpeedyTalk este alcătuită dintr-un client implementat folosind Angular și un server implementat în Java. Datele sunt persistent într-o bază de date de tip SQLite.

Angular este unul dintre cele mai populare șabloane pentru aplicații web *single-page*, introdus de către Misko Hevery and Adam Abrons , ingineri la Google. Angular folosește arhitectura Model-View-Controller (MVC) . Codul este scris în TypeScript și compilat în JavaScript, limbaj dezvoltat de către Brendan Eich în 1995. O caracteristică importantă care face Angular așa de popular este sincronizarea între codul TypeScript și HTML ce permite vizualizarea modificărilor din HTML în variabilele de TypeScript și invers. Pentru stilizarea codului am folosit librăria Material din Angular și SCSS.

TypeScript este un limbaj de programare dezvolat și menținut de către Microsoft în 1 octombrie 2012, care compilează în JavaScript. Acesta a fost dezvoltat pentru dezvoltarea de aplicații mari. TypeScript este un superset peste JavaScript, ceea ce înseamnă că programele scrise în JavaScript sunt programe valide în TypeScript.

HTML (*HyperText Markup Language*) este un limbaj de marcare pentru aplicații web ce a fost dezvoltat de către World Wide Web Consortium.

Java este un limbaj de programare orientat pe obiecte inventat de către James Gosling și lansat pentru prima data de către Sun Microsystems în 1995. Java urmărește diviza *Write once, Run anywhere* (”Scrie o data, rulează peste tot”), codul compilat în Java poate rula pe orice platform care suportă Java. Pentru aplicație am folosit și Spring Boot care este un șablon ce facilitează dezvoltarea aplicațiilor Spring.

4.1.4 Manual de utilizare

După ce utilizatorul accesează aplicația SpeedyTalk, va fii întâmpinat de pagina de ”Bun venit” a aplicației din Figura 13. În partea de sus a ecranului se poate observa un meniu.

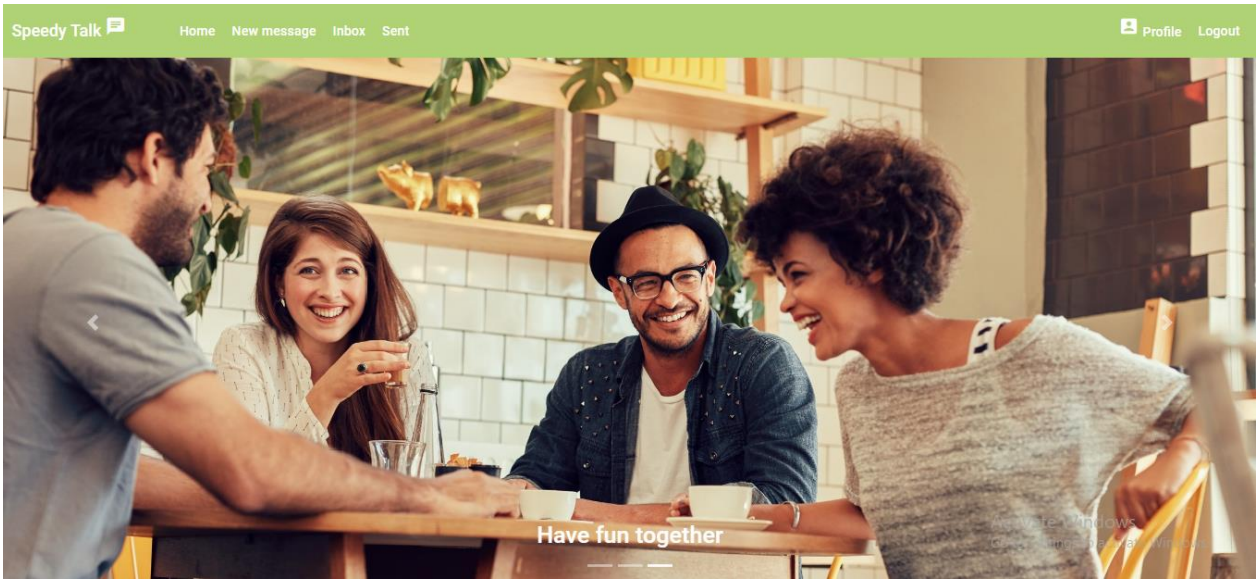

Figura 13. Pagina de pornire

Apăsând pe opțiunea *New message*, utilizatorul este direcționat spre pagina unde poate scrie compune un nou mesaj. Utilizatorul completează câmpurile *username, subject* și *text,*după care apasă pe butonul *Sent* pentru a trimite mesajul (Figura 15). Aici utilizatorul este asistat de către mecanismul de autocompletare în scrierea mesajului (Figura 14).

Pentru anularea acțiunii se apasă pe butonul *Cancel.*

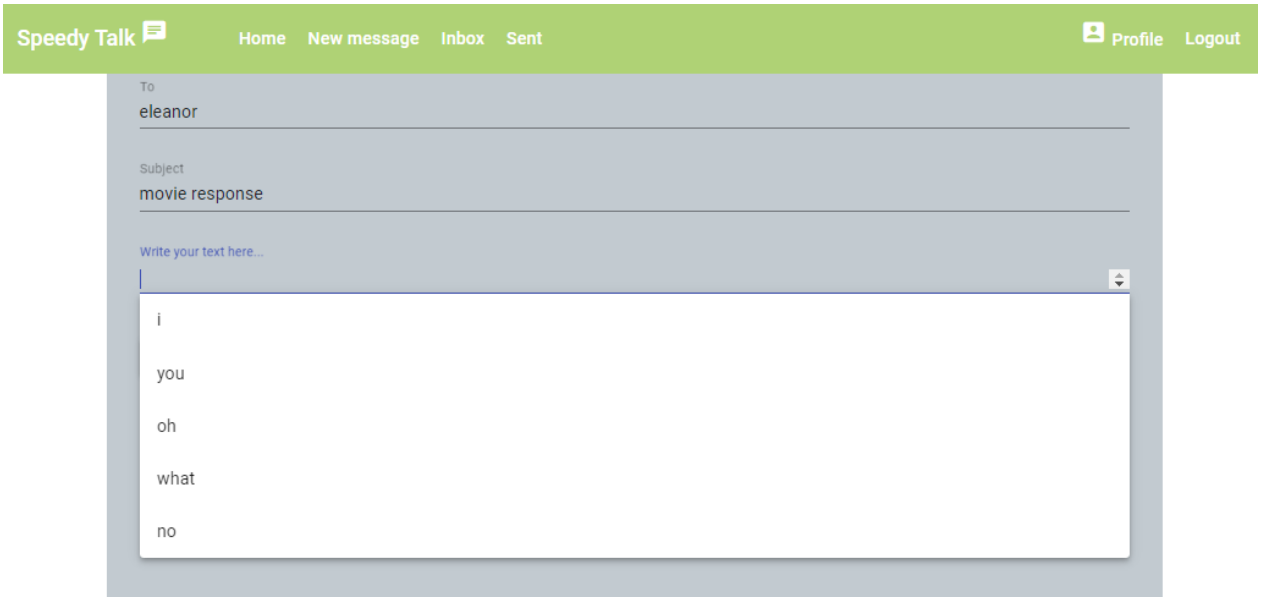

Figura 14. Sistemul de autocompletare

Figura 15. Completarea și trimiterea mesajului

Pentru a vizualiza mesajele primite, din meniul din partea de sus a ecranului se selectează opțiunea *Inbox*. Astfel, apare ecranul din Figura 16.

Pentru a vizualiza un mesaj se face click pe mesajul dorit, iar detaliile acestuia vor apărea în partea dreaptă a ecranului. Utilizatorul poate observa expeditorul, subiectul, textul și data expedierii mesajului după cum sevede în Figura 17.

Pentru a vizualiza mesajele trimise, din meniu se selectează opțiunea *Sent*, apoi se repeat pașii de la vizualizarea mesajelor primite.

Figura 16. Vizualizarea mesajelor

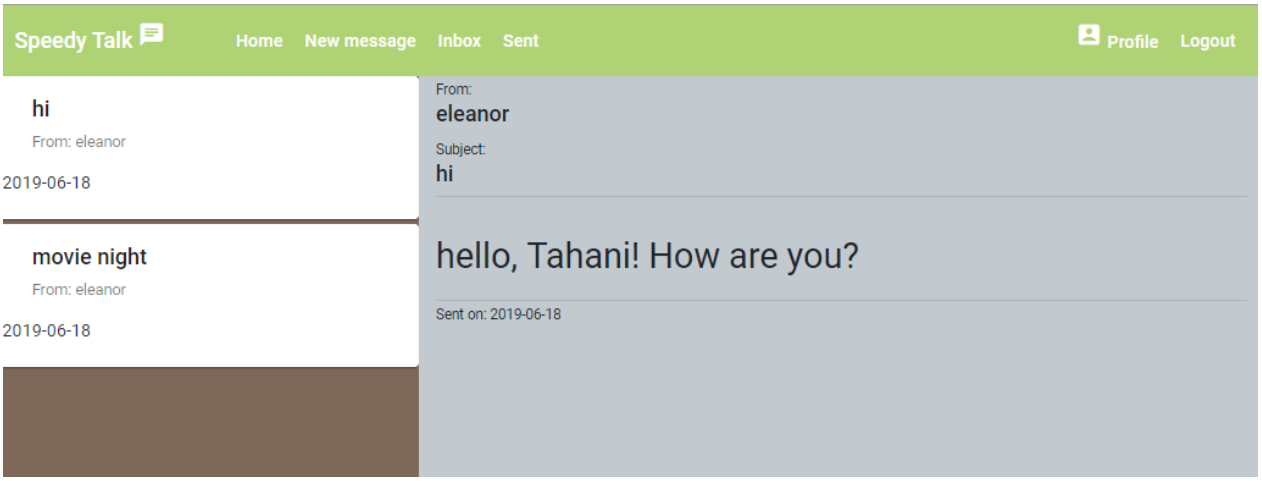

Figura 17. Vizualizarea detaliilor unui mesaj

4.2 Rezultate experimentale

4.2.1 Setul de date

Pentru a antrena modelul am ales să folosesc date în limba engleză, pentru simplitatea morfologică a limbii în comparative cu limba română. Pentru a observa mai bine cum lucrează modelul am antrenat modelul pe 3 seturi de date. Primul se de date este un set mai mic, de 39.647 de caractere și constă în conversații simple pe diverse teme luate de pe site-ul https://agendaweb.org/ . Al doilea set și cel mai important o reprezintă scenraiul complet al celor

3 sezoane ale serialului *The good place*, care are 747.981 de caractere, din care au fost eliminate indicațiile scenice. Al treilea set conține datele din primle 2 seturi la care s-a adăugat scenariul serialului *One day at a* time. În toate cazuri , datele sunt împărțite în date de antrenare (80%) și date de testare (20%).

Am ales să folosesc un scenariu de film deoarece este alcătuit în cea mai mare parte din dialog , iar acțiunea din serialele *The good place* și *One day at a time* este amplasată în prezent. Am considerat acest lucru important pentru aplicație deoarece , fiind o aplicație de email, o să fie folosită pentru conversații cu termini actuali.

4.2.1 Rezultate

Pentru a testa modelul am vrut să verificăm cât din textul din datele de test este prezis corect. Am definit o predicție ca fiind corectă dacă se află în primele *k* predicții făcute de model. Rezultatele obținute se pot observa în tabelul următor.

	Top 3			Top 5			Top 8		
	%	corecte	total	%	corecte	total	%	corecte	total
Setul 1	36,75	576	1567	40,39	633	1567	43,58	683	1567
Setul 2	26,18	7763	29650	30,19	8954	29650	33,99	10080	29650
Setul 3	25,57	13804	53966	29,76	16,065	53966	33,74	18213	53966

Top k indică faptul că am luat ca predictive corecte primele *k* cuvinte, *%* indică acuratețea *corecte* este numărul de predicții corecte realizate, iar *total* este numărul total de cuvinte.

Deși la prima vedere rezultatele nu par foarte bune, e de menționat faptul că este dificl să antrenezi un model să prezică un limbaj. Trebuie să se țină cont de multiplele cuvinte diferite care pot să apară o singură dată în text , cât și de propoziții formate dintr-un singur cuvânt sau chiar de posibilie greșeli de scriere. Considerând toate acestea , plus simplitatea modelului Markov cu stpri ascunse, putem spune că algoritmul a oferit rezultate bune. Mai mult, modelul prezice mai bine decât unele exemple întâlnite în literatură precum modelul cu n-grame propus de Jaysidh Dumbali și Nagaraja Rao A, care a reușit să atingă o acuratețe între 20.46% și 26.08% .[6] În

acest scop am implementat în aplicație o metodă ca modelul să continue să se antreneze , în funcție de textele trimise. Consider că acest mecanism este necesar , deoarece datele utilizate sunt destul de diverse ca și topică și limbaj, dar în conversațiile care pot avea loc în aplicație anumite șabloane pentru un utilizator se pot repeta mult mai des.

4.3 Extinderi posibile

Adăugarea de noi funcționalități aplicației SpeedyTalk ar îmbunătății aplicația și ar adduce noi experiențe plăcate utilizatorilor. Câteva dintre acestea ar putea fii următoarele:

- Includerea unor tehnici de procesare a limbajului natural pentru creșterea acurateții
- Personalizarea modelului de predicție pentru fiecare utilizator în parte, pentru a se model ape șabloanele limbajului fiecăruia
- Posibilitatea de a salva contactele favorite pentru a fii mai rapidă comunicarea
- Adăugarea unui mecanism de autentificare
- Sortarea mesajelor după anumite criteria alese de utilizator
- Căutarea unui mesaj după anumite cuvinte cheie
- Personalizarea profilelor utilizatorilor

CONCLUZII

Aplicațiile care folosesc inteligența artificială sunt din ce în ce mai întâlnite, atât în aplicațiile complexe, cât și în cele care pot să dea impresia unui utilizator neexperimentat că sunt banale. Lumea a arătat un interes crescând pentru învățarea automata, care permite utilizatorilor o comunicare facilă cu lumea și tehnologia. Din acest motiv, am ales să prezint o temă axată pe învățarea supervizată și să implementez o aplicație care să pună în evidență această tehnică, care eficientizează scrierea unui mesaj prin predicție.

Aplicația SpeedyTalk este o aplicație web pe care orice utilizator o poate accesa pentru a comunica prin mesaje cu alți utilizatori. La baza aplicației stă limbajul Java și șablonul Spring pentru partea de server, iar pentru partea de client Angular și , implicit, TypeScript și HTML. Algoritmul pentru predicția textelor a fost implementat în Java, acesta analizează un volum mare de text și determină stările și tranzițiile modelului Markov cu stări ascunse, ca pe baza acestuia să realizeze predicții automate pe parcurs ce utilizatorul compune mesajul.

BIBLIOGRAFIE

[1] Aldwin N. , *What is Angular? – A Beginner's Guider*. Disponibil pe hostinger.com

[2] Blunsom, Phil. *Hidden markov models.* Lecture notes, *August* 15.18-19 : 48, 2004

[3] Chung, Junyoung, et al. *Empirical evaluation of gated recurrent neural networks on sequence modeling.* arXiv preprint arXiv:1412.3555 , 2014.

[4] Das Subhasis, Chinmayee Shah. *Contextual Code Completion Using Machine Learning*, 2015.

[5] Eddy, S. R., *Hidden Markov models. Current Opinion in Structural Biology*, 6(3), 361–365. doi:10.1016/s0959-440x(96)80056-x , 1996

[6] Jaysidh Dumbali, Nagaraja Rao A. , *Real Time Word Prediction Using N-grams Model*, International Journal of Innovative Technology and Exploring Engineering, 2019

[7] Jurafsky, Daniel & Martin H. James , *Speech and Language Processing*, capitolul 8, *Hidden Markov Models*, 2018

[8] Kim, Yoon, et al. *Character-aware neural language models*. In: Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[9] Kostadinov, Simeon, *Understanding GRU Networks.* Disponibil pe towardsdatascience.com/ , 2017

[10] Kröse, Ben, et al. , *An introduction to neural networks*, 1993

[11] Lipton, Zachary C., John Berkowitz, and Charles Elkan. *A critical review of recurrent neural networks for sequence learning*. arXiv preprint arXiv:1506.00019 , 2015.

[12] Luong, Minh-Thang et al. *Effective approaches to attention-based neural machine translation*. arXiv preprint arXiv:1508.04025 , 2015

[13] Ma, Edward, *Besides Word Embedding, why you need to know Character Embedding?*. Disponibil pe towardsdatascience.com/ , 2018

[14] Maad M. Mijwel, *Artificial Neural Networks Advantages and Disadvantages.* Disponibil pe https://www.linkedin.com , 2018

[15] Mikolov, Tomas, et al. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* , 2013.

[16] Moawad, Assaad, *The magic of LSTM neural networks.* Disponibil pe https://medium.com , 2018

[17] Olah, Christopher, *Understanding LSTM Networks.* Disponibil pe https://colah.github.io/, 2015

[18] Porumbel Valentin, *Recunoaștere vocală minimală pe sisteme embedded folosind inteligența artificială*, Universitatea ”Politehnică” din București, 2016

[19] Raffel, Colin, and Daniel PW Ellis. *Feed-forward networks with attention can solve some long-term memory problems*, arXiv preprint arXiv:1512.08756 , 2015.

[20] Ratatype , https://www.ratatype.com/learn/average-typing-speed/ , 2019.

[21] Srivastava, Rupesh Kuma et al. *Highway networks*. arXiv preprint arXiv:1505.00387 ,2015.

[22] Sutskever, Ily et al. *Sequence to sequence learning with neural networks*. In *Advances in neural information processing systems*, 2014.

[23] Tsang, Sik-Ho, *Review: Highway Networks—Gating Function To Highway (Image Classification).* Disponibil pe towardsdatascience.com/, 2019

[24]Tushar, Gupta, *Deep Learning: Feedforward Neural Network.* Disponibil pe towardsdatascience.com/, 2017

[25] van Handel, Ramon. *Hidden markov models* . Unpublished lecture notes ,2008.

[26] Vikas, Gupta, *Understanding Feedforward Neural Network*, Disponibil pe learnopencv.com/ , 2017

[27] Yank Jack, *JACOB: An Enterprise Framework for Computational Chemistry*, Disponibil pe researchgate.net , 2013